C++

DEVELOPMENT

THIS DOCUMENT DESCRIBES GOOD PRACTICES AND METHODS FOR C++ DEVELOPMENT. ALL THE INFO HERE IS GATHERED FROM DIFFERENT SOURCES AND I THE WRITER OF THIS DOCUMENT HAVE TRIED TO MADE THE REFERANCES TO THE ORIGINAL SOURCES AS ACCURATE AS POSSIBLE. HOPE THIS HELPS SOMEONE AND IF THERE ARE PROBLEMS OR ERROR WITH THIS DOCUMENT YOU CAN DO IT THROUGH MY BLOG AND I WILL TRY TO DO MY BEST TO FIX THEM. USE THIS INFORMATION AS YOU SEE FIT AT YOUR OWN DISCRETION.

THIS DOCUMENT WAS MADE BY ADRIAN SIMIONESCU 4.12.2014 – VERSION 1.2 - HTTP://LIONADI.WORDPRESS.COM/

VERSION HISTORY:

VERSION	HISTORY	DATE CREATED
1.0	INITIAL VERSION	31.10.2014
1.1	ADDED MODERN C++11/14 GUIDES	14.11.2014
1.2	ADDED MORE MODERN C++ SOURCES OF KNOWLEDGE TO EXPLAIN BETTER NEW FUNCTIONALITIES	4.12.2014

C++ GUIDELINES AND RECOMMENDED PRACTICES

CONTENTS

Optimization 6
80 – 20 Rule 6
Important C++ Keywords and Concepts 6
Casting 7
New style casting 7
C++ Basics – Part 1 8
format specifiers 8
Statements and expressions 11
Sample 11
Comments 12
Variables 12
Control flow 15
Functions 16
Pass by value 18
Pass by reference 18
Pass by address 20
Inline functions 23
Function Pointers 24
Handling errors 27
The stack and the heap 35
Operators and Expressions 39
Preprocessors 40
Basic addressing and variable declaration 41
Keywords and naming identifiers 41
Precedence and associativity 42
Pointers 45

Classes 47
Inheritance and access specifiers 50
This pointer 52
Constructor 56
Destructors 57
Const class objects and member functions 57
Virtual table 70
Templates 71
Exceptions 72
The Standard Template Library 72
C++ Basics – Part 2 72
Prefer consts, enums, and inlines to #defines 73
#define 73
Header constants 73
Class-specific constants 73
prefer inline functions to #defines 74
Use const whenever possible 74
Make sure that objects are initialized before they're used 74
Constructors, Destructors, and Assignment Operators 75
Declare destructors virtual in polymorphic base classes 76
Prevent exceptions from leaving destructors 77
Handle assignment to self in operator= 78
Resource Management 79
Use objects to manage resources 80
Store newed objects in smart pointers in standalone statements 80
Designs and Declarations 81
Treat class design as type design 81
Don't try to return a reference when you must return an object 83
Declare non-member functions when type conversions should apply to all parameters 83
Implementations 84
Exception-safe functions 85
Understand inlining 86
Implicit inline example 86
Explicit inline example 86
Notes on inlining 86

Minimize compilation dependencies between files ... 86
Inheritance and Object-Oriented Design ... 87
Make sure public inheritance models "is-a." ... 88
Differentiate between inheritance of interface and inheritance of implementation ... 88
Designing classes ... 88
Common mistakes ... 89
Alternatives to virtual functions ... 89
Never redefine an inherited non-virtual function ... 89
Never redefine a function's inherited default parameter value ... 91
Model "has-a" or "is-implemented-in-terms-of" through composition ... 91
Example of has-a relationship ... 91
Example of is-implemented-in-terms-of ... 92
Use private inheritance judiciously ... 93
How to choose between is-implemented-in-terms-of of private inheritance and composition ... 93
Modern C++ (C++11, C++14) ... 93
The Biggest Changes in C++11 (and Why You Should Care) ... 93
Lambda Expressions ... 95
Automatic Type Deduction and `decltype` ... 96
Uniform Initialization Syntax ... 97
Deleted and Defaulted Functions ... 98
nullptr ... 99
Delegating Constructors ... 100
Rvalue References ... 100
C++11 Standard Library ... 102
Terminology ... 104
Rvalues and lvalues ... 104
Function calls, Arguments and Parameters ... 129
Exception-safe ... 129
Function object ... 129
Lambda expression, closures and Templates ... 129
Declarations, Definitions and Signatures ... 130
Pointers ... 130
Offset ... 130
Strong and weak typing ... 131
Deducting types ... 131

Template type deductions 132
auto type deduction 132
auto 132
C++11 Tutorial: Let Your Compiler Detect the Types of Your Objects Automatically 133
C++11 Tutorial: Let Your Compiler Detect the Types of Your Objects Automatically 141
Modern C++ functionalities 149
Use C++11 Inheritance Control Keywords to Prevent Inconsistencies in Class Hierarchies – OVERRIDE, FINAL 150
Interfaces done right 156
Declare overriding functions *override* 162
Declare function noexcept if they won't emit exceptions 164
contexpr 165
Understand special member function generation 165
Using constexpr to Improve Security, Performance and Encapsulation in C++ 167
C++11 Tutorial: New Constructor Features that Make Object Initialization Faster and Smoother 175
Smart Pointers 183
C++11 Smart Pointers 184
Prefer std::make_unique and std::make_shared to direct user of new 195
Questions and answers about smart pointers 196
Points on pointers 203
LESSON #4: SMART POINTERS 204
Points on raw pointers 214
Use **std::unique_ptr** for exclusive-ownershipresource management. 216
Use **std::shared_ptr** for shared-ownershipresource management 216
Use **std::weak_ptr** for **std::shared_ptr**likepointers that can dangle 223
Prefer std::make_unique and std::make_shared to direct use of new 223
When using the Pimpl Idiom, define specialmember functions in the implementation file 225
rvalue references, move semantics and perfect forwarding 226
C++11 Tutorial: Introducing the Move Constructor and the Move Assignment Operator 228
Rule of three (C++ programming) 236
Universal Reference 240
Operators overloading and std::move 259
On the Superfluousness of std::move 262
Difference between std::move and std::forward 265
A Brief Introduction to Rvalue References 267
The Drawbacks of Implementing Move Assignment in Terms of Swap 275

Understand std::move and std::forward. 277
C++11: Perfect forwarding - Explained 280
C++: Smart pointers – Explained 291
Lambda expressions 316
std::bind explained (if needed) 316
C++11 Tutorial: Lambda Expressions — The Nuts and Bolts of Functional Programming 323
Lambda Functions in C++11 - the Definitive Guide 330
A Note About Function Pointers 337
C++11 lambda to function pointer or std::function 339
Concurrency API 346
New since End of 2014 346
Several C++ singleton implementations 346
A Glimpse into C++14: Combine Flexibility and Performance with Dynamic Arrays and Runtime-Sized Arrays 356
C++ memory management and vectors 363
References 363

BASICS

OPTIMIZATION

80 – 20 RULE

Don't forget the empirically determined rule of 80-20, which states that atypical program spends 80% of its time executing only 20% of its code. It's an important rule, because it reminds you that your goal as a software developer is to identify the 20% of your code that can increase your program's overall performance.

IMPORTANT C++ KEYWORDS AND CONCEPTS

Inline

Mutable

Typedef

tr1::shared_ptr

using

In order to determine the size of data types on a particular machine, C++ provides an operator named sizeof. The **sizeof operator** is a unary operator that takes either a type or a variable, and returns its size in bytes.

empty base optimization (EBO)?

"A **buffer** is memory set aside temporarily to hold data. In this case, we're temporarily holding the user input before we write it out using cout.

If the user were to enter more characters than our array could hold, we would get a buffer overflow. A **buffer overflow** occurs when the program tries to store more data in a buffer than the buffer can hold. Buffer overflow results in other memory being overwritten, which usually causes a program crash, but can cause any number of other issues." (LearnCPP, 2014)

Explicit

"Using the explicit keyword, a constructor is declared to be "nonconverting", and explicit constructor syntax is required:

```
class A {
        public:
                explicit A(int);
        };

        void f(A) {}

        void g()
        {
                A a1 = 37;      // illegal

                A a2 = A(47);   // OK

                A a3(57);       // OK

                a1 = 67;        // illegal

                f(77);          // illegal
        }
```

" http://www.glenmccl.com/tip_023.htm

CASTING

(T) expression // cast expression to be of type T(old style)

NEW STYLE CASTING

const_cast<T>(expression)

dynamic_cast<T>(expression)

reinterpret_cast<T>(expression)

static_cast<T>(expression)

- const_cast is typically used to cast away the constness of objects. It is the only C++-style cast that can do this.
- dynamic_cast is primarily used to perform "safe downcasting," i.e., to determine whether an object is of a particular type in an inheritance hierarchy. It is the only cast that cannot be performed using the old-style syntax. It is also the only cast that may have a significant runtime cost.
- reinterpret_cast is intended for low-level casts that yield implementation-dependent (i.e., unportable) results, e.g., casting a pointer to an int. Such casts should be rare outside low-level code.
- static_cast can be used to force implicit conversions (e.g., non-const object to const object, int to double, etc.). It can also be used to perform the reverse of many such conversions (e.g., void* pointers to typed pointers, pointer-to-base to pointer-to-derived), though it cannot cast from const to non-const objects. (Only const cast can do that.)

C++ BASICS – PART 1

FORMAT SPECIFIERS

C string that contains the text to be written to stdout.
It can optionally contain embedded *format specifiers* that are replaced by the values specified in subsequent additional arguments and formatted as requested.

A *format specifier* follows this prototype: [see compatibility note below]
`%[flags][width][.precision][length]specifier`

Where the *specifier character* at the end is the most significant component, since it defines the type and the interpretation of its corresponding argument:

specifier	**Output**	**Example**
`d` *or* `i`	Signed decimal integer	`392`
`u`	Unsigned decimal integer	`7235`
`o`	Unsigned octal	`610`
`x`	Unsigned hexadecimal integer	`7fa`
`X`	Unsigned hexadecimal integer (uppercase)	`7FA`
`f`	Decimal floating point, lowercase	`392.65`
`F`	Decimal floating point, uppercase	`392.65`

e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.	
%	A % followed by another % character will write a single % to the stream.	%

The *format specifier* can also contain sub-specifiers: *flags*, *width*, *.precision* and *modifiers* (in that order), which are optional and follow these specifications:

flags	**description**
-	Left-justify within the given field width; Right justification is the default (see *width* sub-specifier).
+	Forces to preceed the result with a plus or minus sign (+ or −) even for positive numbers. By default, only negative numbers are preceded with a − sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceeded with 0, 0x or 0X respectively for

	values different than zero. Used with `a`, `A`, `e`, `E`, `f`, `F`, `g` or `G` it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
`0`	Left-pads the number with zeroes (`0`) instead of spaces when padding is specified (see *width* sub-specifier).

width	**description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The *width* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	**description**
.number	For integer specifiers (`d`, `i`, `o`, `u`, `x`, `X`): *precision* specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A *precision* of `0` means that no character is written for the value `0`. For `a`, `A`, `e`, `E`, `f` and `F` specifiers: this is the number of digits to be printed **after** the decimal point (by default, this is 6). For `g` and `G` specifiers: This is the maximum number of significant digits to be printed. For `s`: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for *precision*, `0` is assumed.
.*	The *precision* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

The *length* sub-specifier modifies the length of the data type. This is a chart showing the types used to interpret the corresponding arguments with and without *length* specifier (if a different type is used, the proper type promotion or conversion is performed, if allowed):

	specifiers						
length	**d i**	**u o x X**	**f F e E g G a A**	**c**	**s**	**p**	**n**

(none)	int	unsigned int	double	int	char*	void*	int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

Note regarding the `c` specifier: it takes an `int` (or wint_t) as argument, but performs the proper conversion to a`char` value (or a `wchar_t`) before formatting it for output.

Note: Yellow rows indicate specifiers and sub-specifiers introduced by C99.
See <cinttypes> for the specifiers for extended types.

`... (additional arguments)`

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a value to be used to replace a *format specifier* in the *format* string (or a pointer to a storage location, for `n`).
There should be at least as many of these arguments as the number of values specified in the *format specifiers*. Additional arguments are ignored by the function.

STATEMENTS AND EXPRESSIONS

"The most common type of instruction in a program is the **statement**. A statement in C++ is the smallest independent unit in the language. In human language, it is analogous to a sentence. We write sentences in order to convey an idea. In C++, we write statements in order to convey to the compiler that we want to perform a task. **Statements in C++ are terminated by a semicolon**." (LearnCPP, 2014)

SAMPLE

```
int x;
x = 5;
cout << x;
```

"`int x` is a **declaration statement**. It tells the compiler that x is a variable. All variables in a program must be declared before they are used. We will talk more about variables shortly. `x = 5` is an **assignment statement**. It assigns a value (5) to a variable (x). `cout << x;` is an output statement. It outputs the value of x (which we set to 5 in the previous statement) to the screen.

The compiler is also capable of resolving expressions. An **expression** is an mathematical entity that evaluates to a value. For example, the statement `x = 2 + 3;` is a valid assignment statement. The expression 2+3 evaluates to the value of 5. This value of 5 is then assigned to x." (LearnCPP, 2014)

"In C++, statements are typically grouped into units called functions. A **function** is a collection of statements that executes sequentially." (LearnCPP, 2014)

"Libraries are groups of functions that have been "packaged up" for reuse in many different programs. The core C++ language is actually very small and minimalistic — however, C++ comes with a bunch of libraries, known as the C++ standard libraries, that provide programmers with lots of extra functionality. For example, the iostream library contains functions for doing input and output. During the link stage of the compilation process, the libraries from the C++ standard library are the runtime support libraries that are linked into the program." (LearnCPP, 2014)

return statement

COMMENTS

```
// this is a single line comment

/* This is a multi-line comment.
   This line will be ignored.
   So will this one. */

/* This is a multi-line comment.
 * the matching asterisks to the left
 * can make this easier to read
 */
```

VARIABLES

"A **variable** in C++ is a name for a piece of memory that can be used to store information. You can think of a variable as a mailbox, or a cubbyhole, where we can put and retrieve information. All computers have memory, called RAM (random access memory), that is

available for programs to use. When a variable is declared, a piece of that memory is set aside for that variable." (LearnCPP, 2014)

"In order to declare a variable, we generally use a **declaration statement**." (LearnCPP, 2014)

"An **l-value** is a value that has an address (in memory). Since all variables have addresses, all variables are l-values. They were originally named l-values because they are the only values that can be on the left side of an assignment statement. When we do an assignment, the left hand side of the assignment operator must be an l-value." (LearnCPP, 2014)

"The opposite of l-values are r-values. An **r-value** refers to any value that can be assigned to an l-value. r-values are always evaluated to produce a single value." (LearnCPP, 2014)

"A variable that has not been assigned a value is called an **uninitialized variable**. Uninitialized variables are very dangerous because they cause intermittent problems (due to having different values each time you run the program). This can make them very hard to debug. " (LearnCPP, 2014)

"A block of statements, also called a **compound statement**, is a group of statements that is treated by the compiler as if it were a single statement. Blocks begin with a { symbol, end with a } symbol, and the statements to be executed are placed in between. Blocks can be used any place where a single statement is allowed.

Blocks can be nested inside of other blocks. As you have seen, the *if statement* executes a single statement if the condition is true. However, because blocks can be used anywhere a single statement can, we can instead use a block of statements to make the *if statement* execute multiple statements if the condition is true!

A variable's **scope** determines who can see the variable, and how long it lives for. Variables declared inside a block are called **local variables**, and local variables have **block scope** (also called local scope). Variables with block scope can be accessed only within the block that they are declared in, and are destroyed as soon as the block ends.

If a variable is only used in a single block, declare it within that block.

Variables declared outside of a block are called **global variables**. Global variables have **program scope**, which means they can be accessed everywhere in the program, and they are only destroyed when the program ends.

A variable with **file scope** can be accessed by any function or block within a single file. To declare a file scoped variable, simply declare a variable outside of a block (same as a global variable) but use the **static** keyword.

File scoped variables act exactly like global variables, except their use is restricted to the file in which they are declared (which means you can not extern them to other files). File scoped variables are not seen very often in C++ because they have most of the downsides of global variables, just on a smaller scale.

By default, local variables have **automatic duration**, which means they are destroyed when the block they are declared in goes out of scope. You can explicitly declare a variable as having automatic duration by using the **auto** keyword, though this is practically never done because local variables are automatic by default, and it would be redundant.

Using the static keyword on local variables changes them from automatic duration to fixed duration (also called static duration). A **fixed duration** variable is one that retains it's value even after the scope in which it has been created has been exited! Fixed duration variables are only created (and initialized) once, and then they are persisted throughout the life of the program.

Often it is the case that data needs to be converted from one type to another type. This is called **type conversion**.

Implicit type conversion is done automatically by the compiler whenever data from different types is intermixed. When a value from one type is assigned to another type, the compiler implicitly converts the value into a value of the new type. For example:

```
1  double dValue = 3; // implicit conversion to double value 3.0
2  int nValue = 3.14156; // implicit conversion to integer value 3
```

When evaluating expressions, the compiler breaks each expression down into individual subexpressions. Typically, these subexpressions involve a unary or binary operator and some operands. Most binary operators require their operands to be of the same type. If operands of mixed types are used, the compiler will convert one operand to agree with the other. To do this, it uses a heirarchy of data types:

Long double (highest)
Double
Float
Unsigned long int
Long int
Unsigned int
Int (lowest)

A good question is, "why is integer at the bottom of the tree? What about char and short?". Char and short are always implicitly promoted to integers (or unsigned integers) before evaluation. This is called**widening**.

Casting represents a request by the programmer to do an explicit type conversion. In standard C programming, casts are done via the () operator, with the name of the type to cast to inside.

Typedefs allow the programmer to create an alias for a data type, and use the aliased name instead of the actual type name. To declare a typedef, simply use the **typedef** keyword, followed by the type to alias, followed by the alias name:

```
typedef long miles; // define miles as an alias for long

// The following two statements are equivalent:
long nDistance;
miles nDistance;
```

C++ allows us to create our own user-defined aggregate data types. An **aggregate data type** is a data type that groups multiple individual variables together. One of the simplest aggregate data type is the struct. A **struct** (short for structure) allows us to group variables of mixed data types together into a single unit.

Member variables: In order to access the individual members, we use the **member selection operator** (which is a period). As with normal variables, struct member variables are not initialized, and will typically contain junk. We must initialize them manually.

Initializing structs member by member is a little cumbersome, so C++ supports a faster way to initialize structs using an **initializer list**. This allows you to initialize some or all the members of a struct at declaration time." (LearnCPP, 2014)

CONTROL FLOW

"When a program is run, the CPU begins execution at the top of main(), executes some number of statements, and then terminates at the end of main(). The sequence of statements that the CPU executes is called the program's **path**. Most of the programs you have seen so far have been **straight-line programs**. Straight-line programs have **sequential flow** — that is, they take the same path (execute the same statements) every time they are run (even if the user input changes).

Fortunately, C++ provides **control flow statements** (also called FLOW CONTROL STATEMENTS), which allow the programmer to change the CPU's path through the program. There are quite a few different types of control flow statements, so we will cover them briefly here, and then in more detail throughout the rest of the section.

The most basic control flow statement is the **halt**, which tells the program to quit running immediately. In C++, a halt can be accomplished through use of the exit() function that is defined in the cstdlib header. The exit function takes an integer parameter that is returned to the operating system as an exit code, much like the return value of main().

The next most basic flow control statement is the jump. A **jump** unconditionally causes the CPU to jump to another statement. The GOTO, BREAK, and CONTINUE keywords all cause

different types of jumps — we will discuss the difference between these in upcoming sections.

Function calls also cause jump-like behavior. When a function call is executed, the CPU jumps to the top of the function being called. When the called function ends, execution returns to the statement after the function call.

A **conditional branch** is a statement that causes the program to change the path of execution based on the value of an expression. The most basic conditional branch is an IF STATEMENT, which you have seen in previous examples.

A **loop** causes the program to repeatedly execute a series of statements until a given condition is false.

Exceptions offer a mechanism for handling errors that occur in functions. If an error occurs that the function can not handle, it can raise an exception, and control jumps to the nearest block of code that has declared it is willing to catch exceptions of that type. Exception handling is a fairly advanced feature of C++." (LearnCPP, 2014)

FUNCTIONS

"A **function** is a sequence of statements designed to do a particular job.

A program will be executing statements sequentially inside one function when it encounters a function call. A **function call** is an expression that tells the CPU to interrupt the current function and execute another function. The CPU "puts a bookmark" at the current point of execution, and then **calls** (executes) the function named in the function call. When the called function terminates, the CPU goes back to the point it bookmarked, and resumes execution.

Functions you write can return a single value to their caller as well. We do this by changing the return type of the function in the function's declaration. A return type of **void** means the function does not return a value. A return type of int means the function returns an integer value to the caller.

In the return values subsection, you learned that a function can return a value back to the caller. **Parameters** are used to allow the caller to pass information to a function! This allows functions to be written to perform generic tasks without having to worry about the specific values used, and leaves the exact values of the variables up to the caller." (LearnCPP, 2014)

"In a **forward declaration**, we declare (but do not define) our function in advance of where we use it, typically at the top of the file. This way, the compiler will understand what our function looks like when it encounters a call to it later. We do this by writing a declaration statement known as a function prototype. A **function prototype** is a declaration of a function that includes the function's name, parameters, and return type, but does not implement the function." (LearnCPP, 2014)

"**Parameters vs Arguments**

In common usage, the terms parameter and argument are often interchanged.

A **function parameter** is a variable declared in the prototype or declaration of a function:

```
1  void foo(int x); // prototype -- x is a parameter
2
3  void foo(int x) // declaration -- x is a parameter
4  {
5  }
```

An **argument** is the value that is passed to the function in place of a parameter:

```
1  foo(6); // 6 is the argument passed to parameter x
2  foo(y+1); // the value of y+1 is the argument passed to parameter x
```

When a function is called, all of the parameters of the function are created as variables, and the value of the arguments are copied into the parameters. For example:

```
1  void foo(int x, int y)
2  {
3  }
4
5  foo(6, 7);
```

When foo() is called with arguments 6 and 7, foo's parameter x is created and assigned the value of 6, and foo's parameter y is created and assigned the value of 7.

Even though parameters are not declared inside the function block, function parameters have local scope. This means that they are created when the function is invoked, and are destroyed when the function block terminates:

```
1  void foo(int x, int y) // x and y are created here
2  {
3  } // x and y are destroyed here
```

There are 3 primary methods of passing arguments to functions: pass by value, pass by reference, and pass by address.

By default, arguments in C++ are passed by value. When arguments are **passed by value**, a copy of the argument is passed to the function.

PASS BY VALUE

"When passing arguments by value, the only way to return a value back to the caller is via the function's return value. While this is suitable in many cases, there are a few cases where better options are available. One such case is when writing a function that needs to modify the values of an array (eg. sorting an array). In this case, it is more efficient and more clear to have the function modify the actual array passed to it, rather than trying to return something back to the caller.

```
void foo(int y)
{
    using namespace std;
    cout << "y = " << y << endl;
}

int main()
{
    foo(5); // first call

    int x = 6;
    foo(x); // second call
    foo(x+1); // third call

    return 0;
}
```

Advantages of passing by value:

- Arguments passed by value can be variables (eg. x), literals (eg. 6), or expressions (eg. x+1).
- Arguments are never changed by the function being called, which prevents side effects.

Disadvantages of passing by value:

- Copying large structs or classes can take a lot of time to copy, and this can cause a performance penalty, especially if the function is called many times." (LearnCPP, 2014)

PASS BY REFERENCE

"One way to allow functions to modify the value of argument is by using pass by reference. In **pass by reference**, we declare the function parameters as references rather than normal variables. When the function is called, y will become a reference to the argument. Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument!

Pass by const reference

One of the major disadvantages of pass by value is that all arguments passed by value are copied to the parameters. When the arguments are large structs or classes, this can take a lot of time. References provide a way to avoid this penalty. When an argument is passed by reference, a reference is created to the actual argument (which takes minimal time) and no copying of values takes place. This allows us to pass large structs and classes with a minimum performance penalty.

However, this also opens us up to potential trouble. References allow the function to change the value of the argument, which in many cases is undesirable. If we know that a function should not change the value of an argument, but don't want to pass by value, the best solution is to pass by const reference.

You already know that a const reference is a reference that does not allow the variable being referenced to be changed. Consequently, if we use a const reference as a parameter, we guarantee to the caller that the function will not (and can not) change the argument!

The following function will produce a compiler error:

```
1  void foo(const int &x)
2  {
3      x = 6;  // x is a const reference and can not be changed!
4  }
```

Using const is useful for several reasons:

- It enlists the compilers help in ensuring values that shouldn't be changed aren't changed.
- It tells the coder whether they need to worry about the function changing the value of the argument
- It helps the coder debug incorrect values by telling the coder whether the function might be at fault or not

Rule: Always pass by const reference unless you need to change the value of the argument

Summary

Advantages of passing by reference:

- It allows us to have the function change the value of the argument, which is sometimes useful.
- Because a copy of the argument is not made, it is fast, even when used with large structs or classes.
- We can pass by const reference to avoid unintentional changes.
- We can return multiple values from a function.

Disadvantages of passing by reference:

- Because a non-const reference can not be made to a literal or an expression, reference arguments must be normal variables.
- It can be hard to tell whether a parameter passed by reference is meant to be input, output, or both.
- It's impossible to tell from the function call that the argument may change. An argument passed by value and passed by reference looks the same. We can only tell whether an argument is passed by value or reference by looking at the function declaration. This can lead to situations where the programmer does not realize a function will change the value of the argument.
- Because references are typically implemented by C++ using pointers, and dereferencing a pointer is slower than accessing it directly, accessing values passed by reference is slower than accessing values passed by value.

" (LearnCPP, 2014)

PASS BY ADDRESS

"**Passing an argument by address** involves passing the address of the argument variable rather than the argument variable itself. Because the argument is an address, the function parameter must be a pointer. The function can then dereference the pointer to access or change the value being pointed to.

Here is an example of a function that takes a parameter passed by address:

```
1  void foo(int *pValue)
2  {
3      *pValue = 6;
4  }
```

```
5
6   int main()
7   {
8       int nValue = 5;
9
10      cout << "nValue = " << nValue << endl;
11      foo(&nValue);
12      cout << "nValue = " << nValue << endl;
13      return 0;
14  }
```

The above snippet prints:

```
nValue = 5
nValue = 6
```

It is always a good idea to ensure parameters passed by address are not null pointers before dereferencing them. Dereferencing a null pointer will typically cause the program to crash.

Advantages of passing by address:

- It allows us to have the function change the value of the argument, which is sometimes useful
- Because a copy of the argument is not made, it is fast, even when used with large structs or classes.
- We can return multiple values from a function.

Disadvantages of passing by address:

- Because literals and expressions do not have addresses, pointer arguments must be normal variables.
- All values must be checked to see whether they are null. Trying to dereference a null value will result in a crash. It is easy to forget to do this.
- Because dereferencing a pointer is slower than accessing a value directly, accessing arguments passed by address is slower than accessing arguments passed by value.

As you can see, pass by address and pass by reference have almost identical advantages and disadvantages. Because pass by reference is generally safer than pass by address, pass by reference should be preferred in most cases.

the only real difference between pointers and references is that references have a cleaner but more restrictive syntax. This makes references easier and safer to use, but also less flexible. This also means that pass by reference and pass by address are essentially identical in terms of efficiency.

Here's the one that may surprise you. When you pass an address to a function, that *address* is actually passed by value! Because the address is passed by value, if you change the value of that address within the function, you are actually changing a temporary copy. Consequently, the original pointer address will not be changed!

Even though the address itself is passed by value, you can still dereference that address to permanently change the value at that address! This is what differentiates pass by address (and reference) from pass by value.

The next logical question is, "What if we want to be able to change the address of an argument from within the function?". Turns out, this is surprisingly easy. You just use pass the pointer itself by reference (effectively passing the address by reference). You already learned that values passed by reference reflect any changes made in the function back to the original arguments. So in this case, we're telling the compiler that any changes made to the address of pTempPtr should be reflected back to pPtr! The syntax for doing a reference to a pointer is a little strange (and easy to get backwards): `int *&pPtr`. However, if you do get it backwards, the compiler will give you an error.

The following program illustrates using a reference to a pointer.

```
1  // pTempPtr is now a reference to a pointer to pPtr!
2  // This means if we change pTempPtr, we change pPtr!
3  void SetToSix(int *&pTempPtr)
4  {
5      using namespace std;
6
7      pTempPtr = &nSix;
8
9      // This will print 6
10     cout << *pTempPtr;
   }
```

” (LearnCPP, 2014)

INLINE FUNCTIONS

“The use of functions provides many benefits, including:

- The code inside the function can be reused.
- It is much easier to change or update the code in a function (which needs to be done once) than for every in-place instance. Duplicate code is a recipe for disaster.
- It makes your code easier to read and understand, as you do not have to know how a function is implemented to understand what it does.
- The function provides type checking.

However, one major downside of functions is that every time a function is called, there is a certain amount of performance overhead that occurs. This is because the CPU must store the address of the current instruction it is executing (so it knows where to return to later) along with other registers, all the function parameters must be created and assigned values, and the program has to branch to a new location. Code written in-place is significantly faster.

For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function’s code. This can result in a substantial performance penalty.

C++ offers a way to combine the advantages of functions with the speed of code written in-place: inline functions. The **inline** keyword is used to request that the compiler treat your function as an inline function. When the compiler compiles your code, all inline functions are expanded in-place — that is, the function call is replaced with a copy of the contents of the function itself, which removes the function call overhead! The downside is that because the inline function is expanded in-place for *every* function call, this can make your compiled code quite a bit larger, especially if the inline function is long and/or there are many calls to the inline function.

” (LearnCPP, 2014)

“Function overloading is a feature of C++ that allows us to create multiple functions with the same name, so long as they have different parameters.

Making a call to an overloaded function results in one of three possible outcomes:

1) A match is found. The call is resolved to a particular overloaded function.
2) No match is found. The arguments can not be matched to any overloaded function.
3) An ambiguous match is found. The arguments matched more than one overloaded function.

A **default parameter** is a function parameter that has a default value provided to it. If the user does not supply a value for this parameter, the default value will be used. If the user does supply a value for the default parameter, the user-supplied value is used. " (LearnCPP, 2014)

FUNCTION POINTERS

"consider the following function:

```
1  int foo();
```

If you guessed that foo is actually a constant pointer to a function, you are correct. When a function is called (via the () operator), the function pointer is dereferenced, and execution branches to the function.

Just like it is possible to declare a non-constant pointer to a variable, it's also possible to declare a non-constant pointer to a function. The syntax for doing so is one of the ugliest things you will ever see:

```
1  // pFoo is a pointer to a function that takes no arguments and returns an integer
2  int (*pFoo) ();
```

The parenthesis around *pFoo are necessary for precedence reasons, as `int *pFoo()` would be interpreted as a function named pFoo that takes no parameters and returns a pointer to an integer.

In the above snippet, pFoo is a pointer to a function that has no parameters and returns an integer. pFoo can "point" to any function that matches this signature.

Assigning a function to a function pointer

There are two primary things that you can do with a pointer to a function. First, you can assign a function to it:

```
1
2  int foo()
3  {
4  }
5
6  int goo()
7  {
8  }
9
10 int main()
11 {
12     int (*pFoo)() = foo; // pFoo points to function foo()
13     pFoo = goo; // pFoo now points to function goo()
14     return 0;
15 }
```

One common mistake is to do this:

```
1  pFoo = goo();
```

This would actually assign the return value from a call to function goo() to pFoo, which isn't what we want. We want pFoo to be assigned to function goo, not the return value from goo(). So no parenthesis are needed.

Note that the signature (parameters and return value) of the function pointer must match the signature of the function. Here is an example of this:

```
1  // function prototypes
2  int foo();
3  double goo();
4  int hoo(int nX);
5
6  // function pointer assignments
7  int (*pFcn1)() = foo; // okay
```

```
int (*pFcn2)() = goo; // wrong -- return types don't match!
double (*pFcn3)() = goo; // okay
pFcn1 = hoo; // wrong -- pFcn1 has no parameters, but hoo() does
int (*pFcn3)(int) = hoo; // okay

```

Calling a function using a function pointer

The second thing you can do with a function pointer is use it to actually call the function. There are two ways to do this. The first is via explicit dereference:

```
int foo(int nX)
{
}

int (*pFoo)(int) = foo; // assign pFoo to foo()

(*pFoo)(nValue); // call function foo(nValue) through pFoo.
```

The second way is via implicit dereference:

```
int foo(int nX)
{
}

int (*pFoo)(int) = foo; // assign pFoo to foo()

pFoo(nValue); // call function foo(nValue) through pFoo.
```

As you can see, the implicit dereference method looks just like a normal function call — which is what you'd expect, since normal function names are pointers to functions anyway! However, some older compilers do not support the implicit dereference method, but all modern compilers should.

Why use pointers to functions?

There are several cases where pointers to function can be of use. One of the most common is the case where you are writing a function to perform a task (such as sorting an array), but you want the user to be able to define how a particular part of that task will be performed (such as whether the array is sorted in ascending or descending order). Let's take a closer look at this problem as applied specifically to sorting, as an example that can be generalized to other similar problems.

All sorting algorithms work on a similar concept: the sorting algorithm walks through a bunch of numbers, does comparisons on pairs of numbers, and reorders the numbers based on the results of those comparisons. Consequently, by varying the comparison (which can be a function), we can change the way the function sorts without affecting the rest of the sorting code.

Making function pointers pretty with typedef

Let's face it — the syntax for pointers to functions is ugly. However, typedefs can be used to make pointers to functions look more like regular variables:

```
1  typedef bool (*pfcnValidate)(int, int);
```

This defines a typedef called "pfcnValidate" that is a pointer to a function that takes two ints and returns a bool.

Now instead of doing this:

```
1  bool Validate(int nX, int nY, bool (*pfcn)(int, int));
```

You can do this:

```
1  bool Validate(int nX, int nY, pfcnValidate pfcn)
```

" (LearnCPP, 2014)

"A **recursive function** in C++ is a function that calls itself." (LearnCPP, 2014)

HANDLING ERRORS

"When writing programs, it is almost inevitable that you will make mistakes. In this section, we will talk about the different kinds of errors that are made, and how they are commonly handled.

Errors fall into two categories: syntax and semantic errors.

Syntax errors

A syntax error occurs when you write a statement that is not valid according to the grammar of the C++ language. For example:

```
if 5 > 6 then write "not equal";
```

Although this statement is understandable by humans, it is not valid according to C++ syntax. The correct C++ statement would be:

```
1  if (5 > 6)
2      std::cout << "not equal";
```

Syntax errors are almost always caught by the compiler and are usually easy to fix. Consequently, we typically don't worry about them too much.

Semantic errors

A semantic error occurs when a statement is syntactically valid, but does not do what the programmer intended. For example:

```
1  for (int nCount=0; nCount<=3; nCount++)
2      std::cout << nCount << " ";
```

The programmer may have intended this statement to print `0 1 2`, but it actually prints `0 1 2 3`.

Semantic errors are not caught by the compiler, and can have any number of effects: they may not show up at all, cause the program to produce the wrong output, cause erratic behavior, corrupt data, or cause the program to crash.

It is largely the semantic errors that we are concerned with.

Semantic errors can occur in a number of ways. One of the most common semantic errors is a logic error. A **logic error** occurs when the programmer incorrectly codes the logic of a statement. The above for statement example is a logic error. Here is another example:

```
1  if (x >= 5)
```

```
2       std::cout << "x is greater than 5";
```

What happens when x is exactly 5? The conditional expression evaluates to true, and the program prints "x is greater than 5". Logic errors can be easy or hard to locate, depending on the nature of the problem.

Another common semantic error is the violated assumption. A **violated assumption** occurs when the programmer assumes that something will be either true or false, and it isn't. For example:

```
1  char strHello[] = "Hello, world!";
2  std::cout << "Enter an index: ";
3
4  int nIndex;
5  std::cin >> nIndex;
6
7  std::cout << "Letter #" << nIndex << " is " << strHello[nIndex] << std::endl;
```

See the potential problem here? The programmer has assumed that the user will enter a value between 0 and the length of "Hello, world!". If the user enters a negative number, or a large number, the array index nIndex will be out of bounds. In this case, since we are just reading a value, the program will probably print a garbage letter. But in other cases, the program might corrupt other variables, the stack, or crash.

Defensive programming is a form of program design that involves trying to identify areas where assumptions may be violated, and writing code that detects and handles any violation of those assumptions so that the program reacts in a predictable way when those violations do occur.

Detecting assumption errors

As it turns out, we can catch almost all assumptions that need to be checked in one of three locations:

- When a function has been called, the user may have passed the function parameters that are semantically meaningless.
- When a function returns, the return value may indicate that an error has occured.
- When program receives input (either from the user, or a file), the input may not be in the correct format.

Consequently, the following rules should be used when programming defensively:

- At the top of each function, check to make sure any parameters have appropriate values.
- After a function is called, check it's return value (if any), and any other error reporting mechanisms, to see if an error occured.
- Validate any user input to make sure it meets the expected formatting or range criteria.

Let's take a look at examples of each of these.

Problem: When a function is called, the user may have passed the function parameters that are semantically meaningless.

```
1  void PrintString(char *strString)
2  {
3      std::cout << strString;
4  }
```

Can you identify the assumption that may be violated? The answer is that the user might pass in a NULL pointer instead of a valid C-style string. If that happens, the program will crash. Here's the function again with code that checks to make sure the function parameter is non-NULL:

```
1  void PrintString(char *strString)
2  {
3      // Only print if strString is non-null
4      if (strString)
5          std::cout << strString;
6  }
```

Problem: When a function returns, the return value may indicate that an error has occured.

```
1  // Declare an array of 10 integers
2  int *panData = new int[10];
3  panData[5] = 3;
```

Can you identify the assumption that may be violated? The answer is that operator new (which actually calls a function to do the allocation) could fail if the user runs out of memory. If that happens, panData will be set to NULL, and when we use the subscript operator on panData, the program will crash. Here's a new version with error checking:

```
// Delcare an array of 10 integers
int *panData = new int[10];
// If something went wrong
if (!panData)
    exit(2); // exit the program with error code 2
panData[5] = 3;
```

Problem: When program receives input (either from the user, or a file), the input may not be in the correct format. Here's the sample program you saw previously:

```
char strHello[] = "Hello, world!";
std::cout << "Enter an index: ";

int nIndex;
std::cin >> nIndex;

std::cout << "Letter #" << nIndex << " is " << strHello[nIndex] << std::endl;
```

And here's the version that checks the user input to make sure it is valid:

```
 char strHello[] = "Hello, world!";

 int nIndex;
 do
 {
     std::cout << "Enter an index: ";
     std::cin >> nIndex;
 } while (nIndex < 0 || nIndex >= strlen(strHello));
```

```
8
9    std::cout << "Letter #" << nIndex << " is " << strHello[nIndex] << std::endl;
10
```

Handling assumption errors

Now that you know where assumption errors typically occur, let's finish up by talking about different ways to handle them when they do occur. There is no best way to handle an error — it really depends on the nature of the problem.

Here are some typical responses:

1) Quietly skip the code that depends on the assumption being valid:

```
1    void PrintString(char *strString)
2    {
3        // Only print if strString is non-null
4        if (strString)
5            std::cout << strString;
6    }
```

In the above example, if strString is null, we don't print anything. We have skipped the code that depends on strString being non-null.

2) If we are in a function, return an error code back to the caller and let the caller deal with the problem.

```
1    int g_anArray[10]; // a global array of 10 characters
2
3    int GetArrayValue(int nIndex)
4    {
5        // use if statement to detect violated assumption
6        if (nIndex < 0 || nIndex > 9)
7            return -1; // return error code to caller
8
```

```
    return g_anArray[nIndex];
}
```

In this case, the function returns -1 if the user passes in an invalid index.

3) If we want to terminate the program immediately, the **exit** function can be used to return an error code to the operating system:

```
int g_anArray[10]; // a global array of 10 characters

int GetArrayValue(int nIndex)
{
    // use if statement to detect violated assumption
    if (nIndex < 0 || nIndex > 9)
       exit(2); // terminate program and return error number 2 to OS

    return g_anArray[nIndex];
}
```

If the user enters an invalid index, this program will terminate immediately (with no error message) and pass error code 2 to the operating system.

4) If the user has entered invalid input, ask the user to enter the input again.

```
char strHello[] = "Hello, world!";

int nIndex;
do
{
    std::cout << "Enter an index: ";
    std::cin >> nIndex;
} while (nIndex < 0 || nIndex >= strlen(strHello));

std::cout << "Letter #" << nIndex << " is " << strHello[nIndex] << std::endl;
```

5) cerr is another mechanism that is meant specifically for printing error messages. **cerr** is an output stream (just like cout) that is also defined in iostream.h. Typically, cerr writes the error messages on the screen (just like cout), but it can also be individually redirected to a file.

```
void PrintString(char *strString)
{
    // Only print if strString is non-null
    if (strString)
        std::cout << strString;
    else
        std::cerr << "PrintString received a null parameter";
}
```

6) If working in some kind of graphical environment (eg. MFC or SDL), it is common to pop up a message box with an error code and then terminate the program.

Assert

Using a conditional statement to detect a violated assumption, along with printing an error message and terminating the program is such a common response to problems that C++ provides a shortcut method for doing this. This shortcut is called an assert.

An **assert statement** is a preprocessor macro that evaluates a conditional expression. If the conditional expression is true, the assert statement does nothing. If the conditional expression evaluates to false, an error message is displayed and the program is terminated. This error message contains the conditional expression that failed, along with the name of the code file and the line number of the assert. This makes it very easy to tell not only what the problem was, but where in the code the problem occurred. This can help with debugging efforts immensely.

The assert functionality lives in the cassert header, and is often used both to check that the parameters passed to a function are valid, and to check that the return value of a function call is valid.

```
int g_anArray[10]; // a global array of 10 characters
```

```
2
3   #include <cassert> // for assert()
4   int GetArrayValue(int nIndex)
5   {
6       // we're asserting that nIndex is between 0 and 9
7       assert(nIndex >= 0 && nIndex <= 9); // this is line 7 in Test.cpp
8
9       return g_anArray[nIndex];
10  }
```

If the user calls GetValue(-3), the program prints the following message:

```
Assertion failed: nIndex >= 0 && nIndex <=9, file C:\VCProjects\Test.cpp,
line 7
```

We strongly encourage you to use assert statements liberally throughout your code.

Exceptions

C++ provides one more method for detecting and handling errors known as exception handling. The basic idea is that when an error occurs, it is "thrown". If the current function does not "catch" the error, the caller of the function has a chance to catch the error. If the caller does not catch the error, the caller's caller has a chance to catch the error. The error progressively moves up the stack until it is either caught and handled, or until main() fails to handle the error. If nobody handles the error, the program typically terminates with an exception error.

" (LearnCPP, 2014)

THE STACK AND THE HEAP

"The memory a program uses is typically divided into four different areas:

- The code area, where the compiled program sits in memory.
- The globals area, where global variables are stored.
- The heap, where dynamically allocated variables are allocated from.
- The stack, where parameters and local variables are allocated from.

The heap

The heap (also known as the “free store”) is a large pool of memory used for dynamic allocation. In C++, when you use the new operator to allocate memory, this memory is assigned from the heap.

```
1  int *pValue = new int; // pValue is assigned 4 bytes from the heap
2  int *pArray = new int[10]; // pArray is assigned 40 bytes from the heap
```

Because the precise location of the memory allocated is not known in advance, the memory allocated has to be accessed indirectly — which is why new returns a pointer. You do not have to worry about the mechanics behind the process of how free memory is located and allocated to the user. However, it is worth knowing that sequential memory requests may not result in sequential memory addresses being allocated!

```
1  int *pValue1 = new int;
2  int *pValue2 = new int;
3  // pValue1 and pValue2 may not have sequential addresses
```

When a dynamically allocated variable is deleted, the memory is “returned” to the heap and can then be reassigned as future allocation requests are received.

The heap has advantages and disadvantages:
1) Allocated memory stays allocated until it is specifically deallocated (beware memory leaks).
2) Dynamically allocated memory must be accessed through a pointer.
3) Because the heap is a big pool of memory, large arrays, structures, or classes should be allocated here.

The stack

The call stack (usually referred to as “the stack”) has a much more interesting role to play. Before we talk about the call stack, which refers to a particular portion of memory, let’s talk about what a stack is.

Consider a stack of plates in a cafeteria. Because each plate is heavy and they are stacked, you can really only do one of three things:
1) Look at the surface of the top plate
2) Take the top plate off the stack
3) Put a new plate on top of the stack

In computer programming, a stack is a container that holds other variables (much like an array). However, whereas an array lets you access and modify elements in any order you wish, a stack is more limited. The operations that can be performed on a stack are identical to the ones above:

1) Look at the top item on the stack (usually done via a function called top())
2) Take the top item off of the stack (done via a function called pop())
3) Put a new item on top of the stack (done via a function called push())

A stack is a last-in, first-out (LIFO) structure. The last item pushed onto the stack will be the first item popped off. If you put a new plate on top of the stack, anybody who takes a plate from the stack will take the plate you just pushed on first. Last on, first off. As items are pushed onto a stack, the stack grows larger — as items are popped off, the stack grows smaller.

The plate analogy is a pretty good analogy as to how the call stack works, but we can actually make an even better analogy. Consider a bunch of mailboxes, all stacked on top of each other. Each mailbox can only hold one item, and all mailboxes start out empty. Furthermore, each mailbox is nailed to the mailbox below it, so the number of mailboxes can not be changed. If we can't change the number of mailboxes, how do we get a stack-like behavior?

First, we use a marker (like a post-it note) to keep track of where the bottom-most empty mailbox is. In the beginning, this will be the lowest mailbox. When we push an item onto our mailbox stack, we put it in the mailbox that is marked (which is the first empty mailbox), and move the marker up one mailbox. When we pop an item off the stack, we move the marker down one mailbox and remove the item from that mailbox. Anything below the marker is considered "on the stack". Anything at the marker or above the marker is not on the stack.

This is almost exactly analogous to how the call stack works. The call stack is a fixed-size chunk of sequential memory addresses. The mailboxes are memory addresses, and the "items" are pieces of data (typically either variables or addreses). The "marker" is a register (a small piece of memory) in the CPU known as the stack pointer. The stack pointer keeps track of where the top of the stack currently is.

The only difference between our hypothetical mailbox stack and the call stack is that when we pop an item off the call stack, we don't have to erase the memory (the equivalent of emptying the mailbox). We can just leave it to be overwritten by the next item pushed to that piece of memory. Because the stack pointer will be below that memory location, we know that memory location is not on the stack.

So what do we push onto our call stack? Parameters, local variables, and… function calls.

The stack in action

Because parameters and local variables essentially belong to a function, we really only need to consider what happens on the stack when we call a function. Here is the sequence of steps that takes place when a function is called:

1. The address of the instruction beyond the function call is pushed onto the stack. This is how the CPU remembers where to go after the function returns.
2. Room is made on the stack for the function's return type. This is just a placeholder for now.
3. The CPU jumps to the function's code.
4. The current top of the stack is held in a special pointer called the stack frame. Everything added to the stack after this point is considered "local" to the function.
5. All function arguments are placed on the stack.
6. The instructions inside of the function begin executing.
7. Local variables are pushed onto the stack as they are defined.

When the function terminates, the following steps happen:

1. The function's return value is copied into the placeholder that was put on the stack for this purpose.
2. Everything after the stack frame pointer is popped off. This destroys all local variables and arguments.
3. The return value is popped off the stack and is assigned as the value of the function. If the value of the function isn't assigned to anything, no assignment takes place, and the value is lost.
4. The address of the next instruction to execute is popped off the stack, and the CPU resumes execution at that instruction.

Typically, it is not important to know all the details about how the call stack works. However, understanding that functions are effectively pushed on the stack when they are called and popped off when they return gives you the fundamentals needed to understand recursion, as well as some other concepts that are useful when debugging.

Stack overflow

The stack has a limited size, and consequently can only hold a limited amount of information. If the program tries to put too much information on the stack, stack overflow will result. **Stack overflow**happens when all the memory in the stack has been allocated — in that case, further allocations begin overflowing into other sections of memory.

Stack overflow is generally the result of allocating too many variables on the stack, and/or making too many nested function calls (where function A calls function B calls function C calls function D etc…) Overflowing the stack generally causes the program to crash.

Here is an example program that causes a stack overflow. You can run it on your system and watch it crash:

```
1  int main()
2  {
3      int nStack[100000000];
4      return 0;
5  }
```

This program tries to allocate a huge array on the stack. Because the stack is not large enough to handle this array, the array allocation overflows into portions of memory the program is not allowed to use. Consequently, the program crashes.

The stack has advantages and disadvantages:

- Memory allocated on the stack stays in scope as long as it is on the stack. It is destroyed when it is popped off the stack.
- All memory allocated on the stack is known at compile time. Consequently, this memory can be accessed directly through a variable.
- Because the stack is relatively small, it is generally not a good idea to do anything that eats up lots of stack space. This includes allocating large arrays, structures, and classes, as well as heavy recursion.

" (LearnCPP, 2014)

OPERATORS AND EXPRESSIONS

"we had defined an expression as "A mathematical entity that evaluates to a value". However, the term *mathematical entity* is somewhat vague. More precisely, an **expression** is a combination of literals, variables, operators, and functions that evaluates to a value." (LearnCPP, 2014)

"A **literal** is simply a number, such as 5, or 3.14159. When we talk about the expression "3 + 4", both 3 and 4 are literals. Literals always evaluate to themselves.

You have already seen variables and functions. Variables evaluate to the values they hold. Functions evaluate to produce a value of the function's return type. Because functions that return void do not have return values, they are usually not part of expressions.

Literals, variables, and functions are all known as operands. **Operands** are the objects of an expression that are acted upon. Operands supply the data that the expression works with." (LearnCPP, 2014)"

The last piece of the expressions puzzle is operators. **Operators** tell how to combine the operands to produce a new result. For example, in the expression "3 + 4", the + is the plus operator. The + operator tells how to combine the operands 3 and 4 to produce a new value (7)." (LearnCPP, 2014)

"Operators come in two types:
Unary operators act on one operand. An example of a unary operator is the – operator. In the expression `-5`, the – operator is only being applied to one operand (5) to produce a new value (-5).

Binary operators act on two operands (known as left and right). An example of a binary operator is the + operator. In the expression `3 + 4`, the + operator is working with a left operand (3) and a right operand (4) to produce a new value (7)." (LearnCPP, 2014)

PREPROCESSORS

"The preprocessor is perhaps best thought of as a separate program that runs before the compiler when you compile your program. It's purpose is to process **directives**. Directives are specific instructions that start with a # symbol and end with a newline (NOT a semicolon). There are several different types of directives, which we will cover below. The preprocessor is not smart — it does not understand C++ syntax; rather, it manipulates text before the compiler gets to it." (LearnCPP, 2014)

"#Include tells the preprocessor to insert the contents of the included file into the current file at the point of the #include directive. This is useful when you have information that needs to be included in multiple places (as forward declarations often are)."

"Macro defines

Macro defines take the form:

```
#define identifier replacement
```

Whenever the preprocessor encounters this directive, any further occurrence of 'identifier' is replaced by 'replacement'. The identifier is traditionally typed in all capital letters, using underscores to represent spaces." (LearnCPP, 2014)

"Conditional compilation

The conditional compilation preprocessor directives allow you to specify under what conditions something will or won't compile. The only conditional compilation directives we are going to cover in this section are #ifdef, #ifndef, and #endif.

Header guards

Because header files can include other header files, it is possible to end up in the situation where a header file gets included multiple times. " (LearnCPP, 2014)

BASIC ADDRESSING AND VARIABLE DECLARATION

"The smallest unit of memory is a binary digit (bit), which can hold a value of 0 or 1. You can think of a bit as being like a traditional light switch — either the light is off (0), or it is on (1). There is no in-between. If you were to look at a sequential piece of memory, all you would see is …011010100101010… or some combination thereof. Memory is organized into individual sections called **addresses**. Perhaps surprisingly, in modern computers, each bit does not get it's own address. The smallest addressable unit of memory is a group of 8 bits known as a **byte**.

Because all data on a computer is just a sequence of bits, we use a **data type** to tell us how to interpret the contents of memory in some meaningful way. You have already seen one example of a data type: the integer. When we declare a variable as an integer, we are telling the computer "the piece of memory that this variable addresses is going to be interpreted as a whole number".

When you assign a value to a data type, the computer takes care of the details of encoding your value into the appropriate sequence of bits for that data type. When you ask for your value back, the program "reconstitutes" your number from the sequence of bits in memory.

You can also assign values to your variables upon declaration. When we assign values to a variable using the assignment operator (equals sign), it's called an **explicit assignment**:

```
1   int nValue = 5; // explicit assignment
```

You can also assign values to variables using an **implicit assignment**:

```
1   int nValue(5); // implicit assignment
```

Even though implicit assignments look a lot like function calls, the compiler keeps track of which names are variables and which are functions so that they can be resolved properly." (LearnCPP, 2014)

KEYWORDS AND NAMING IDENTIFIERS

"C++ reserves a set of 63 words for it's own use. These words are called **keywords**, and each of these keywords has a special meaning with in the C++ language. The 15 keywords that are starred (*) were added to the language after it's initial release, consequently, some older reference books or material may omit these.

Here is a list of all the C++ keywords:

asm auto bool * break case catch char class	const const_cast * continue default delete do double dynamic_cast *	else enum explicit * export * extern false * float for	friend goto if inline int long mutable * namespace *	new operator private protected public register reinterpret_cast * return	short signed sizeof static static_cast * struct switch template	this throw true * try typedef typeid * typename * union	unsigned using * virtual void volatile wchar_t * while

" (LearnCPP, 2014)

"**Identifiers, and naming them**

The name of a variable, function, class, or other entity in C++ is called an **identifier**. C++ gives you a lot of flexibility to name identifiers as you wish. However, there are a few rules that must be followed when naming identifiers:

- The identifier can not be a keyword. Keywords are reserved.
- The identifier can only be composed of letters, numbers, and the underscore character. That means the name can not contains symbols (except the underscore) nor whitespace.
- The identifier must begin with a letter or an underscore. It can not start with a number.
- C++ distinguishes between lower and upper case letters. `nvalue` is different than `nValue` is different than `NVALUE`." (LearnCPP, 2014)

PRECEDENCE AND ASSOCIATIVITY

"In order to properly evaluate an expression such as `4 + 2 * 3`, we must understand both what the operators do, and the correct order to apply them. The order in which operators are evaluated in a compound expression is called **operator precedence**. Using normal mathematical precedence rules (which states that multiplication is resolved before addition), we know that the above expression should evaluate as `4 + (2 * 3) = 10`.

In C++, all operators are assigned a level of precedence. Those with the highest precedence are evaluated first. You can see in the table below that multiplication and division

(precedence level 5) have a higher precedence than addition and subtraction (precedence level 6). The compiler uses these levels to determine how to evaluate expressions it encounters.

If two operators with the same precedence level are adjacent to each other in an expression, the **associativity rules** tell the compiler whether to evaluate the operators from left to right or from right to left. For example, in the expression `3 * 4 / 2`, the multiplication and division operators are both precedence level 5. Level 5 has an associativity of left to right, so the expression is resolved from left to right: `(3 * 4) / 2 = 6`.

Prec/Ass	Operator	Description	Example
1 None	:: ::	Global scope (unary) Class scope (binary)	::g_nGlobalVar = 5; Class::m_nMemberVar = 5;

2 L->R	()
	()
	()
	[]
	.
	->
	++
	––
	typeid
	const_cast
	dynamic_cast
	reinterpret_cast
	static_cast
		Parenthesis
		Function call
		Implicit assignment
		Array subscript
		Member access from object
		Member access from object ptr
		Post-increment
		Post-decrement
		Run-time type information
		Cast away const
		Run-time type-checked cast
		Cast one type to another
		Compile-time type-checked cast
			(x + y) * 2;
			Add(x, y);
			int nValue(5);
			aValue[3] = 2;
			cObject.m_nValue = 4;
			pObject->m_nValue = 4;
			nValue++;
			nValue––;
			typeid(cClass).name();
			const_cast<int*>(pnConstValue);
			dynamic_cast<Shape*>(pShape);
			reinterpret_cast<Class2>(cClass1);
			fValue = static_cast<float>(nValue);

3 R->L	+
	-
	++
	––
	!
	~
	(type)
	sizeof
		Unary plus
		Unary minus
		Pre-increment
		Pre-decrement
		Logical NOT
		Bitwise NOT
		C-style cast
		Size in bytes
			nValue = +5;M
			nValue = -1;
			++nValue;
			––nValue;
			if (!bValue)
			nFlags = ~nFlags;
			float fValue = (float)nValue;
			sizeof(int);

	&
	*
	new
	new[]
	delete
	delete[]
		Address of
		Dereference
		Dynamic memory allocation
		Dynamic array allocation
		Dynamic memory deletion
		Dynamic array deletion
			address = &nValue;
			nValue = *pnValue;
			int *pnValue = new int;
			int *panValue = new int[5];
			delete pnValue;
			delete[] panValue;

4 L->R	->* .*	Member pointer selector Member object selector	pObject->*pnValue = 24; cObject->.*pnValue = 24;

5 L->R	*
	/
	%
		Multiplication
		Division
		Modulus
			int nValue = 2 * 3;
			float fValue = 5.0 / 2.0;
			int nRemainder = 10 % 3;

6 L->R	+ -	Addition Subtraction	int nValue = 2 + 3; int nValue = 2 – 3;
7 L->R	<< >>	Bitwise shift left Bitwise shift right	int nFlags = 17 << 2; int nFlags = 17 >> 2;

8 L->R	<
	<=
	>
	>=
		Comparison less than
		Comparison less than or equals
		Comparison greater than
		Comparison greater than or equals
			if (x < y)
			if (x <= y)
			if (x > y)
			if (x >= y)

9 L->R	== !=	Equality Inequality	if (x == y) if (x != y)
10 L->R	&	Bitwise AND	nFlags = nFlags & 17;
11 L->R	^	Bitwise XOR	nFlags = nFlags ^ 17;
12 L->R	|	Bitwise OR	nFlags = nFlags | 17;

13 L->R	&&	Logical AND	if (bValue1 && bValue2)
14 L->R	||	Logical OR	if (bValue1 || bValue2)
15 L->R	?:	Arithmetic if	return (x < y) ? true : false;

16 R->L	=
	*=
	/=
	%=
	+=
	-=
	<<=
	>>=
	&=
	|=
	^=
		Assignment
		Multiplication assignment
		Division assignment
		Modulus assignment
		Addition assignment
		Subtraction assignment
		Bitwise shift left assignment
		Bitwise shift right assignment
		Logical AND assingment
		Logical OR assignment
		Logical XOR assignment
			nValue = 5;
			nValue *= 5;
			fValue /= 5.0;
			nValue %= 5;
			nValue += 5;
			nValue -= 5;
			nFlags <<= 2;
			nFlags >>= 2;
			nFlags &= 17;
			nFlags |= 17;
			nFlags ^= 17;

17 L->R	,	Comma operator	iii++, jjj++, kkk++;

" (LearnCPP, 2014)

POINTERS

"A **pointer** is a variable that holds the address of another variable. To declare a pointer, we use an asterisk between the data type and the variable name

Note that an asterisk placed between the data type and the variable name means the variable is being declared as a pointer. In this context, the asterisk is not a multiplication. It does not matter if the asterisk is placed next to the data type, the variable name, or in the middle — different programmers prefer different styles, and one is not inherently better than the other.

Since pointers only hold addresses, when we assign a value to a pointer, the value has to be an address. To get the address of a variable, we can use the **address-of operator (&)**:

```
1   int nValue = 5;
    int *pnPtr = &nValue; // assign address of nValue to pnPtr
```

The other operator that is commonly used with pointers is the **dereference operator (*)**. A dereferenced pointer evaluates to the *contents* of the address it is pointing to.

Sometimes it is useful to make our pointers point to nothing. This is called a **null pointer**. We assign a pointer a null value by setting it to address 0.

The size of a pointer is dependent upon the architecture of the computer — a 32-bit computer uses 32-bit memory addresses — consequently, a pointer on a 32-bit machine is 32 bits (4 bytes). On a 64-bit machine, a pointer would be 64 bits (8 bytes). Note that this is true regardless of what is being pointed to.

The C language allows you to perform integer addition or subtraction operations on pointers. If pnPtr points to an integer, `pnPtr + 1` is the address of the next integer in memory after pnPtr. `pnPtr - 1` is the address of the previous integer before pnPtr.

Note that `pnPtr+1` does not return the ADDRESS after pnPtr, but the NEXT OBJECT OF THE TYPE that pnPtr points to. If pnPtr points to an integer (assuming 4 bytes), `pnPtr+3` means 3 integers after pnPtr, which is 12 addresses after pnPtr. If pnPtr points to a char, which is always 1 byte, `pnPtr+3` means 3 chars after pnPtr, which is 3 addresses after pnPtr.

When calculating the result of a pointer arithmetic expression, the compiler always multiplies the integer operand by the size of the object being pointed to. This is called **scaling**.

Dynamic memory allocation allows us to allocate memory of whatever size we want when we need it.

When we are done with a dynamically allocated variable, we need to explicitly tell C++ to free the memory for reuse. This is done via the scalar (non-array) form of the **delete** operator.

The **void pointer**, also known as the generic pointer, is a special type of pointer that can be pointed at objects of any data type! A void pointer is declared like a normal pointer, using the void keyword as the pointer's type:

```
1   void *pVoid; // pVoid is a void pointer
```

A void pointer can point to objects of any data type:

```
1   int nValue;
2   float fValue;
```

```
3
4    struct Something
5    {
6        int nValue;
7        float fValue;
8    };
9
10   Something sValue;
11
12   void *pVoid;
13   pVoid = &nValue; // valid
14   pVoid = &fValue; // valid
15   pVoid = &sValue; // valid
```

However, because the void pointer does not know what type of object it is pointing to, it can not be dereferenced! Rather, the void pointer must first be explicitly cast to another pointer type before it is dereferenced.

it is not possible to do pointer arithmetic on a void pointer. Note that since void pointers can't be dereferenced, there is no such thing as a void reference. " (LearnCPP, 2014)

CLASSES

"In C++, classes are very much like structs, except that classes provide much more power and flexibility. In fact, the following struct and class are effectively identical.

Just like a struct definition, a class definition does not declare any memory. It only defines what the class looks like. In order to use a class, a variable of that class type must be declared.

In C++, when we declare a variable of a class, we call it **instantiating** the class. The variable itself is called an **instance** of the class. A variable of a class type is also called an **object**.

In addition to holding data, classes can also contain functions!

Public members are members of a struct or class that can be accessed by any function in the program.

Private members are members of a class that can only be accessed by other functions within the class.

One of the primary differences between classes and structs is that classes can explicitly use **access specifiers** to restrict who can access members of a class. C++ provides 3 different access specifier keywords: public, private, and protected.

An **access function** is a short public function whose job is to return the value of a private member variable. For example, in the above mentioned String class, you might see something like this:

```
class String
{
private:
    char *m_chString; // a dynamically allocated string
    int m_nLength; // the length of m_chString

public:
    int GetLength() { return m_nLength; }
};
```

GetLength() is an access function that simply returns the value of m_nLength.

Access functions typically come in two flavors: getters and setters. **Getters** are functions that simply return the value of a private member variable. Setters are functions that simply set the value of a private member variable.

Here's an example class that has some getters and setters:

```
class Date
{
private:
    int m_nMonth;
    int m_nDay;
    int m_nYear;

public:
    // Getters
    int GetMonth() { return m_nMonth; }
```

```
10      int GetDay() { return m_nDay; }
11      int GetYear() { return m_nYear; }
12
13      // Setters
14      void SetMonth(int nMonth) { m_nMonth = nMonth; }
15      void SetDay(int nDay) { m_nDay = nDay; }
16      void SetYear(int nYear) { m_nYear = nYear; }
17  };
18
```

Why bother to make a member variable private if we're going to provide public access functions to it? The answer is: "encapsulation".

The downside of virtual functions

Since most of the time you'll want your functions to be virtual, why not just make all functions virtual? The answer is because it's inefficient — resolving a virtual function call takes longer than a resolving a regular one. Furthermore, the compiler also has to allocate an extra pointer for each class object that has one or more virtual functions. We'll talk about this more in the next couple of lessons.

When a C++ program is executed, it executes sequentially, beginning at the top of main(). When a function call is encountered, the point of execution jumps to the beginning of the function being called. How does the CPU know to do this?

When a program is compiled, the compiler converts each statement in your C++ program into one or more lines of machine language. Each line of machine language is given it's own unique sequential address. This is no different for functions — when a function is encountered, it is converted into machine language and given the next available address. Thus, each function ends up with a unique machine language address.

Binding refers to the process that is used to convert identifiers (such as variable and function names) into machine language addresses. Although binding is used for both variables and functions, in this lesson we're going to focus on function binding.

Early binding

Direct function calls can be resolved using a process known as early binding. **Early binding** (also called static binding) means the compiler is able to directly associate the identifier name (such as a function or variable name) with a machine address. Remember

that all functions have a unique machine address. So when the compiler encounters a function call, it replaces the function call with a machine language instruction that tells the CPU to jump to the address of the function.

Late Binding

In some programs, it is not possible to know which function will be called until runtime (when the program is run). This is known as **late binding** (or dynamic binding). In C++, one way to get late binding is to use function pointers. To review function pointers briefly, a function pointer is a type of pointer that points to a function instead of a variable. The function that a function pointer points to can be called by using the function call operator (()) on the pointer.

Late binding is slightly less efficient since it involves an extra level of indirection. With early binding, the compiler can tell the CPU to jump directly to the function's address. With late binding, the program has to read the address held in the pointer and then jump to that address. This involves one extra step, making it slightly slower. However, the advantage of late binding is that it is more flexible than early binding, because decisions about what function to call do not need to be made until run time.

INHERITANCE AND ACCESS SPECIFIERS

"**Public inheritance**

This is fairly straightforward. The things worth noting are:

1. Derived classes can not directly access private members of the base class.
2. The protected access specifier allows derived classes to directly access members of the base class while not exposing those members to the public.
3. The derived class uses access specifiers from the base class.
4. The outside uses access specifiers from the derived class.

To summarize in table form:

Public inheritance			
Base access specifier	**Derived access specifier**	**Derived class access?**	**Public access?**
Public	Public	Yes	Yes
Private	Private	No	No

Protected	Protected	Yes	No

Private inheritance

With private inheritance, all members from the base class are inherited as private. This means private members stay private, and protected and public members become private.

Note that this does not affect that way that the derived class accesses members inherited from its parent! It only affects the code trying to access those members through the derived class.

To summarize in table form:

Private inheritance			
Base access specifier	**Derived access specifier**	**Derived class access?**	**Public access?**
Public	Private	Yes	No
Private	Private	No	No
Protected	Private	Yes	No

Protected inheritance

Protected inheritance is the last method of inheritance. It is almost never used, except in very particular cases. With protected inheritance, the public and protected members become protected, and private members stay private.

To summarize in table form:

Protected inheritance			
Base access specifier	**Derived access specifier**	**Derived class access?**	**Public access?**
Public	Protected	Yes	No

Private	Private	No	No
Protected	Protected	Yes	No

Protected inheritance is similar to private inheritance. However, classes derived from the derived class still have access to the public and protected members directly. The public (stuff outside the class) does not.

Summary

The way that the access specifiers, inheritance types, and derived classes interact causes a lot of confusion. To try and clarify things as much as possible:

First, the base class sets it's access specifiers. The base class can always access it's own members. The access specifiers only affect whether outsiders and derived classes can access those members.

Second, derived classes have access to base class members based on the access specifiers of the immediate parent. The way a derived class accesses inherited members is not affected by the inheritance method used!

Finally, derived classes can change the access type of inherited members based on the inheritance method used. This does not affect the derived classes own members, which have their own access specifiers. It only affects whether outsiders and classes derived from the derived class can access those inherited members.

" (LearnCPP, 2014)

THIS POINTER

"The **this pointer** is a hidden pointer inside every class member function that points to the class object the member function is working with.

Most of the time, you never need to explicitly reference the "this" pointer. However, there are a few occasions where it can be useful:

1) If you have a constructor (or member function) that has a parameter of the same name as a member variable, you can disambiguate them by using "this":

```
1  class Something
2  {
```

```
private:
    int nData;

public:
    Something(int nData)
    {
        this->nData = nData;
    }
};

```

Note that our constructor is taking a parameter of the same name as a member variable. In this case, "nData" refers to the parameter, and "this->nData" refers to the member variable. Although this is acceptable coding practice, we find using the "m_" prefix on all member variable names provides a better solution by preventing duplicate names altogether!

2) Occasionally it can be useful to have a function return the object it was working with. Returning *this will return a reference to the object that was implicitly passed to the function by C++.

One use for this feature is that it allows a series of functions to be "chained" together, so that the output of one function becomes the input of another function! The following is somewhat more advanced and can be considered optional material at this point.

Consider the following class:

```
class Calc
{
private:
    int m_nValue;

public:
    Calc() { m_nValue = 0; }

    void Add(int nValue) { m_nValue += nValue; }
    void Sub(int nValue) { m_nValue -= nValue; }
```

```
10      void Mult(int nValue) { m_nValue *= nValue; }
11
12      int GetValue() { return m_nValue; }
13  };
14
```

If you wanted to add 5, subtract 3, and multiply by 4, you'd have to do this:

```
1  Calc cCalc;
2  cCalc.Add(5);
3  cCalc.Sub(3);
4  cCalc.Mult(4);
```

However, if we make each function return *this, we can chain the calls together. Here is the new version of Calc with "chainable" functions:

```
1
2   class Calc
3   {
4   private:
5       int m_nValue;
6
7   public:
8       Calc() { m_nValue = 0; }
9
10      Calc& Add(int nValue) { m_nValue += nValue; return *this; }
11      Calc& Sub(int nValue) { m_nValue -= nValue; return *this; }
12      Calc& Mult(int nValue) { m_nValue *= nValue; return *this; }
13
14      int GetValue() { return m_nValue; }
15  };
16
```

Note that Add(), Sub() and Mult() are now returning *this, which is a reference to the class itself. Consequently, this allows us to do the following:

```
1  Calc cCalc;
2  cCalc.Add(5).Sub(3).Mult(4);
```

We have effectively condensed three lines into one expression! Let's take a closer look at how this works.

First, cCalc.Add(5) is called, which adds 5 to our m_nValue. Add() then returns *this, which is a reference to cCalc. Our expression is now cCalc.Sub(3).Mult(4). cCalc.Sub(3) subtracts 3 from m_nValue and returns cCalc. Our expression is now cCalc.Mult(4). cCalc.Mult(4) multiplies m_nValue by 4 and returns cCalc, which is then ignored. However, since each function modified cCalc as it was executed, cCalc now contains the value ((0 + 5) – 3) * 4), which is 8.

Although this is a pretty contrived example, chaining functions in such a manner is common with String classes. For example, it is possible to overload the + operator to do a string append. If the + operator returns *this, then it becomes possible to write expressions like:

```
1  cMyString = "Hello " + strMyName + " welcome to " + strProgramName + ".";
```

C++ supports: Static member variables and Static member functions

A **friend function** is a function that can access the private members of a class as though it were a member of that class. In all other regards, the friend function is just like a normal function. A friend function may or may not be a member of another class. To declare a friend function, simply use the *friend* keyword in front of the prototype of the function you wish to be a friend of the class. It does not matter whether you declare the friend function in the private or public section of the class. It is also possible to make an entire class a friend of another class.

An **anonymous variable** is a variable that is given no name. Anonymous variables in C++ have "expression scope", meaning they are destroyed at the end of the expression in which they are created. Consequently, they must be used immediately!

Here is the Add() function rewritten using an anonymous variable:

```
1  int Add(int nX, int nY)
2  {
```

```
3       return nX + nY;
4   }
```

” (LearnCPP, 2014)

CONSTRUCTOR

“A **constructor** is a special kind of class member function that is executed when an object of that class is instantiated. Constructors are typically used to initialize member variables of the class to appropriate default values, or to allow the user to easily initialize those member variables to whatever values are desired.

Unlike normal functions, constructors have specific rules for how they must be named:
1) Constructors should always have the same name as the class (with the same capitalization)
2) Constructors have no return type (not even void)

A constructor that takes no parameters (or has all optional parameters) is called a **default constructor**.

While the default constructor is great for ensuring our classes are initialized with reasonable default values, often times we want instances of our class to have specific values. Fortunately, constructors can also be declared with parameters.

A default constructor that will be called in the default case, and a second constructor that takes two parameters. These two constructors can coexist peacefully in the same class due to function overloading. In fact, you can define as many constructors as you want, so long as each has a unique signature (number and type of parameters).

What happens if we do not declare a default constructor and then instantiate our class? The answer is that C++ will allocate space for our class instance, but will not initialize the members of the class (similar to what happens when you declare an int, double, or other basic data type).

Constructor chaining and initialization issues

When you instantiate a new object, the object’s constructor is called implicitly by the C++ compiler. Let’s take a look at two related situations that often cause problems for new programmers:

First, sometimes a class has a constructor which needs to do the same work as another constructor, plus something extra. The process of having one constructor call another constructor is called **constructor chaining**. Although some languages such as C# support constructor chaining, C++ does not. If you try to chain constructors, it will usually compile,

but it will not work right, and you will likely spend a long time trying to figure out why, even with a debugger. However, constructors *are* allowed to call non-constructor functions in the class. Just be careful that any members the non-constructor function uses have already been initialized.

Although you may be tempted to copy code from the first constructor into the second constructor, having duplicate code makes your class harder to understand and more burdensome to maintain. The best solution to this issue is to create a non-constructor function that does the common initialization, and have both constructors call that function.

Second, you may find yourself in the situation where you want to write a member function to re-initialize a class back to default values. Because you probably already have a constructor that does this, you may be tempted to try to call the constructor from your member function. As mentioned, chaining constructor calls are illegal in C++. You could copy the code from the constructor in your function, which would work, but lead to duplicate code. The best solution in this case is to move the code from the constructor to your new function, and have the constructor call your function to do the work of initializing the data." (LearnCPP, 2014)

DESTRUCTORS

"A **destructor** is another special kind of class member function that is executed when an object of that class is destroyed. They are the counterpart to constructors. When a variable goes out of scope, or a dynamically allocated variable is explicitly deleted using the delete keyword, the class destructor is called (if it exists) to help clean up the class before it is removed from memory. For simple classes, a destructor is not needed because C++ will automatically clean up the memory for you. However, if you have dynamically allocated memory, or if you need to do some kind of maintenance before the class is destroyed (eg. closing a file), the destructor is the perfect place to do so.

Like constructors, destructors have specific naming rules:
1) The destructor must have the same name as the class, preceded by a tilde (~).
2) The destructor can not take arguments.
3) The destructor has no return type.

Note that rule 2 implies that only one destructor may exist per class, as there is no way to overload destructors since they can not be differentiated from each other based on arguments.

" (LearnCPP, 2014)

CONST CLASS OBJECTS AND MEMBER FUNCTIONS

"Just like the built-in data types (int, double, char, etc…), class objects can be made const by using the const keyword. All const variables must be initialized at time of creation. In the

case of built-in data types, initilization is done through explicit or implicit assignment. In the case of classes, this initialization is done via constructors.

If a class is not initialized using a parameterized constructor, a public default constructor *must* be provided — if no public default constructor is provided in this case, a compiler error will occur.

Once a const class object has been initialized via constructor, any attempt to modify the member variables of the object is disallowed, as it would violate the constness of the object. This includes both changing member variables directly (if they are public), or calling member functions that sets the value of member variables.

```
1  class Something
2  {
3  public:
4      int m_nValue;
5
6      Something() { m_nValue = 0; }
7
8      void ResetValue() { m_nValue = 0; }
9      void SetValue(int nValue) { m_nValue = nValue; }
10
11     int GetValue() { return m_nValue; }
12 };
13
14 int main()
15 {
16     const Something cSomething; // calls default constructor
17
18     cSomething.m_nValue = 5; // violates const
19     cSomething.ResetValue(); // violates const
20     cSomething.SetValue(5); // violates const
21
22     return 0;
23 }
```

```
22
23
```

All three of the above lines involving cSomething are illegal because they violate the constness of cSomething by attempting to change a member variable or calling a member function that attempts to change a member variable.

Now, consider the following call:

```
1  std::cout << cSomething.GetValue();
```

Surprisingly, this will cause a compile error! This is because const class objects can only call const member functions, GetValue() has not been marked as a const member function.
A **const member function**is a member function that guarantees it will not change any class variables or call any non-const member functions.

To make GetValue() a const member function, we simply append the const keyword to the function prototype:

```
1
    class Something
2
    {
3
    public:
4       int m_nValue;
5
6       Something() { m_nValue = 0; }
7
8       void ResetValue() { m_nValue = 0; }
9       void SetValue(int nValue) { m_nValue = nValue; }
10
11      int GetValue() const { return m_nValue; }
    };
12
```

Now GetValue() has been made a const member function, which means we can call it on any const objects.

Const member functions declared outside the class definition must specify the const keyword on both the function prototype in the class definition and on the function prototype in the code file.

Note that constructors should not be marked as const. This is because const objects should initialize their member variables, and a const constructor would not be able to do so.

Finally, although it is not done very often, it is possible to overload a function in such a way to have a const and non-const version of the same function:

```
class Something
{
public:
    int m_nValue;

    const int& GetValue() const { return m_nValue; }
    int& GetValue() { return m_nValue; }
};
```

The const version of the function will be called on any const objects, and the non-const version will be called on any non-const objects:

```
Something cSomething;
cSomething.GetValue(); // calls non-const GetValue();

const Something cSomething2;
cSomething2.GetValue(); // calls const GetValue();
```

Overloading a function with a const and non-const version is typically done when the return value needs to differ in constness. In the example above, the const version of GetValue() returns a const reference, whereas the non-const version returns a non-const reference." (LearnCPP, 2014)

OPERATOR OVERLOADING

"**Operator overloading** allows the programmer to define how operators (such as +, -, ==, =, and !) should interact with various data types. Because operators in C++ are

implemented as functions, operator overloading works very analogously to function overloading.

Almost any operator in C++ can be overloaded. The exceptions are: arithmetic if (?:), sizeof, scope (::), member selector (.), and member pointer selector (.*). You can overload the + operator to concatenate your user-defined string class, or add two Fraction class objects together. You can overload the << operator to make it easy to print your class to the screen (or a file). You can overload the equality operator (==) to compare two objects. This makes operator overloading one of the most useful features in C++ -- simply because it allows you to work with your classes in a more intuitive way.

Before we go into more details, there are a few things to keep in mind going forward.

First, at least one of the operands in any overloaded operator must be a user-defined type. This means you can not overload the plus operator to work with one integer and one double. However, you could overload the plus operator to work with an integer and a Mystring.

Second, you can only overload the operators that exist. You can not create new operators. For example, you could not create an operator ** to do exponents.

Third, all operators keep their current precedence and associativity, regardless of what they're used for. For example, the bitwise XOR operator (^) could be overloaded to do exponents, except it has the wrong precedence and associativity and there is no way to change this.

Within those confines, you will still find plenty of useful functionality to overload for your custom classes!

Operators as functions

When you see the expression `nX + nY`, you can translate this in your head to `operator+(nX, nY)` (where operator+ is the name of the function). Similarly `dX + dY` becomes `operator+(dX, dY)`. Even though both expressions call a function named operator+(), function overloading is used to resolve the function calls to different versions of the function based on parameter type(s).

More generally, when evaluating an expression with operators, C++ looks at the operands around the operator to see what type they are. If *all* operands are built-in types, C++ calls a built-in routine. If *any*of the operands are user data types (eg. one of your classes), it looks to see whether the class has an overloaded operator function that it can call. If the compiler finds an overloaded operator whose parameters match the types of the operands, it calls that function. Otherwise, it produces a compiler error.

When C++ evaluates the expression `x + y`, x becomes the first parameter, and y becomes the second parameter. When x and y have the same type, it does not matter if you add x + y or y + x — either way, the same version of operator+ gets called. However, when the operands have different types, x + y is not the same as y + x.

For example, `Cents(4) + 6` would call operator+(Cents, int), and `6 + Cents(4)` would call operator+(int, Cents). Consequently, whenever we overload binary operators for operands of different types, we actually need to write two functions — one for each case. Here is an example of that." (LearnCPP, 2014)

OVERLOADING THE TYPECAST OPERATORS

"Overloading the typecast operators allows us to convert our class into another data type.

we'll overload the int cast, which will allow us to cast our Cents class into an int. The following example shows how this is done:

```
1
2   class Cents
3   {
4   private:
5       int m_nCents;
6   public:
7       Cents(int nCents=0)
8       {
9           m_nCents = nCents;
10      }
11
12      // Overloaded int cast
13      operator int() { return m_nCents; }
14
15      int GetCents() { return m_nCents; }
16      void SetCents(int nCents) { m_nCents = nCents; }
    };
```

There are two things to note:
1) To overload the function that casts our class to an int, we write a new function in our class called operator int(). Note that there is a space between the word operator and the

type we are casting to.
2) Casting operators do not have a return type. C++ assumes you will be returning the correct type.

Now in our example, we call PrintInt() like this:

```
1  int main()
2  {
3      Cents cCents(7);
4      PrintInt(cCents); // print 7
5
6      return 0;
7  }
```

The compiler will first note that PrintInt takes an integer parameter. Then it will note that cCents is not an int. Finally, it will look to see if we've provided a way to convert a Cents into an int. Since we have, it will call our operator int() function, which returns an int, and the returned int will be passed to PrintInt().

We can now also explicitly cast our Cents variable to an int:

```
1  Cents cCents(7);
2  int nCents = static_cast<int>(cCents);
```

You can overload cast operators for any data type you wish, including your own user-defined data types!

Here's a new class called Dollars that provides an overloaded Cents cast operator:

```
1   class Dollars
2   {
3   private:
        int m_nDollars;
4   public:
5       Dollars(int nDollars=0)
6       {
```

```
7           m_nDollars = nDollars;
8       }
9
10      // Allow us to convert Dollars into Cents
11      operator Cents() { return Cents(m_nDollars * 100); }
12  };
13
```

This allows us to convert a Dollars object directly into a Cents object! This allows you to do something like this:

```
1
2   void PrintCents(Cents cCents)
3   {
4       cout << cCents.GetCents();
5   }
6
7   int main()
8   {
9       Dollars cDollars(9);
10      PrintCents(cDollars); // cDollars will be cast to a Cents
11
12      return 0;
13  }
```

Consequently, this program will print the value:

```
900
```

which makes sense, since 9 dollars is 900 cents!

" (LearnCPP, 2014)

OVERLOADING OPERATORS USING MEMBER FUNCTIONS

In the lesson on **overloading the arithmetic operators**, you learned that when the operator does not modify it's operands, it's best to implement the overloaded operator as a friend function of the class. For operators that do modify their operands, we typically overload the operator using a member function of the class.

Overloading operators using a member function is very similar to overloading operators using a friend function. When overloading an operator using a member function:

- The leftmost operand of the overloaded operator must be an object of the class type.
- The leftmost operand becomes the implicit *this parameter. All other operands become function parameters.

Most operators can actually be overloaded either way, however there are a few exception cases:

- If the leftmost operand is not a member of the class type, such as when overloading operator+(int, YourClass), or operator<<(ostream&, YourClass), the operator must be overloaded as a friend.
- The assignment (=), subscript ([]), call (()), and member selection (->) operators must be overloaded as member functions.

Overloading the unary negative (-) operator

The negative operator is a unary operator that can be implemented using either method. Before we show you how to overload the operator using a member function, here's a reminder of how we overloaded it using a friend function:

```
1   class Cents
2   {
3   private:
4       int m_nCents;
5
6   public:
7       Cents(int nCents) { m_nCents = nCents; }
8
9       // Overload -cCents
10      friend Cents operator-(const Cents &cCents);
11  };
```

```
12
13  // note: this function is not a member function!
14  Cents operator-(const Cents &cCents)
15  {
16      return Cents(-cCents.m_nCents);
17  }
```

Now let's overload the same operator using a member function instead:

```
1   class Cents
2   {
3   private:
4       int m_nCents;
5
6   public:
7       Cents(int nCents) { m_nCents = nCents; }
8
9       // Overload -cCents
10      Cents operator-();
11  };
12
13  // note: this function is a member function!
14  Cents Cents::operator-()
15  {
16      return Cents(-m_nCents);
17  }
```

You'll note that this method is pretty similar. However, the member function version of operator- doesn't take any parameters! Where did the parameter go? In the lesson on **the hidden this pointer**, you learned that a member function has an implicit *this pointer which always points to the class object the member function is working on. The parameter we had to list explicitly in the friend function version (which doesn't have a *this pointer) becomes the implicit *this parameter in the member function version.

Remember that when C++ sees the function prototype `Cents Cents::operator-();`, the compiler internally converts this to `Cents operator-(const Cents *this)`, which you will note is almost identical to our friend version `Cents operator-(const Cents &cCents)`!

Overloading the binary addition (+) operator

Let's take a look at an example of a binary operator overloaded both ways. First, overloading operator+ using the friend function:

```
1  class Cents
2  {
3  private:
4      int m_nCents;
5
6  public:
7      Cents(int nCents) { m_nCents = nCents; }
8
9      // Overload cCents + int
10     friend Cents operator+(Cents &cCents, int nCents);
11
12     int GetCents() { return m_nCents; }
13 };
14
15 // note: this function is not a member function!
16 Cents operator+(Cents &cCents, int nCents)
17 {
18     return Cents(cCents.m_nCents + nCents);
19 }
```

Now, the same operator overloaded using the member function method:

```
1  class Cents
2  {
```

```
3   private:
4       int m_nCents;
5
6   public:
7       Cents(int nCents) { m_nCents = nCents; }
8
9       // Overload cCents + int
10      Cents operator+(int nCents);
11
12      int GetCents() { return m_nCents; }
13  };
14
15  // note: this function is a member function!
16  Cents Cents::operator+(int nCents)
17  {
18      return Cents(m_nCents + nCents);
19  }
```

Our two-parameter friend function becomes a one-parameter member function, because the leftmost parameter (cCents) becomes the implicit *this parameter in the member function version.

Most programmers find the friend function version easier to read than the member function version, because the parameters are listed explicitly. Furthermore, the friend function version can be used to overload some things the member function version can not. For example, `friend operator+(int, cCents)` can not be converted into a member function because the leftmost parameter is not a class object.

However, when dealing with operands that modify the class itself (eg. operators =, +=, -=, ++, --, etc…) the member function method is typically used because C++ programmers are used to writing member functions (such as access functions) to modify private member variables. Writing friend functions that modify private member variables of a class is generally not considered good coding style, as it violates encapsulation.

Furthermore, as mentioned, some specific operators must be implemented as member functions.

” (LearnCPP, 2014)

THE COPY CONSTRUCTOR AND OVERLOADING THE ASSIGNMENT OPERATOR

“copy constructor is a special constructor that initializes a *new object* from an existing object.

The purpose of the copy constructor and the assignment operator are almost equivalent — both copy one object to another. However, the assignment operator copies to existing objects, and the copy constructor copies to newly created objects.

The difference between the copy constructor and the assignment operator causes a lot of confusion for new programmers, but it’s really not all that difficult. Summarizing:

- If a new object has to be created before the copying can occur, the copy constructor is used.
- If a new object does not have to be created before the copying can occur, the assignment operator is used.

There are three general cases where the copy constructor is called instead of the assignment operator:

1. When instantiating one object and initializing it with values from another object (as in the example above).
2. When passing an object by value.
3. When an object is returned from a function by value.

In each of these cases, a new variable needs to be created before the values can be copied — hence the use of the copy constructor.

Because the copy constructor and assignment operator essentially do the same job (they are just called in different cases), the code needed to implement them is almost identical.

the parameter MUST be passed by reference, and not by value. A copy constructor is called when a parameter is passed by value. If we pass our cSource parameter by value, it would need to call the copy constructor to do so. But calling the copy constructor again would mean the parameter is passed by value again, requiring another call to the copy constructor. This would result in an infinite recursion (well, until the stack memory ran out and the the program crashed).

A **shallow copy** means that C++ copies each member of the class individually using the assignment operator. When classes are simple (eg. do not contain any dynamically allocated memory), this works very well.

However, when designing classes that handle dynamically allocated memory, memberwise (shallow) copying can get us in a lot of trouble! This is because the standard pointer assignment operator just copies the address of the pointer — it does not allocate any memory or copy the contents being pointed to!

A **deep copy** duplicates the object or variable being pointed to so that the destination (the object being assigned to) receives it's own local copy. This way, the destination can do whatever it wants to it's local copy and the object that was copied from will not be affected. Doing deep copies requires that we write our own copy constructors and overloaded assignment operators.

Summary

- The default copy constructor and default assignment operators do shallow copies, which is fine for classes that contain no dynamically allocated variables.
- Classes with dynamically allocated variables need to have a copy constructor and assignment operator that do a deep copy.
- The assignment operator is usually implemented using the same code as the copy constructor, but it checks for self-assignment, returns *this, and deallocates any previously allocated memory before deep copying.
- If you don't want a class to be copyable, use a private copy constructor and assignment operator prototype in the class header.

" (LearnCPP, 2014)

VIRTUAL TABLE

"To implement virtual functions, C++ uses a special form of late binding known as the virtual table. The **virtual table** is a lookup table of functions used to resolve function calls in a dynamic/late binding manner. The virtual table sometimes goes by other names, such as "vtable", "virtual function table", "virtual method table", or "dispatch table".

Because knowing how the virtual table works is not necessary to use virtual functions, this section can be considered optional reading.

The virtual table is actually quite simple, though it's a little complex to describe in words. First, every class that uses virtual functions (or is derived from a class that uses virtual functions) is given it's own virtual table. This table is simply a static array that the compiler sets up at compile time. A virtual table contains one entry for each virtual function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the most-derived function accessible by that class.

Second, the compiler also adds a hidden pointer to the base class, which we will call *__vptr. *__vptr is set (automatically) when a class instance is created so that it points to

the virtual table for that class. Unlike the *this pointer, which is actually a function parameter used by the compiler to resolve self-references, *__vptr is a real pointer. Consequently, it makes each class object allocated bigger by the size of one pointer. It also means that *__vptr is inherited by derived classes, which is important.

By using these tables, the compiler and program are able to ensure function calls resolve to the appropriate virtual function, even if you're only using a pointer or reference to a base class!

Calling a virtual function is slower than calling a non-virtual function for a couple of reasons: First, we have to use the *__vptr to get to the appropriate virtual table. Second, we have to index the virtual table to find the correct function to call. Only then can we call the function. As a result, we have to do 3 operations to find the function to call, as opposed to 2 operations for a normal indirect function call, or one operation for a direct function call. However, with modern computers, this added time is usually fairly insignificant." (LearnCPP, 2014)

TEMPLATES

"**What is a function template?**

If you were to look up the word "template" in the dictionary, you'd find a definition that was similar to the following: "a template is a model that serves as a pattern for creating similar objects". One type of template that is very easy to understand is that of a stencil. A stencil is an object (eg. a piece of cardboard) with a shape cut out of it (eg. the letter J). By placing the stencil on top of another object, then spraying paint through the hole, you can very quickly produce stenciled patterns in many different colors! Note that you only need to create a given stencil once — you can then use it as many times as you like to create stenciled patterns in whatever color(s) you like. Even better, you don't have to decide the color of the stenciled pattern you want to create until you decide to actually use the stencil.

In C++, **function templates** are functions that serve as a pattern for creating other similar functions. The basic idea behind function templates is to create a function without having to specify the exact type(s) of some or all of the variables. Instead, we define the function using placeholder types, called **template type parameters**. Once we have created a function using these placeholder types, we have effectively created a "function stencil".

It turns out that you can't call a function template directly — this is because the compiler doesn't know how to handle placeholder types directly. Instead, when you call a template function, the compiler "stencils" out a copy of the template, replacing the placeholder types with the actual variable types in your function call! Using this methodology, the compiler can create multiple "flavors" of a function from one template! We'll take a look at this process in more detail in the next lesson.

template functions can save a lot of time, because you only need to write one function, and it will work with many different types. Once you get used to writing function templates, you'll find they actually don't take any longer to write than functions with actual types. Template functions reduce code maintenance, because duplicate code is reduced significantly. And finally, template functions can be safer, because there is no need to copy functions and change types by hand whenever you need the function to work with a new type!

Template functions do have a few drawbacks, and we would be remiss not to mention them. First, older compilers generally do not have very good template support. However, modern compilers are much better at supporting and implementing template functionality properly. Second, template functions produce crazy-looking error messages that are much harder to decipher than those of regular functions. However, these drawbacks are fairly minor compared with the power and flexibility templates bring to your programming toolkit!

" (LearnCPP, 2014)

Work in progress

EXCEPTIONS

Work in progress

THE STANDARD TEMPLATE LIBRARY

Work in progress

OPTIMIZATION AND GUIDES

C++ BASICS – PART 2

Things to Remember

- **Prefer consts, enums, and inlines to #defines**
  - For simple constants, prefer const objects or enums to #defines.
  - For function-like macros, prefer inline functions to #defines.
- **Use const whenever possible**
  - Declaring something const helps compilers detect usage errors. const can be applied to objects at any scope, to function parameters and return types, and to member functions as a whole.
  - Compilers enforce bitwise constness, but you should program using conceptual constness.
  - When const and non-const member functions have essentially identical implementations, code duplication can be avoided by having the non-const version call the const version.
- **Make sure that objects are initialized before they're used**

- Manually initialize objects of built-in type, because C++ only sometimes initializes them itself.
- In a constructor, prefer use of the member initialization list to assignment inside the body of the constructor.List data members in the initialization list in the same order they're declared in the class.
- Avoid initialization order problems across translation units by replacing non-local static objects with local static objects.

PREFER CONSTS, ENUMS, AND INLINES TO #DEFINES

#DEFINE

Change:

#define ASPECT_RATIO 1.653

To

const double AspectRatio = 1.653;

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

HEADER CONSTANTS

Declare the pointer const, in addition to what the pointer points to.

const char * const sometext = "Hello World!";

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

CLASS-SPECIFIC CONSTANTS

To limit the scope of a constant to a class, you must

make it a member, and to ensure there's at most one copy of the constant, you must make it a static member:

class MyClass {

private:

static const int dataCount = 5; // constant declaration

int arrayOfData[dataCount]; // use of constant

...

};

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

PREFER INLINE FUNCTIONS TO #DEFINES

```
// call f with the maximum of a and b
#define CALL_WITH_MAX(a, b) f((a) > (b) ? (a) : (b))
int a = 5, b = 0;
CALL_WITH_MAX(++a, b); // a is incremented twice
CALL_WITH_MAX(++a, b+10); // a is incremented once
```

```
template<typename T> // because we don't
inline void callWithMax(const T& a, const T& b) // know what T is, we
{ // pass by reference-tof(
a > b ? a : b); // const — see Item 20
}
```

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

USE CONST WHENEVER POSSIBLE

```
char greeting[] = "Hello";
char *p = greeting; // non-const pointer,
// non-const data
const char *p = greeting; // non-const pointer,
// const data
char * const p = greeting; // const pointer,
// non-const data
const char * const p = greeting; // const pointer,
// const data
```

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

MAKE SURE THAT OBJECTS ARE INITIALIZED BEFORE THEY'RE USED

Do not simply assign values in a constructor of a class but use member initialization:

So Instead of

```
VECTOR2D::VECTOR2D()
		{
			this->X = 0;
			this->Y = 0;
		}
```

Do this:

```
VECTOR2D::VECTOR2D() : X(0), Y(0)
		{

		}
```

CONSTRUCTORS, DESTRUCTORS, AND ASSIGNMENT OPERATORS

- **Know what functions C++ silently writes and calls**
  - Compilers may implicitly generate a class's default constructor, copy constructor, copy assignment operator, and destructor.
- **Explicitly disallow the use of compiler-generated functions you do not want**
  - To disallow functionality automatically provided by compilers, declare the corresponding member functions private and give no implementations. Using a base class is one way to do this.
- **Declare destructors virtual in polymorphic base classes**
  - Polymorphic base classes should declare virtual destructors. If a class has any virtual functions, it should have a virtual destructor.
  - Classes not designed to be base classes or not designed to be used polymorphically should not declare virtual destructors.
- **Prevent exceptions from leaving destructors**
  - Destructors should never emit exceptions. If functions called in a destructor may throw, the destructor should catch any exceptions, then swallow them or terminate the program.
  - If class clients need to be able to react to exceptions thrown during an operation, the class should provide a regular (i.e., non-destructor) function that performs the operation.
- **Never call virtual functions during construction or destruction**
  - Don't call virtual functions during construction or destruction, because such calls will never go to a more derived class than that of the currently executing constructor or destructor. In other words, since you can't use virtual functions to call down from base classes during construction, you can compensate by having derived classes pass necessary construction information up to base class constructors instead.
- **Have assignment operators return a reference to *this**

    - Have assignment operators return a reference to *this.
- **Handle assignment to self in operator=**
    - Make sure operator= is well-behaved when an object is assigned to itself. Techniques include comparing addresses of source and target objects, careful statement ordering, and copy-and-swap.
    - Make sure that any function operating on more than one object behaves correctly if two or more of the objects are the same.
- **Copy all parts of an object**
    - Copying functions should be sure to copy all of an object's data members and all of its base class parts.
    - Don't try to implement one of the copying functions in terms of the other. Instead, put common functionality in a third function that both call. If you find that your copy constructor and copy assignment operator have similar code bodies, eliminate the duplication by creating a third member function that both call. Such a function is typically private and is often named init. This strategy is a safe, proven way to eliminate code duplication in copy constructors and copy assignment operators.

DECLARE DESTRUCTORS VIRTUAL IN POLYMORPHIC BASE CLASSES

A class that is going to be used as a base class:

```
class PMover : virtual public PObjectBaseBasic
		{

		public:
			Mover MoverObject;
			PMover(const mRECT &areaSize);
			virtual ~PMover();
			virtual void Render();


		};
```

A class that is not going to be used as a base class (NOTICE: No Virtual destructor):

```
class Mover : virtual public RandomBaseComplete
		{

		private:
			mutable VECTOR2D position;
			mutable VECTOR2D velocity;
			mutable VECTOR2D acceleration;
			VECTOR2D target;
```

```
        public:
                Mover(const mRECT &areaSize);
                Mover(const VECTOR2D &position, const VECTOR2D &velocity, const
mRECT &areaSize);
                Mover(const VECTOR2D &position, const VECTOR2D &velocity, const
VECTOR2D &acceleration, const mRECT &areaSize);
                ~Mover();




        };
```

PREVENT EXCEPTIONS FROM LEAVING DESTRUCTORS

Sample to move the exception rise in a separate function that keeps a state if the connection has been closed and create a functionality in the destructor so that the DB connection is closed only if it has not been closed or failed before. This way the user of the class can handle the exception when they use the close function and the destructor will try to release resources and log them properly if errors occur.

```
class DBConn {

public:

...

void close() // new function for

{ // client use

db.close();

closed = true;

}

~DBConn()

{

if (!closed) {

try { // close the connection

db.close(); // if the client didn't

}

catch (...) { // if closing fails,
```

```
make log entry that call to close failed; // note that and
... // terminate or swallow
}
}
private:
DBConnection db;
bool closed;
};
```

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

HANDLE ASSIGNMENT TO SELF IN OPERATOR=

Example 1:

```
Widget& Widget::operator=(const Widget& rhs)
{
if (this == &rhs) return *this; // identity test: if a self-assignment,
// do nothing
delete pb;
pb = new Bitmap(*rhs.pb);
return *this;
}
```

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

Example 2:

```
Widget& Widget::operator=(const Widget& rhs)
{
Bitmap *pOrig = pb; // remember original pb
pb = new Bitmap(*rhs.pb); // make pb point to a copy of *pb
delete pOrig; // delete the original pb
```

```
return *this;

}
```

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

Example 3:

```
Widget& Widget::operator=(const Widget& rhs)

{

Widget temp(rhs); // make a copy of rhs's data

swap(temp); // swap *this's data with the copy's

return *this;

}
```

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

RESOURCE MANAGEMENT

Memory is only one of many resources you must manage. Other common resources include file descriptors, mutex locks, fonts and brushes in graphical user interfaces (GUIs), database connections, and network sockets. Regardless of the resource, it's important that it be released when you're finished with it.

- **Use objects to manage resources**
  - To prevent resource leaks, use RAII objects that acquire resources in their constructors and release them in their destructors.
  - Two commonly useful RAII classes are TR1::shared_ptr and auto_ptr. tr1::shared_ptr is usually the better choice, because its behavior when copied is intuitive. Copying an auto_ptr sets it to null.
- **Think carefully about copying behavior in resource-managing classes**
  - Copying an RAII object entails copying the resource it manages, so the copying behavior of the resource determines the copying behavior of the RAII object.
  - Common RAII class copying behaviors are disallowing copying and performing reference counting, but other behaviors are possible.
- **Provide access to raw resources in resource-managing classes**
  - If you use [] in a new expression, you must use [] in the corresponding delete expression. If you don't use [] in a new expression, you mustn't use [] in the corresponding delete expression.
- **Store newed objects in smart pointers in standalone statements**
  - Store newed objects in smart pointers in standalone statements. Failure to do this can lead to subtle resource leaks when exceptions are thrown.
  - By putting resources inside objects, we can rely on C++'s automatic destructor invocation to make sure that the resources are released.

USE OBJECTS TO MANAGE RESOURCES

```
void f()
{
std::auto_ptr<Investment> pInv(createInvestment()); // call factory
// function
... // use pInv as
// before
}
```

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

This simple example demonstrates the two critical aspects of using objects to manage resources:

- **Resources are acquired and immediately turned over to resource-managing objects.** Above, the resource returned by createInvestment is used to initialize the auto_ptr that will manage it. In fact, the idea of using objects to manage resources is often called Resource Acquisition Is Initialization (RAII), because it's so common to acquire a resource and initialize a resource-managing object in the same statement. Sometimes acquired resources are assigned to resource-managing objects instead of initializing them, but either way, every resource is immediately turned over to a resource-managing object at the time the resource is acquired.
- **Resource-managing objects use their destructors to ensure that resources are released.** Because destructors are called automatically when an object is destroyed (e.g., when an object goes out of scope), resources are correctly released, regardless of how control leaves a block. Things can get tricky when the act of releasing resources can lead to exceptions being thrown, but that's a matter addressed by Item 8, so we'll not worry about it here.

Because an auto_ptr automatically deletes what it points to when the auto_ptr is destroyed, it's important that there never be more than one auto_ptr pointing to an object. Using auto_ptr or TR1::shared_ptr with dynamically allocated arrays is a bad idea, though, regrettably, one that will compile.

STORE NEWED OBJECTS IN SMART POINTERS IN STANDALONE STATEMENTS

Instead of this:

```
processWidget(std::tr1::shared_ptr<Widget>(new Widget), priority());
```

Use this:

```
std::tr1::shared_ptr<Widget> pw(new Widget); // store newed object
```

// in a smart pointer in a

// standalone statement

processWidget(pw, priority()); // this call won't leak

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

DESIGNS AND DECLARATIONS

- Good interfaces are easy to use correctly and hard to use incorrectly. Your should strive for these characteristics in all your interfaces.
- Ways to facilitate correct use include consistency in interfaces and behavioral compatibility with built-in types.
- Ways to prevent errors include creating new types, restricting operations on types, constraining object values, and eliminating client resource management responsibilities.
- TR1::shared_ptr supports custom deleters. This prevents the cross-DLL problem, can be used to automatically unlock mutexes etc.
- Prefer pass-by-reference-to-const over pass-by-value. It's typically more efficient and it avoids the slicing problem. Sample: void printNameAndDisplay(const Window& w);
- The rule doesn't apply to built-in types and STL iterator and function object types. For them, passby-value is usually appropriate.
- Never return a pointer or reference to a local stack object, a reference to a heap-allocated object, or a pointer or reference to a local static object if there is a chance that more than one such object will be needed.
- Declare data members private. It gives clients syntactically uniform access to data, affords fine-grained access control, allows invariants to be enforced, and offers class authors implementation flexibility. This is to avoid problem if and when you might change you data member and how they work. This will not cause implementers to stop working or cause problems.
- protected is no more encapsulated than public.
- Prefer non-member non-friend functions to member functions. Doing so increases encapsulation, packaging flexibility, and functional extensibility.
- If you need type conversions on all parameters to a function (including the one pointed to by the this pointer), the function must be a non-member.
- Provide a swap member function when std::swap would be inefficient for your type. Make sure your swap doesn't throw exceptions.
- If you offer a member swap, also offer a non-member swap that calls the member. For classes (not templates), specialize std::swap, too.
- When calling swap, employ a using declaration for std::swap, then call swap without namespace qualification.
- It's fine to totally specialize std templates for user-defined types, but never try to add something completely new to std.

TREAT CLASS DESIGN AS TYPE DESIGN

How, then, do you design effective classes? First, you must understand the issues you face. Virtually every class requires that you confront the following questions, the answers to which often lead to constraints on your design:

How should objects of your new type be created and destroyed ? How this is done influences the design of your class's constructors and destructor, as well as its memory allocation and deallocation functions (operator new, operator new[], operator delete, and operator delete[]), if you write them.

How should object initialization differ from object assignment? The answer to this question determines the behavior of and the differences between your constructors and your assignment operators. It's important not to confuse initialization with assignment, because they correspond to different function calls (see Item 4).

What does it mean for objects of your new type to be passed by value ? Remember, the copy constructor defines how pass-by-value is implemented for a type.

What are the restrictions on legal values for your new type? Usually, only some combinations of values for a class's data members are valid. Those combinations determine the invariants your class will have to maintain. The invariants determine the error checking you'll have to do inside your member functions, especially your constructors, assignment operators, and "setter" functions. It may also affect the exceptions your functions throw and, on the off chance you use them, your functions' exception specifications.

Does your new type fit into an inheritance graph? If you inherit from existing classes, you are constrained by the design of those classes, particularly by whether their functions are virtual or non-virtual. If you wish to allow other classes to inherit from your class, that affects whether the functions you declare are virtual, especially your destructor (see Item 7).

What kind of type conversions are allowed for your new type ? Your type exists in a sea of other types, so should there be conversions between your type and other types? If you wish to allow objects of type T1 to be implicitly converted into objects of type T2, you will want to write either a type conversion function in class T1 (e.g., operator T2) or a non-explicit constructor in class T2 that can be called with a single argument. If you wish to allow explicit conversions only, you'll want to write functions to perform the conversions, but you'll need to avoid making them type conversion operators or non-explicit constructors that can be called with one argument. (For an example of both implicit and explicit conversion functions.)

What operators and functions make sense for the new type? The answer to this question determines which functions you'll declare for your class. Some functions will be member functions, but some will not.

What standard functions should be disallowed? Those are the ones you'll need to declare private.

Who should have access to the members of your new type ? This question helps you determine which members are public, which are protected, and which are private. It also helps you determine which classes and/or functions should be friends, as well as whether it makes sense to nest one class inside another.

What is the "undeclared interface" of your new type ? What kind of guarantees does it offer with respect to performance, exception safety (see Item 29), and resource usage (e.g., locks and dynamic memory)? The guarantees you offer in these areas will impose constraints on your class implementation.

How general is your new type ? Perhaps you're not really defining a new type. Perhaps you're defining a whole *family* of types. If so, you don't want to define a new class, you want to define a new class *template*.

Is a new type really what you need? If you're defining a new derived class only so you can add functionality to an existing class, perhaps you'd better achieve your goals by simply defining one or more non-member functions or templates.

DON'T TRY TO RETURN A REFERENCE WHEN YOU MUST RETURN AN OBJECT

```
inline const Rational operator*(const Rational& lhs, const Rational& rhs)

{

return Rational(lhs.n * rhs.n, lhs.d * rhs.d);

}
```

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

DECLARE NON-MEMBER FUNCTIONS WHEN TYPE CONVERSIONS SHOULD APPLY TO ALL PARAMETERS

Instead of:

```
class Rational {

public:

Rational(int numerator = 0, // ctor is deliberately not explicit;

int denominator = 1); // allows implicit int-to-Rational

// conversions

int numerator() const; // accessors for numerator and

int denominator() const; // denominator — see Item 22

const Rational operator*(const Rational& rhs) const;

private:

...

};
```

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

Error situation:

```
result = oneHalf * 2; // fine
result = 2 * oneHalf; // error!
```

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

Use this instead:

```
class Rational {
... // contains no operator*
};
const Rational operator*(const Rational& lhs, // now a non-member
const Rational& rhs) // function
{
return Rational(lhs.numerator() * rhs.numerator(),
lhs.denominator() * rhs.denominator());
}
Rational oneFourth(1, 4);
Rational result;

result = oneFourth * 2; // fine
result = 2 * oneFourth; // hooray, it works!
```

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

IMPLEMENTATIONS

- **Postpone variable definitions as long as possible**
  - Postpone variable definitions as long as possible. It increases program clarity and improves program efficiency. You should postpone a variable's definition until right before you have to use the variable, you should also try to postpone the definition until you have initialization arguments for it. By doing so, you avoid constructing and destructing unneeded objects, and you avoid unnecessary default constructions. Further, you help document the purpose of variables by initializing them in contexts in which their meaning is clear.
- **Minimize casting**
  - Avoid casts whenever practical, especially dynamic_casts in performance-sensitive code. If a design requires casting, try to develop a cast-free alternative.
  - When casting is necessary, try to hide it inside a function. Clients can then call the function instead of putting casts in their own code.
  - Prefer C++-style casts to old-style casts. They are easier to see, and they are more specific about what they do.

- **Avoid returning "handles" to object internals**
    - Avoid returning handles (references, pointers, or iterators) to object internals. It increases encapsulation, helps const member functions act const, and minimizes the creation of dangling handles.
- **Strive for exception-safe code**
    - Exception-safe functions leak no resources and allow no data structures to become corrupted, even when exceptions are thrown. Such functions offer the basic, strong, or nothrow guarantees.
    - The strong guarantee can often be implemented via copy-and-swap, but the strong guarantee is not practical for all functions.
    - A function can usually offer a guarantee no stronger than the weakest guarantee of the functions it calls.
- **Understand the ins and outs of inlining**
    - Limit most inlining to small, frequently called functions. This facilitates debugging and binary upgradability, minimizes potential code bloat, and maximizes the chances of greater program speed.
    - Don't declare function templates inline just because they appear in header files.
- **Minimize compilation dependencies between files**
    - The general idea behind minimizing compilation dependencies is to depend on declarations instead of definitions. Two approaches based on this idea are Handle classes and Interface classes.
    - Library header files should exist in full and declaration-only forms. This applies regardless of whether templates are involved.

EXCEPTION-SAFE FUNCTIONS

- Functions offering **the basic guarantee** promise that if an exception is thrown, everything in the program remains in a valid state. No objects or data structures become corrupted, and all objects are in an internally consistent state (e.g., all class invariants are satisfied). However, the exact state of the program may not be predictable. For example, we could write changeBackground so that if an exception were thrown, the PrettyMenu object might continue to have the old background image, or it might have some default background image, but clients wouldn't be able to predict which. (To find out, they'd presumably have to call some member function that would tell them what the current background image was.)
- Functions offering **the strong guarantee** promise that if an exception is thrown, the state of the program is unchanged. Calls to such functions are *atomic* in the sense that if they succeed, they succeed completely, and if they fail, the program state is as if they'd never been called. Working with functions offering the strong guarantee is easier than working with functions offering only the basic guarantee, because after calling a function offering the strong guarantee, there are only two possible program states: as expected following successful execution of the function, or the state that existed at the time the function was called. In contrast, if a call to a function offering only the basic guarantee yields an exception, the program could be in *any* valid state.
- Functions offering the nothrow guarantee promise never to throw exceptions, because they always do what they promise to do. All operations on built-in types (e.g., ints, pointers, etc.) are nothrow (i.e., offer the nothrow guarantee). This is a critical building block of exception-safe code.

UNDERSTAND INLINING

IMPLICIT INLINE EXAMPLE

```
class Person {

public:

...

int age() const { return theAge; } // an implicit inline request: age is

... // defined in a class definition

private:

int theAge;

};
```

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

EXPLICIT INLINE EXAMPLE

```
template<typename T> // an explicit inline

inline const T& std::max(const T& a, const T& b) // request: std::max is

{ return a < b ? b : a; } // preceded by "inline"
```

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

NOTES ON INLINING

Before we do that, let's finish the observation that inline is a request that compilers may ignore. Most compilers refuse to inline functions they deem too complicated (e.g., those that contain loops or are recursive), and all but the most trivial calls to virtual functions defy inlining. This latter observation shouldn't be a surprise. virtual means "wait until runtime to figure out which function to call," and inline means "before execution, replace the call site with the called function." If compilers don't know which function will be called, you can hardly blame them for refusing to inline the function's body.

Whether a given inline function is actually inlined depends on the build environment you're using — primarily on the compiler.

Don't forget the empirically determined rule of 80-20, which states that atypical program spends 80% of its time executing only 20% of its code. It's an important rule, because it reminds you that your goal as a software developer is to identify the 20% of your code that can increase your program's overall performance. You can inline and otherwise tweak your functions until the cows come home, but it's wasted effort unless you're focusing on the *right* functions.

MINIMIZE COMPILATION DEPENDENCIES BETWEEN FILES

- **Avoid using objects when object references and pointers will do**. You may define references and pointers to a type with only a declaration for the type. Defining *objects* of a type necessitates the presence of the type's definition.
- **Depend on class declarations instead of class definitions whenever you can**. Note that you *never* need a class definition to declare a function using that class, not even if the function passes or returns the class type by value.
- **Provide separate header files for declarations and definitions**. In order to facilitate adherence to the above guidelines, header files need to come in pairs: one for declarations, the other for definitions. These files must be kept consistent, of course. If a declaration is changed in one place, it must be changed in both. As a result, library clients should always #include a declaration file instead of forward-declaring something themselves, and library authors should provide both header files.

Use Handle classes and Interface classes during development to minimize the impact on clients when implementations change. Replace Handle classes and Interface classes with concrete classes for production use when it can be shown that the difference in speed and/or size is significant enough to justify the increased coupling between classes.

INHERITANCE AND OBJECT-ORIENTED DESIGN

- **Make sure public inheritance models "is-a."**
  - Public inheritance means "is-a." Everything that applies to base classes must also apply to derived classes, because every derived class object *is* a base class object.
- **Avoid hiding inherited names**
  - Names in derived classes hide names in base classes. Under public inheritance, this is never desirable.
  - To make hidden names visible again, employ using declarations or forwarding functions.
- **Differentiate between inheritance of interface and inheritance of implementation**
  - Inheritance of interface is different from inheritance of implementation. Under public inheritance, derived classes always inherit base class interfaces.
  - Pure virtual functions specify inheritance of interface only.
  - Simple (impure) virtual functions specify inheritance of interface plus inheritance of a default implementation.
  - Non-virtual functions specify inheritance of interface plus inheritance of a mandatory implementation.
- **Consider alternatives to virtual functions**
  - Alternatives to virtual functions include the NVI idiom and various forms of the Strategy design pattern. The NVI idiom is itself an example of the Template Method design pattern.
  - A disadvantage of moving functionality from a member function to a function outside the class is that the non-member function lacks access to the class's non-public members.
  - tr1::function objects act like generalized function pointers. Such objects support all callable entities compatible with a given target signature.
- **Never redefine an inherited non-virtual function**
  - Never redefine an inherited non-virtual function.
- **Never redefine a function's inherited default parameter value**
  - Never redefine an inherited default parameter value, because default parameter values are statically bound, while virtual functions — the only functions you should be overriding — are dynamically bound.

- **Model "has-a" or "is-implemented-in-terms-of" through composition**
  - Composition has meanings completely different from that of public inheritance.
  - In the application domain, composition means has-a. In the implementation domain, it means is-implemented-in-terms-of.
- **Use private inheritance judiciously**
  - Private inheritance means is-implemented-in-terms of. It's usually inferior to composition, but it makes sense when a derived class needs access to protected base class members or needs to redefine inherited virtual functions.
  - Unlike composition, private inheritance can enable the empty base optimization. This can be important for library developers who strive to minimize object sizes.
- **Use multiple inheritance judiciously**
  - Multiple inheritance is more complex than single inheritance. It can lead to new ambiguity issues and to the need for virtual inheritance.
  - Virtual inheritance imposes costs in size, speed, and complexity of initialization and assignment. It's most practical when virtual base classes have no data.
  - Multiple inheritance does have legitimate uses. One scenario involves combining public inheritance from an Interface class with private inheritance from a class that helps with implementation.
    - **Sample**: class CPerson: **public IPerson, private PersonInfo** {…}

MAKE SURE PUBLIC INHERITANCE MODELS "IS-A."

```
class Person {...};

class Student: public Person {...};
```

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

Every student is a person, but not every person is a student. That is exactly what this hierarchy asserts. We expect that anything that is true of a person — for example, that he or she has a date of birth — is also true of a student. We do not expect that everything that is true of a student — that he or she is enrolled in a particular school, for instance — is true of people in general. The notion of a person is more general than is that of a student; a student is a specialized type of person.

DIFFERENTIATE BETWEEN INHERITANCE OF INTERFACE AND INHERITANCE OF IMPLEMENTATION

(public) inheritance: composed of two separable parts: inheritance of function interfaces and inheritance of function implementations. The difference between these two kinds of inheritance corresponds exactly to the difference between function declarations and function definitions. There are only two kinds of functions you can inherit: virtual and non-virtual.

DESIGNING CLASSES

- **Pure virtual function:** The purpose of declaring a pure virtual function is to have derived classes inherit a function interface only. They must be redeclared by any concrete class that inherits them, and they typically have no definition in abstract classes.

    - **Usage idea:** You sometimes want derived classes to inherit only the interface (declaration) of a member function. => interface only
- **Simple virtual function:** The purpose of declaring a simple virtual function is to have derived classes inherit a function interface as well as a default implementation. Derived classes inherit the interface of the function, but simple virtual functions provide an implementation that derived classes may override. The interface says that every class must support a function to be called when needed, but each class is free to define how the function works.
    - **Usage idea:** Sometimes you want derived classes to inherit both a function's interface and implementation, but you want to allow them to override the implementation they inherit. => interface and a default implementation
- **Non-virtual function:** The purpose of declaring a non-virtual function is to have derived classes inherit a function interface as well as a mandatory implementation. When a member function is non-virtual, it's not supposed to behave differently in derived classes. In fact, a nonvirtual member function specifies an *invariant over specialization*, because it identifies behavior that is not supposed to change, no matter how specialized a derived class becomes.
    - **Usage idea:** And sometimes you want derived classes to inherit a function's interface and implementation without allowing them to override anything. => interface and a mandatory implementation

COMMON MISTAKES

1. The first mistake is to declare all functions non-virtual. That leaves no room for specialization in derived classes; nonvirtual destructors are particularly problematic. it's perfectly reasonable to design a class that is not intended to be used as a base class. In that case, a set of exclusively non-virtual member functions is appropriate.
2. The other common problem is to declare all member functions virtual. Sometimes this is the right thing to do such with Interface classes but this may be to the lack of courage on the designer to take a stand on what he wants to say with the class design, how it should be used.

ALTERNATIVES TO VIRTUAL FUNCTIONS

Some alternatives:

- Use the **non-virtual interface idiom** (NVI idiom), a form of the Template Method design pattern that wraps public non-virtual member functions around less accessible virtual functions.
- Replace virtual functions with **function pointer data members**, a stripped-down manifestation of the Strategy design pattern.
- Replace virtual functions with **tr1::function data members**, thus allowing use of any callable entity with a signature compatible with what you need. This, too, is a form of the Strategy design pattern.
- Replace virtual functions in one hierarchy with **virtual functions in another hierarchy**. This is the conventional implementation of the Strategy design pattern.

NEVER REDEFINE AN INHERITED NON-VIRTUAL FUNCTION

Non-virtual functions have two-faced behavior. Non-virtual functions are statically bound. That means that because pB is declared to be of type pointer-to- B, non-virtual functions invoked through pB will always be those defined for class B, even if pB points to an object of a class derived from B, as it does in this example.

Virtual functions, on the other hand, are dynamically bound so they don't suffer from this problem. If mf were a virtual function, a call to mf through either pB or pD would result in an invocation of D::mf, because what pB and pD *really* point to is an object of type D.

If you are writing class D and you redefine a non-virtual function mf that you inherit from class B, D objects will likely exhibit inconsistent behavior. In particular, any given D object may act like either a B or a D when mf is called, and the determining factor will have nothing to do with the object itself, but with the declared type of the pointer that points to it. References exhibit the same baffling behavior as do pointers.

Explanation:

Declaring a non-virtual function in a class establishes an invariant over specialization for that class. If you apply these observations to the classes B and D and to the non-virtual member function B::mf, then:

- Everything that applies to B objects also applies to D objects, because every D object is-a B object;
- Classes derived from B must inherit both the interface *and* the implementation of mf, because mf is nonvirtual in B.

Now, if D redefines mf, there is a contradiction in your design. If D really needs to implement mf differently from B, and if every B object — no matter how specialized — really has to use the B implementation for mf, then it's simply not true that every D is-a B. In that case, D shouldn't publicly inherit from B. On the other hand, if D really has to publicly inherit from B, and if D really needs to implement mf differently from B, then it's just not true that mf reflects an invariant over specialization for B. In that case, mf should be virtual. Finally, if every D really is-a B, and if mf really corresponds to an invariant over specialization for B, then D can't honestly need to redefine mf, and it shouldn't try to.

Sample:

```
class B {

public:

void mf();

...

};

class D: public B {

void mf();  // hides B::mf

 };


D x; // x is an object of type D

B *pB = &x; // get pointer to x


pB->mf(); // call mf through pointer
```

```
D *pD = &x; // get pointer to x

pD->mf(); // call mf through pointer
```

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

NEVER REDEFINE A FUNCTION'S INHERITED DEFAULT PARAMETER VALUE

Virtual functions are dynamically bound, but default parameter values are statically bound.

An object's dynamic type is determined by the type of the object to which it currently refers. That is, its dynamic type indicates how it will behave. Dynamic types, as their name suggests, can change as a program runs, typically through assignments.

Virtual functions are dynamically bound, meaning that the particular function called is determined by the dynamic type of the object through which it's invoked.

Explanation:

This means that you may end up invoking a virtual function defined in a derived class but using a default parameter value from a base class.

Why does C++ insist on acting in this perverse manner? The answer has to do with runtime efficiency. If default parameter values were dynamically bound, compilers would have to come up with a way to determine the appropriate default value(s) for parameters of virtual functions at runtime, which would be slower and more complicated than the current mechanism of determining them during compilation. The decision was made to err on the side of speed and simplicity of implementation, and the result is that you now enjoy execution behavior that is efficient, but, if you fail to heed the advice of this Item, confusing.

Notice: If you try to follow this rule and also offer default parameter values to users of both base and derived classes. If the default parameter value is changed in the base class, all derived classes that repeat it must also be changed. Otherwise they'll end up redefining an inherited default parameter value.

MODEL "HAS-A" OR "IS-IMPLEMENTED-IN-TERMS-OF" THROUGH COMPOSITION

Composition => It's also known as layering, containment, aggregation, and embedding.

Composition has a meaning, too. Actually, it has two meanings. Composition means either "has-a" or "is-implemented-in-terms-of." That's because you are dealing with two different domains in your software:

- Some objects in your programs correspond to things in the world you are modeling, e.g., people, vehicles, video frames, etc. Such objects are part of the application domain. => When composition occurs between objects in the application domain, it expresses a has-a relationship.
- Other objects are purely implementation artifacts, e.g., buffers, mutexes, search trees, etc. These kinds of objects correspond to your software's implementation domain. When it occurs in the implementation domain, it expresses an is-implemented-in-terms-of relationship.

EXAMPLE OF HAS-A RELATIONSHIP

CODE SAMPLE

```
class Address { ... }; // where someone lives

class PhoneNumber { ... };

class Person {

public:

...

private:

std::string name; // composed object

Address address; // ditto

PhoneNumber voiceNumber; // ditto

PhoneNumber faxNumber; // ditto

};
```

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

EXPLANATION

The Person class above demonstrates the has-a relationship. A Person object has a name, an address, and voice and fax telephone numbers. You wouldn't say that a person is a name or that a person is an address. You would say that a person has a name and has an address. Most people have little difficulty with this distinction, so confusion between the roles of is-a and has-a is relatively rare.

EXAMPLE OF IS-IMPLEMENTED-IN-TERMS-OF

WRONG WAY:

```
template<typename T> // the wrong way to use list for Set

class Set: public std::list<T> { ... };
```

(Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)

RIGHT WAY:

```
template<class T> // the right way to use list for Set

class Set {

public:

bool member(const T& item) const;

void insert(const T& item);

void remove(const T& item);
```

```
std::size_t size() const;

private:

std::list<T> rep; // representation for Set data

}; (Meyers, Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs, 2005)
```

USE PRIVATE INHERITANCE JUDICIOUSLY

Private inheritance means is-implemented-in-terms-of.

If you make a class D privately inherit from a class B, you do so because you are interested in taking advantage of some of the features available in class B, not because there is any conceptual relationship between objects of types B and D. As such, private inheritance is purely an implementation technique. (That's why everything you inherit from a private base class becomes private in your class: it's all just implementation detail.) Private inheritance means that implementation *only* should be inherited; interface should be ignored. If D privately inherits from B, it means that D objects are implemented in terms of B objects, nothing more. Private inheritance means nothing during software *design*, only during software *implementation*.

In contrast to public inheritance, compilers will generally *not* convert a derived class object (such as Student) into a base class object (such as Person) if the inheritance relationship between the classes is private. That's why the call to eat fails for the object crated from Student. The second rule is that members inherited from a private base class become private members of the derived class, even if they were protected or public in the base class.

Private inheritance is most likely to be a legitimate design strategy when you're dealing with two classes not related by is-a where one either needs access to the protected members of another or needs to redefine one or more of its virtual functions. Even in that case, we've seen that a mixture of public inheritance and containment can often yield the behavior you want, albeit with greater design complexity. Using private inheritance *judiciously* means employing it when, having considered all the alternatives, it's the best way to express the relationship between two classes in your software.

HOW TO CHOOSE BETWEEN IS-IMPLEMENTED-IN-TERMS-OF OF PRIVATE INHERITANCE AND COMPOSITION

The fact that private inheritance means is-implemented-in-terms-of is a little disturbing, because as earlier pointed out that composition can mean the same thing. How are you supposed to choose between them? The answer is simple: use composition whenever you can, and use private inheritance whenever you must. When must you? Primarily when protected members and/or virtual functions enter the picture, though there's also an edge case where space concerns can tip the scales toward private inheritance.

ALTERNATIVES TO IS-IMPLEMENTED-IN-TERMS-OF OF PRIVATE INHERITANCE

- public inheritance plus composition

MODERN C++ (C++11, C++14)

THE BIGGEST CHANGES IN C++11 (AND WHY YOU SHOULD CARE)

“IT’S BEEN 13 YEARS SINCE THE FIRST ITERATION OF THE C++ LANGUAGE. DANNY KALEV, A FORMER MEMBER OF THE C++ STANDARDS COMMITTEE, EXPLAINS HOW THE PROGRAMMING LANGUAGE HAS BEEN IMPROVED AND HOW IT CAN HELP YOU WRITE BETTER CODE.

Bjarne Stroustrup, the creator of C++, said recently that C++11 “feels like a new language — the pieces just fit together better.” Indeed, core C++11 has changed significantly. It now supports lambda expressions, automatic type deduction of objects, uniform initialization syntax, delegating constructors, deleted and defaulted function declarations,`nullptr`, and most importantly, rvalue references — a feature that augurs a paradigm shift in how one conceives and handles objects. And that’s just a sample.

The C++11 Standard Library was also revamped with new algorithms, new container classes, atomic operations, type traits, regular expressions, new smart pointers, `async()` facility, and of course a multithreading library.

A complete list of the new core and library features of C++11 is available here.

After the approval of the C++ standard in 1998, two committee members prophesied that the next C++ standard would “certainly” include a built-in garbage collector (GC), and that it probably wouldn’t support multithreading because of the technical complexities involved in defining a portable threading model. Thirteen years later, the

new C++ standard, C++11, is almost complete. Guess what? It lacks a GC but it does include a state-of–the-art threading library.

In this article I explain the biggest changes in the language, and why they are such a big deal. As you'll see, threading libraries are not the only change. The new standard builds on the decades of expertise and makes C++ even more relevant. As Rogers Cadenhead points out, "That's pretty amazing for something as old as disco, Pet Rocks, and Olympic swimmers with chest hair."

First, let's look at some of the prominent C++11 core-language features.

LAMBDA EXPRESSIONS

A lambda expression lets you define functions locally, at the place of the call, thereby eliminating much of the tedium and security risks that function objects incur. A lambda expression has the form:

[capture](parameters)->return-type {body}

The `[]` construct inside a function call's argument list indicates the beginning of a lambda expression. Let's see a lambda example.

Suppose you want to count how many uppercase letters a string contains.
Using `for_each()` to traverses a char array, the following lambda expression determines whether each letter is in uppercase. For every uppercase letter it finds, the lambda expression increments `Uppercase`, a variable defined outside the lambda expression:

```
int main()
{
    char s[]="Hello World!";
    int Uppercase = 0; //modified by the lambda
    for_each(s, s+sizeof(s), [&Uppercase] (char c) {
     if (isupper(c))
       Uppercase++;
```

```
    });
 cout<< Uppercase<<" uppercase letters in: "<< s<<endl;
}
```

It's as if you defined a function whose body is placed inside another function call. The ampersand in [&UPPERCASE] means that the lambda body gets a reference to UPPERCASE so it can modify it. Without the ampersand,UPPERCASE would be passed by value. C++11 lambdas include constructs for member functions as well.

AUTOMATIC TYPE DEDUCTION AND DECLTYPE

In C++03, you must specify the type of an object when you declare it. Yet in many cases, an object's declaration includes an initializer. C++11 takes advantage of this, letting you declare objects without specifying their types:

```
auto x=0; //x has type int because 0 is int
auto c='a'; //char
auto d=0.5; //double
auto national_debt=14400000000000LL;//long long
```

Automatic type deduction is chiefly useful when the type of the object is verbose or when it's automatically generated (in templates). Consider:

```
void func(const vector<int> &vi)
{
vector<int>::const_iterator ci=vi.begin();
}
```

Instead, you can declare the iterator like this:

```
auto ci=vi.begin();
```

The keyword `auto` isn't new; it actually dates back the pre-ANSI C era. However, C++11 has changed its meaning; `auto`no longer designates an object with automatic storage type. Rather, it declares an object whose type is deducible from its initializer. The old meaning of `auto` was removed from C++11 to avoid confusion.

C++11 offers a similar mechanism for capturing the type of an object or an expression. The new operator `decltype` takes an expression and “returns” its type:

```
const vector<int> vi;
typedef decltype (vi.begin()) CIT;
CIT another_const_iterator;
```

UNIFORM INITIALIZATION SYNTAX

C++ has at least four different initialization notations, some of which overlap.

Parenthesized initialization looks like this:

```
std::string s("hello");
int m=int(); //default initialization
```

You can also use the `=` notation for the same purpose in certain cases:

```
std::string s="hello";
int x=5;
```

For POD aggregates, you use braces:

```
int arr[4]={0,1,2,3};
struct tm today={0};
```

Finally, constructors use member initializers:

```
struct S {
 int x;
 S(): x(0) {} };
```

This proliferation is a fertile source for confusion, not only among novices. Worse yet, in C++03 you can’t initialize POD array members and POD arrays allocated using `new[]`. C++11 cleans up this mess with a uniform brace notation:

```
class C
{
int a;
```

```
int b;
public:
 C(int i, int j);
};

C c {0,0}; //C++11 only. Equivalent to: C c(0,0);

int* a = new int[3] { 1, 2, 0 }; /C++11 only

class X {
   int a[4];
public:
   X() : a{1,2,3,4} {} //C++11, member array initializer
};
```

With respect to containers, you can say goodbye to a long list of`push_back()` calls. In C++11 you can initialize containers intuitively:

```
// C++11 container initializer
vector<string> vs={ "first", "second", "third"};
map singers =
   { {"Lady Gaga", "+1 (212) 555-7890"},
     {"Beyonce Knowles", "+1 (212) 555-0987"}};
```

Similarly, C++11 supports in-class initialization of data members:

```
class C
{
 int a=7; //C++11 only
public:
 C();
};
```

DELETED AND DEFAULTED FUNCTIONS

A function in the form:

```
struct A
{
 A()=default; //C++11
 virtual ~A()=default; //C++11
};
```

is called a DEFAULTED FUNCTION. The `=default;` part instructs the compiler to generate the default implementation for the function. Defaulted functions have two advantages: They are more efficient than manual implementations, and they rid the programmer from the chore of defining those functions manually.

The opposite of a defaulted function is a DELETED FUNCTION:

```
int func()=delete;
```

Deleted functions are useful for preventing object copying, among the rest. Recall that C++ automatically declares a copy constructor and an assignment operator for classes. To disable copying, declare these two special member functions `=delete`:

```
struct NoCopy
{
 NoCopy & operator =( const NoCopy & ) = delete;
 NoCopy ( const NoCopy & ) = delete;
};
NoCopy a;
NoCopy b(a); //compilation error, copy ctor is deleted
```

NULLPTR

At last, C++ has a keyword that designates a null pointer constant. `nullptr`replaces the bug-prone `NULL` macro and the literal 0 that have been used as null pointer substitutes for many years. `nullptr` is strongly-typed:

```
void f(int); //#1
void f(char *);//#2
//C++03
f(0); //which f is called?
```

```
//C++11
f(nullptr) //unambiguous, calls #2
```

`nullptr` is applicable to all pointer categories, including function pointers and pointers to members:

```
const char *pc=str.c_str(); //data pointers
if (pc!=nullptr)
   cout<<pc<<endl;
int (A::*pmf)()=nullptr; //pointer to member function
void (*pmf)()=nullptr; //pointer to function
```

DELEGATING CONSTRUCTORS

In C++11 a constructor may call another constructor of the same class:

```
class M //C++11 delegating constructors
{
 int x, y;
 char *p;
public:
 M(int v) : x(v), y(0), p(new char [MAX]) {} //#1 target
 M(): M(0) {cout<<"delegating ctor"<<endl;} //#2 delegating
};
```

Constructor #2, the delegating constructor, invokes the TARGET CONSTRUCTOR#1.

RVALUE REFERENCES

Reference types in C++03 can only bind to lvalues. C++11 introduces a new category of reference types called RVALUE REFERENCES. Rvalue references can bind to rvalues, e.g. temporary objects and literals.

The primary reason for adding rvalue references is MOVE SEMANTICS. Unlike traditional copying, moving means that a target object PILFERS the resources of the source object, leaving the source in an “empty” state. In certain cases where making a copy of an object is both expensive and unnecessary, a move operation can be used

instead. To appreciate the performance gains of move semantics, consider string swapping. A naive implementation would look like this:

```
void naiveswap(string &a, string & b)
{
 string temp = a;
 a=b;
 b=temp;
}
```

This is expensive. Copying a string entails the allocation of raw memory and copying the characters from the source to the target. In contrast, moving strings merely swaps two data members, without allocating memory, copying char arrays and deleting memory:

```
void moveswapstr(string& empty, string & filled)
{
//pseudo code, but you get the idea
 size_t sz=empty.size();
 const char *p= empty.data();
//move filled's resources to empty
 empty.setsize(filled.size());
 empty.setdata(filled.data());
//filled becomes empty
 filled.setsize(sz);
 filled.setdata(p);
}
```

If you’re implementing a class that supports moving, you can declare a move constructor and a move assignment operator like this:

```
class Movable
{
Movable (Movable&&); //move constructor
Movable&& operator=(Movable&&); //move assignment operator
};
```

The C++11 Standard Library uses move semantics extensively. Many algorithms and containers are now move-optimized.

C++11 STANDARD LIBRARY

C++ underwent a major facelift in 2003 in the form of the Library Technical Report 1 (TR1). TR1 included new container classes (`unordered_set`,`unordered_map`, `unordered_multiset`, and `unordered_multimap`) and several new libraries for regular expressions, tuples, function object wrapper and more. With the approval of C++11, TR1 is officially incorporated into standard C++ standard, along with new libraries that have been added since TR1. Here are some of the C++11 Standard Library features:

THREADING LIBRARY

Unquestionably, the most important addition to C++11 from a programmer's perspective is concurrency. C++11 has a thread class that represents an execution thread, promises and futures, which are objects that are used for synchronization in a concurrent environment, the async() function template for launching concurrent tasks, and the thread_local storage type for declaring thread-unique data. For a quick tour of the C++11 threading library, read Anthony Williams' Simpler Multithreading in C++0x.

NEW SMART POINTER CLASSES

C++98 defined only one smart pointer class, `auto_ptr`, which is now deprecated. C++11 includes new smart pointer classes: shared_ptr and the recently-added unique_ptr. Both are compatible with other Standard Library components, so you can safely store these smart pointers in standard containers and manipulate them with standard algorithms.

NEW ALGORITHMS

The C++11 Standard Library defines new algorithms that mimic the set theory operations `all_of()`, `any_of()` and `none_of()`. The following listing applies the predicate `ispositive()` to the range `[first, first+n)` and uses `all_of()`, `any_of()` and `none_of()` to examine the range's properties:

```
#include <algorithm>
//C++11 code
```

```
//are all of the elements positive?
all_of(first, first+n, ispositive()); //false
//is there at least one positive element?
any_of(first, first+n, ispositive());//true
// are none of the elements positive?
none_of(first, first+n, ispositive()); //false
```

A new category of `copy_n` algorithms is also available. Using `copy_n()`, copying an array of 5 elements to another array is a cinch:

```
#include
int source[5]={0,12,34,50,80};
int target[5];
//copy 5 elements from source to target
copy_n(source,5,target);
```

The algorithm `iota()` creates a range of sequentially increasing values, as if by assigning an initial value to `*first`, then incrementing that value using prefix ++. In the following listing, `iota()` assigns the consecutive values {10,11,12,13,14} to the array `arr`, and {'a', 'b', 'c'} to the char array `c`.

```
include <numeric>
int a[5]={0};
char c[3]={0};
iota(a, a+5, 10); //changes a to {10,11,12,13,14}
iota(c, c+3, 'a'); //{'a','b','c'}
```

C++11 still lacks a few useful libraries such as an XML API, sockets, GUI, reflection — and yes, a proper automated garbage collector. However, it does offer plenty of new features that will make C++ more secure, efficient (yes, even more efficient than it has been thus far! See Google's benchmark tests), and easier to learn and use.

If the changes in C++11 seem overwhelming, don't be alarmed. Take the time to digest these changes gradually. At the end of this process you will probably agree with Stroustrup: C++11 DOES feel like a new language — a much better one.

" http://blog.smartbear.com/c-plus-plus/the-biggest-changes-in-c11-and-why-you-should-care/

TERMINOLOGY

RVALUES AND LVALUES

“C++11’s most pervasive feature is probably move semantics, and the foundation of move semantics is distinguishing expressions that are rvalues from those that are lvalues. That’s because rvalues indicate objects eligible for move operations, while lvalues generally don’t. In concept (though not always in practice), rvalues correspond to temporary objects returned from functions, while lvalues correspond to objects you can refer to, either by name or by following a pointer or lvalue reference.

A useful heuristic to determine whether an expression is an lvalue is to ask if you can take its address. If you can, it typically is. If you can’t, it’s usually an rvalue. A nice feature of this heuristic is that it helps you remember that the type of an expression is independent of whether the expression is an lvalue or an rvalue.” (Meyers, Effective Modern C++, 2014)

“Every C++ expression is either an lvalue or an rvalue. An lvalue refers to an object that persists beyond a single expression. You can think of an lvalue as an object that has a name. All variables, including nonmodifiable (**const**) variables, are lvalues. An rvalue is a temporary value that does not persist beyond the expression that uses it. To better understand the difference between lvalues and rvalues, consider the following example:

```
#include <iostream>

using namespace std;

int main()

{

  int x = 3 + 4;

  cout << x << endl;

}
```

In this example, x is an lvalue because it persists beyond the expression that defines it. The expression 3 + 4 is an rvalue because it evaluates to a temporary value that does not persist beyond the expression that defines it.

” (Microsoft, 2014)

“An *lvalue* (*locator value*) represents an object that occupies some identifiable location in memory (i.e. has an address).

rvalues are defined by exclusion, by saying that every expression is either an *lvalue* or an *rvalue*. Therefore, from the above definition of *lvalue*, an *rvalue* is an expression that *does not* represent an object occupying some identifiable location in memory.

Basic examples

The terms as defined above may appear vague, which is why it's important to see some simple examples right away.

Let's assume we have an integer variable defined and assigned to:

```
int var;
var = 4;
```

An assignment expects an lvalue as its left operand, and `var` is an lvalue, because it is an object with an identifiable memory location. On the other hand, the following are invalid:

```
4 = var;       // ERROR!
(var + 1) = 4; // ERROR!
```

Neither the constant `4`, nor the expression `var + 1` are lvalues (which makes them rvalues). They're not lvalues because both are temporary results of expressions, which don't have an identifiable memory location (i.e. they can just reside in some temporary register for the duration of the computation). Therefore, assigning to them makes no semantic sense - there's nowhere to assign to.

So it should now be clear what the error message in the first code snippet means. `foo` returns a temporary value which is an rvalue. Attempting to assign to it is an error, so when seeing `foo() = 2;` the compiler complains that it expected to see an lvalue on the left-hand-side of the assignment statement.

Not all assignments to results of function calls are invalid, however. For example, C++ references make this possible:

```
int globalvar = 20;

int& foo()
{
    return globalvar;
}

int main()
{
    foo() = 10;
    return 0;
```

```
}
```

Here `foo` returns a reference, WHICH IS AN LVALUE, so it can be assigned to. Actually, the ability of C++ to return lvalues from functions is important for implementing some overloaded operators. One common example is overloading the brackets operator `[]` in classes that implement some kind of lookup access. `std::map` does this:

```
std::map<int, float> mymap;
mymap[10] = 5.6;
```

The assignment `mymap[10]` works because the non-const overload of `std::map::operator[]` returns a reference that can be assigned to.

Conversions between lvalues and rvalues

Generally speaking, language constructs operating on object values require rvalues as arguments. For example, the binary addition operator `'+'` takes two rvalues as arguments and returns an rvalue:

```
int a = 1;     // a is an lvalue
int b = 2;     // b is an lvalue
int c = a + b; // + needs rvalues, so a and b are converted to rvalues
               // and an rvalue is returned
```

As we've seen earlier, `a` and `b` are both lvalues. Therefore, in the third line, they undergo an implicit LVALUE-TO-RVALUE CONVERSION. All lvalues that aren't arrays, functions or of incomplete types can be converted thus to rvalues.

What about the other direction? Can rvalues be converted to lvalues? Of course not! This would violate the very nature of an lvalue according to its definition [1].

This doesn't mean that lvalues can't be produced from rvalues by more explicit means. For example, the unary `'*'`(dereference) operator takes an rvalue argument but produces an lvalue as a result. Consider this valid code:

```
int arr[] = {1, 2};
int* p = &arr[0];
*(p + 1) = 10;   // OK: p + 1 is an rvalue, but *(p + 1) is an lvalue
```

Conversely, the unary address-of operator `'&'` takes an lvalue argument and produces an rvalue:

```
int var = 10;
int* bad_addr = &(var + 1); // ERROR: lvalue required as unary '&' operand
int* addr = &var;           // OK: var is an lvalue
&var = 40;                  // ERROR: lvalue required as left operand
                            // of assignment
```

The ampersand plays another role in C++ - it allows to define reference types. These are called "lvalue references". Non-const lvalue references cannot be assigned rvalues, since that would require an invalid rvalue-to-lvalue conversion:

```
std::string& sref = std::string();  // ERROR: invalid initialization of
                                    // non-const reference of type
                                    // 'std::string&' from an rvalue of
                                    // type 'std::string'
```

Constant lvalue references CAN be assigned rvalues. Since they're constant, the value can't be modified through the reference and hence there's no problem of modifying an rvalue. This makes possible the very common C++ idiom of accepting values by constant references into functions, which avoids unnecessary copying and construction of temporary objects.

CV-qualified rvalues

If we read carefully the portion of the C++ standard discussing lvalue-to-rvalue conversions [2], we notice it says:

> An lvalue (3.10) of a non-function, non-array type T can be converted to an rvalue. [...] If T is a non-class type, the type of the rvalue is the cv-unqualified version of T. Otherwise, the type of the rvalue is T.

What is this "cv-unqualified" thing? CV-QUALIFIER is a term used to describe CONST and VOLATILE type qualifiers.

From section 3.9.3:

> Each type which is a cv-unqualified complete or incomplete object type or is void (3.9) has three corresponding cv-qualified versions of its type: a CONST-QUALIFIED version, a VOLATILE-QUALIFIED version, and a CONST-VOLATILE-QUALIFIEDversion. [...] The cv-qualified or cv-

unqualified versions of a type are distinct types; however, they shall have the same representation and alignment requirements (3.9)

But what has this got to do with rvalues? Well, in C, rvalues never have cv-qualified types. Only lvalues do. In C++, on the other hand, class rvalues can have cv-qualified types, but built-in types (like `int`) can't. Consider this example:

```
#include <iostream>

class A {
public:
    void foo() const { std::cout << "A::foo() const\n"; }
    void foo() { std::cout << "A::foo()\n"; }
};

A bar() { return A(); }
const A cbar() { return A(); }



int main()
{
    bar().foo();  // calls foo
    cbar().foo(); // calls foo const
}
```

The second call in `main` actually calls the `foo () const` method of `A`, because the type returned by `cbar` is `const A`, which is distinct from `A`. This is exactly what's meant by the last sentence in the quote mentioned earlier. Note also that the return value from `cbar` is an rvalue. So this is an example of a cv-qualified rvalue in action." (Bendersky, 2014)

"C++ has always produced fast programs. Unfortunately, until C++11, there has been an obstinate wart that slows down many C++ programs: the creation of temporary objects. Sometimes these temporary objects can be optimized away by the compiler (the return value optimization, for example). But this is not always the case, and it can result in expensive object copies. What do I mean?

Let's say that you have the following code:

```
1   #include <iostream>
2
3   using namespace std;
4
5   vector<int> doubleValues (const vector<int>& v)
6   {
7       vector<int> new_values;
8       new_values.reserve();
9       for (auto itr = v.begin(), end_itr = v.end(); itr != end_itr; ++itr )
10      {
11          new_values.push_back( 2 * *itr );
12      }
13      return new_values;
14  }
15
16  int main()
17  {
18      vector<int> v;
19      for ( int i = 0; i < 100; i++ )
20      {
21          v.push_back( i );
22      }
23      v = doubleValues( v );
24  }
```

If you've done a lot of high performance work in C++, sorry about the pain that brought on. If you haven't--well, let's walk through why this code is terrible C++03 code. (The rest of this tutorial will be about why it's fine C++11 code.) The problem is with the copies. When doubleValues is called, it constructs a vector, new_values, and fills it up. This alone might not be ideal performance, but if we

want to keep our original vector unsullied, we need a second copy. But what happens when we hit the return statement?

The entire contents of new_values must be copied! In principle, there could be up to two copies here: one into a temporary object to be returned, and a second when the vector assignment operator runs on the line v = doubleValues(v);. The first copy may be optimized away by the compiler automatically, but there is no avoiding that the assignment to v will have to copy all the values again, which requires a new memory allocation and another iteration over the entire vector.

This example might be a little bit contrived--and of course you can find ways to avoid this kind of problem--for example, by storing and returning the vector by pointer, or by passing in a vector to be filled up. The thing is, neither of these programming styles is particularly natural. Moreover, an approach that requires returning a pointer has introduced at least one more memory allocation, and one of the design goals of C++ is to avoid memory allocations.

The worst part of this whole story is that the object returned from doubleValues is a temporary value that's no longer needed. When you have the line v = doubleValues(v), the result of doubleValues(v) is just going to get thrown away once it is copied! In theory, it should be possible to skip the whole copy and just pilfer the pointer inside the temporary vector and keep it in v. In effect, why can't we **move** the object? In C++03, the answer is that there was no way to tell if an object was a temporary or not, you had to run the same code in the assignment operator or copy constructor, no matter where the value came from, so no pilfering was possible. In C++11, the answer is--you can!

That's what rvalue references and move semantics are for! Move semantics allows you to avoid unnecessary copies when working with temporary objects that are about to evaporate, and whose resources can safely be taken from that temporary object and used by another.

Move semantics relies on a new feature of C++11, called rvalue references, which you'll want to understand to really appreciate what's going on. So first let's talk about what an rvalue is, and then what an rvalue reference is. Finally, we'll come back to move semantics and how it can be implemented with rvalue references.

RVALUES AND LVALUES - BITTER RIVALS, OR BEST OF FRIENDS?

In C++, there are rvalues and lvalues. An lvalue is an expression whose address can be taken, a locator value--essentially, an lvalue provides a (semi)permanent piece of memory. You can make assignments to lvalues. For example:

```
1  int a;
2  a = 1; // here, a is an lvalue
```

You can also have lvalues that aren't variables:

```
1  int x;
2  int& getRef ()
3  {
4          return x;
```

```
5   }
6
7   getRef() = 4;
```

Here, getRef returns a reference to a global variable, so it's returning a value that is stored in a permanent location. (You could literally write & getRef() if you wanted to, and it would give you the address of x.)

Rvalues are--well, rvalues are not lvalues. An expression is an rvalue if it results in a temporary object. For example:

```
1   int x;
2   int getVal ()
3   {
4       return x;
5   }
6   getVal();
```

Here, getVal() is an rvalue--the value being returned is not a reference to x, it's just a temporary value. This gets a little bit more interesting if we use real objects instead of numbers:

```
1   string getName ()
2   {
3       return "Alex";
4   }
5   getName();
```

Here, getName returns a string that is constructed inside the function. You can assign the result of getName to a variable:

```
1   string name = getName();
```

But you're assigning from a temporary object, not from some value that has a fixed location. getName() is an rvalue.

DETECTING TEMPORARY OBJECTS WITH RVALUE REFERENCES

The important thing is that rvalues refer to temporary objects--just like the value returned from doubleValues. Wouldn't it be great if we could know, without a shadow of a doubt, that a value returned from an expression was temporary, and somehow write code that is overloaded to behave differently for temporary objects? Why, yes, yes indeed it would be. And this is what rvalue references are for. An rvalue reference is a reference that will bind only to a temporary object. What do I mean?

Prior to C++11, if you had a temporary object, you could use a "regular" or "lvalue reference" to bind it, but only if it was const:

```
1  const string& name = getName(); // ok
2  string& name = getName(); // NOT ok
```

The intuition here is that you cannot use a "mutable" reference because, if you did, you'd be able to modify some object that is about to disappear, and that would be dangerous. Notice, by the way, that holding on to a const reference to a temporary object ensures that the temporary object isn't immediately destructed. This is a nice guarantee of C++, but it is still a temporary object, so you don't want to modify it.

In C++11, however, there's a new kind of reference, an "rvalue reference", that will let you bind a mutable reference to an rvalue, but not an lvalue. In other words, rvalue references are perfect for detecting if a value is temporary object or not. Rvalue references use the && syntax instead of just &, and can be const and non-const, just like lvalue references, although you'll rarely see a const rvalue reference (as we'll see, mutable references are kind of the point):

```
1  const string&& name = getName(); // ok
2  string&& name = getName(); // also ok - praise be!
```

So far this is all well and good, but how does it help? The most important thing about lvalue references vs rvalue references is what happens when you write functions that take lvalue or rvalue references as arguments. Let's say we have two functions:

```
1  printReference (const String& str)
2  {
3          cout << str;
4  }
5
6  printReference (String&& str)
7  {
8          cout << str;
9  }
```

Now the behavior gets interesting--the printReference function taking a const lvalue reference will accept any argument that it's given, whether it be an lvalue or an rvalue, and regardless of whether the lvalue or rvalue is mutable or not. However, in the presence of the second overload, printReference taking an rvalue reference, it will be given all values EXCEPT mutable rvalue-references. In other words, if you write:

```
1  string me( "alex" );
2  printReference(  me ); // calls the first printReference function, taking an lvalue ref
3
4  printReference( getName() ); // calls the second printReference function, taking a muta
```

Now we have a way to determine if a reference variable refers to a temporary object or to a permanent object. The rvalue reference version of the method is like the secret back door entrance to the club that you can only get into if you're a temporary object (boring club, I guess). Now that we have our method of determining if an object was a temporary or a permanent thing, how can we use it?

MOVE CONSTRUCTOR AND MOVE ASSIGNMENT OPERATOR

The most common pattern you'll see when working with rvalue references is to create a move constructor and move assignment operator (which follows the same principles). A move constructor, like a copy constructor, takes an instance of an object as its argument and creates a new instance based on the original object. However, the move constructor can avoid memory reallocation because we know it has been provided a temporary object, so rather than copy the fields of the object, we will move them.

What does it mean to move a field of the object? If the field is a primitive type, like int, we just copy it. It gets more interesting if the field is a pointer: here, rather than allocate and initialize new memory, we can simply steal the pointer and null out the pointer in the temporary object! We know the temporary object will no longer be needed, so we can take its pointer out from under it.

Imagine that we have a simple ArrayWrapper class, like this:

```
1   class ArrayWrapper
2   {
3       public:
4           ArrayWrapper (int n)
5               : _p_vals( new int[ n ] )
6               , _size( n )
7           {}
8           // copy constructor
```

```
9           ArrayWrapper (const ArrayWrapper& other)
10              : _p_vals( new int[ other._size  ] )
11              , _size( other._size )
12          {
13              for ( int i = 0; i < _size; ++i )
14              {
15                  _p_vals[ i ] = other._p_vals[ i ];
16              }
17          }
18          ~ArrayWrapper ()
19          {
20              delete [] _p_vals;
21          }
22      private:
23      int *_p_vals;
24      int _size;
25  };
```

Notice that the copy constructor has to both allocate memory and copy every value from the array, one at a time! That's a lot of work for a copy. Let's add a move constructor and gain some massive efficiency.

```
1   class ArrayWrapper
2   {
3   public:
4       // default constructor produces a moderately sized array
5       ArrayWrapper ()
6           : _p_vals( new int[ 64 ] )
7           , _size( 64 )
8       {}
```

```
9
10      ArrayWrapper (int n)
11          : _p_vals( new int[ n ] )
12          , _size( n )
13      {}
14
15      // move constructor
16      ArrayWrapper (ArrayWrapper&& other)
17          : _p_vals( other._p_vals  )
18          , _size( other._size )
19      {
20          other._p_vals = NULL;
21          other._size = 0;
22      }
23
24      // copy constructor
25      ArrayWrapper (const ArrayWrapper& other)
26          : _p_vals( new int[ other._size  ] )
27          , _size( other._size )
28      {
29          for ( int i = 0; i < _size; ++i )
30          {
31              _p_vals[ i ] = other._p_vals[ i ];
32          }
33      }
34      ~ArrayWrapper ()
35      {
            delete [] _p_vals;
```

```
36        }
37
38    private:
39        int *_p_vals;
40        int _size;
41    };
42
```

Wow, the move constructor is actually simpler than the copy constructor! That's quite a feat. The main things to notice are:

1. The parameter is a non-const rvalue reference
2. other._p_vals is set to NULL

The second observation explains the first--we couldn't set other._p_vals to NULL if we'd taken a const rvalue reference. But why do we need to set other._p_vals = NULL? The reason is the destructor--when the temporary object goes out of scope, just like all other C++ objects, its destructor will run. When its destructor runs, it will free _p_vals. The same _p_vals that we just copied! If we don't set other._p_vals to NULL, the move would not really be a move--it would just be a copy that introduces a crash later on once we start using freed memory. This is the whole point of a move constructor: to avoid a copy by changing the original, temporary object!

Again, the overload rules work such that the move constructor is called only for a temporary object--and only a temporary object that can be modified. One thing this means is that if you have a function that returns a const object, it will cause the copy constructor to run instead of the move constructor--so don't write code like this:

```
1    const ArrayWrapper getArrayWrapper (); // makes the move constructor useless, the tempo
```

There's still one more situation we haven't discussed how to handle in a move constructor--when we have a field that is an object. For example, imagine that instead of having a size field, we had a metadata field that looked like this:

```
1    class MetaData
2    {
3    public:
4        MetaData (int size, const std::string& name)
5            : _name( name )
6            , _size( size )
```

```
7         {}
8
9         // copy constructor
10        MetaData (const MetaData& other)
11            : _name( other._name )
12            , _size( other._size )
13        {}
14
15        // move constructor
16        MetaData (MetaData&& other)
17            : _name( other._name )
18            , _size( other._size )
19        {}
20
21        std::string getName () const { return _name; }
22        int getSize () const { return _size; }
23        private:
24        std::string _name;
25        int _size;
26    };
```

Now our array can have a name and a size, so we might have to change the definition of ArrayWrapper like so:

```
1     class ArrayWrapper
2     {
3     public:
4         // default constructor produces a moderately sized array
5         ArrayWrapper ()
```

```
6           : _p_vals( new int[ 64 ] )
7           , _metadata( 64, "ArrayWrapper" )
8       {}
9
10      ArrayWrapper (int n)
11          : _p_vals( new int[ n ] )
12          , _metadata( n, "ArrayWrapper" )
13      {}
14
15      // move constructor
16      ArrayWrapper (ArrayWrapper&& other)
17          : _p_vals( other._p_vals  )
18          , _metadata( other._metadata )
19      {
20          other._p_vals = NULL;
21      }
22
23      // copy constructor
24      ArrayWrapper (const ArrayWrapper& other)
25          : _p_vals( new int[ other._metadata.getSize() ] )
26          , _metadata( other._metadata )
27      {
28          for ( int i = 0; i < _metadata.getSize(); ++i )
29          {
30              _p_vals[ i ] = other._p_vals[ i ];
31          }
32      }
        ~ArrayWrapper ()
```

```
33      {
34          delete[] _p_vals;
35      }
36  private:
37      int*_p_vals;
38      MetaData _metadata;
39  };
40
```

Does this work? It seems very natural, doesn't it, to just call the MetaData move constructor from within the move constructor for ArrayWrapper? The problem is that this just doesn't work. The reason is simple: the value of other in the move constructor--it's an rvalue reference. But an rvalue reference is not, in fact, an rvalue. It's an lvalue, and so the copy constructor is called, not the move constructor. This is weird. I know--it's confusing. Here's the way to think about it. A rvalue is an expression that creates an object that is about to evaporate into thin air. It's on its last legs in life--or about to fulfill its life purpose. Suddenly we pass the temporary to a move constructor, and it takes on new life in the new scope. In the context where the rvalue expression was evaluated, the temporary object really is over and done with. But in our constructor, the object has a name; it will be alive for the entire duration of our function. In other words, we might use the variable other more than once in the function, and the temporary object has a defined location that truly persists for the entire function. It's an lvalue in the true sense of the term locator value, we can locate the object at a particular address that is stable for the entire duration of the function call. We might, in fact, want to use it later in the function. If a move constructor were called whenever we held an object in an rvalue reference, we might use a moved object, by accident!

```
1  // move constructor
2  ArrayWrapper (ArrayWrapper&& other)
3      : _p_vals( other._p_vals  )
4      , _metadata( other._metadata )
5  {
6      // if _metadata( other._metadata ) calls the move constructor, using
7      // other._metadata here would be extremely dangerous!
8      other._p_vals = NULL;
9  }
```

Put a final way: both lvalue and rvalue references are lvalue expressions. The difference is that an lvalue reference must be const to hold a reference to an rvalue, whereas an rvalue reference can always hold a reference to an rvalue. It's like the difference between a pointer, and what is pointed to.

The thing pointed-to came from an rvalue, but when we use rvalue reference itself, it results in an lvalue.

STD::MOVE

So what's the trick to handling this case? We need to use std::move, from <utility>--std::move is a way of saying, "ok, honest to God I know I have an lvalue, but I want it to be an rvalue." std::move does not, in and of itself, move anything; it just turns an lvalue into an rvalue, so that you can invoke the move constructor. Our code should look like this:

```
1  #include <utilityh> // for std::move
2
3      // move constructor
4      ArrayWrapper (ArrayWrapper&& other)
5          : _p_vals( other._p_vals  )
6          , _metadata( std::move( other._metadata ) )
7      {
8          other._p_vals = NULL;
9      }
```

And of course we should really go back to MetaData and fix its own move constructor so that it uses std::move on the string it holds:

```
1  MetaData (MetaData&& other)
2      : _name( std::move( other._name ) ) // oh, blissful efficiency
3      : _size( other._size )
4  {}
```

MOVE ASSIGNMENT OPERATOR

Just as we have a move constructor, we should also have a move assignment operator. You can easily write one using the same techniques as for creating a move constructor.

MOVE CONSTRUCTORS AND IMPLICITLY GENERATED CONSTRUCTORS

As you know, in C++ when you declare any constructor, the compiler will no longer generate the default constructor for you. The same is true here: adding a move constructor to a class will require you to declare and define your own default constructor. On the other hand, declaring a move constructor does not prevent the compiler from providing an implicitly generated copy constructor,

and declaring a move assignment operator does not inhibit the creation of a standard assignment operator.

HOW DOES STD::MOVE WORK

You might be wondering, how does one write a function like std::move? How do you get this magical property of transforming an lvalue into an rvalue reference? The answer, as you might guess, istypecasting. The actual declaration for std::move is somewhat more involved, but at its heart, it's just a static_cast to an rvalue reference. This means, actually, that you don't really NEED to use move--but you should, since it's much more clear what you mean. The fact that a cast is required is, by the way, a very good thing! It means that you cannot accidentally convert an lvalue into an rvalue, which would be dangerous since it might allow an accidental move to take place. You must explicitly use std::move (or a cast) to convert an lvalue into an rvalue reference, and an rvalue reference will never bind to an lvalue on its own.

RETURNING AN EXPLICIT RVALUE-REFERENCE FROM A FUNCTION

Are there ever times where you should write a function that returns an rvalue reference? What does it mean to return an rvalue reference anyway? Aren't functions that return objects by value already rvalues?

Let's answer the second question first: returning an explicit rvalue reference is different than returning an object by value. Take the following simple example:

```
1   int x;
2
3   int getInt ()
4   {
5       return x;
6   }
7
8   int && getRvalueInt ()
9   {
10      // notice that it's fine to move a primitive type--remember, std::move is just a c
11      return std::move( x );
12  }
```

Clearly in the first case, despite the fact that getInt() is an rvalue, there is a copy of the variable x being made. We can even see this by writing a little helper function:

```
1  void printAddress (const int& v) // const ref to allow binding to rvalues
2  {
3      cout << reinterpret_cast<const void*>( & v ) << endl;
4  }
5
6  printAddress( getInt() );
7  printAddress( x );
```

When you run this program, you'll see that there are two separate values printed.

On the other hand,

```
1  printAddress( getRvalueInt() );
2  printAddress( x );
```

prints the same value because we are explicitly returning an rvalue here.

So returning an rvalue reference is a different thing than not returning an rvalue reference, but this difference manifests itself most noticeably if you have a pre-existing object you are returning instead of a temporary object created in the function (where the compiler is likely to eliminate the copy for you).

Now on to the question of whether you want to do this. The answer is: probably not. In most cases, it just makes it more likely that you'll end up with a dangling reference (a case where the reference exists, but the temporary object that it refers to has been destroyed). The issue is quite similar to the danger of returning an lvalue reference--the referred-to object may no longer exist. Rvalue references cannot magically keep an object alive for you. Returning an rvalue reference would primarily make sense in very rare cases where you have a member function and need to return the result of calling std::move on a field of the class from that function--and how often are you going to do that?

MOVE SEMANTICS AND THE STANDARD LIBRARY

Going back to our original example--we were using a vector, and we don't have control over the vector class and whether or not it has a move constructor or move assignment operator. Fortunately, the standards committee is wise, and move semantics has been added to the standard library. This means that you can now efficiently return vectors, maps, strings and whatever other standard library objects you want, taking full advantage of move semantics.

MOVEABLE OBJECTS IN STL CONTAINERS

In fact, the standard library goes one step further. If you enable move semantics in your own objects by creating move assignment operators and move constructors, when you store those objects in a

container, the STL will automatically use std::move, automatically taking advantage of move-enabled classes to eliminate inefficient copies.

” http://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html

“Lvalues and rvalues were introduced in a seminal article by Strachey et al (1963) that presented CPL. A CPL expression appearing on the left hand side of an assignment expression is evaluated as a memory address into which the right-hand side value is written. Later, left-hand expressions and right-hand expressions became lvalues and rvalues, respectively.

One of CPL’s descendants, B, was the language on which Dennis Ritchie based C. Ritchie borrowed the term lvalue to refer to a memory region to which a C program can write the right hand side value of an assignment expression. He left out rvalues, feeling that lvalue and “not lvalue” would suffice.

Later, rvalue made it into K&R C and ISO C++. C++11 extended the notion of rvalues even further by letting you bind rvalue references to them. Although nowadays lvalue and rvalues have slightly different meanings from their original CPL meanings, they are encoded “in the genes of C++,” to quote Bjarne Stroustrup. Therefore, understanding what they mean and how the addition of move semantics affected them can help you understand certain C++ features and idioms better — and write better code.

RIGHT FROM WRONG

Before attempting to define lvalues, let’s look at some examples:

```
int x=9;
std::string s;
int *p=0;
int &ri=x;
```

The identifiers x, s, p and ri are all lvalues. Indeed, they can appear on the left-hand side of an assignment expression and therefore seem to justify the CPL generalization: “Anything that can appear on the left-hand side of an assignment expression is an lvalue.” However, counter-examples are readily available:

```
void func(const int * pi, const int & ri) {
*pi=7;//compilation error, *pi is const
ri=8; //compilation error, ri is const
}
```

*pi and ri are const lvalues. Therefore, they cannot appear on the left-hand side of an expression after their initialization. This property doesn't make them rvalues, though.

Now let's look at some examples of rvalues. Literals such as 7, 'a', false and"hello world!" are instances of rvalues:

```
7==x;
char c= 'a';
bool clear=false;
const char s[]="hello world!";
```

Another subcategory of rvalues is *temporaries*. During the evaluation of an expression, an implementation may create a temporary object that stores an intermediary result:

```
int func(int y, int z, int w){
int x=0;
x=(y*z)+w;
return x ;
}
```

In this case, an implementation may create a temporary int to store the result of the sub-expression y*z. Conceptually, a temporary *expires* once its expression is fully evaluated. Put differently, it goes out of scope or gets destroyed upon reaching the nearest semicolon.

You can create temporaries explicitly, too. An expression in the formC(arguments) creates a temporary object of type C:

```
cout<<std::string ("test").size()<<endl;
```

Contrary to the CPL generalization, rvalues *may* appear on the left-hand side of an assignment expression in certain cases:

```
string ()={"hello"}; //creates a temp string
```

You're probably more familiar with the shorter form of this idiom:

```
string("hello"); //creates a temp string
```

Clearly, the CPL generalization doesn't really cut it for C++, although intuitively, it does capture the semantic difference between lvalues andrvalues.

So, what do lvalues and rvalues really stand for?

A MATTER OF IDENTITY

An *expression* is a program statement that yields a value, for example a function call, a sizeof expression, an arithmetic expression, a logical expression, and so on. You can classify C++ expressions into two categories: values with identity and values with no identity. In this context,identity means a name, a pointer, or a reference that enable you to determine if two objects are the same, to change the state of an object, or copy it:

```
struct {int x; int y;} s; //no type name, value has id
string &rs= *new string;
const char *p= rs.data();
```

s, rs and p are identities of values. We can draw the following generalization: lvalues in C++03 are values that have identity. By contrast,rvalues in C++03 are values that have no identity. C++03 rvalues are accessible only inside the expression in which they are created:

```
int& func();
int func2();
func(); //this call is an lvalue
func2(); //this call is an rvalue
sizeof(short); //rvalue
new double; //new expressions are rvalues
S::S() {this->x=0; /*this is an rvalue expression*/}
```

A function's name (not to be confused with a function call) is an rvalueexpression that evaluates to the function's address. Similarly, an array's name is an rvalue expression that evaluates to the address of the first element of the array:

```
int& func3();
int& (*pf)()=func3;//func3 is an rvalue
int arr[2];
int* pi=arr;//arr is an rvalue
```

Because rvalues are short-lived, you have to capture them in lvalues if you wish to access them outside the context of their expression:

```
std::size_t n=sizeof(short);
double *pd=new double;
struct S
{
int x, y;
S() { S *p=this; p->x=0; p->y=0;}
};
```

Remember that any expression that evaluates to an lvalue reference (e.g., a function call, an overloaded assignment operator, etc.) is an lvalue. Any expression that returns an object by value is an rvalue.

Prior to C++11, identity (or the lack thereof) were the main criterion for distinguishing between lvalues and rvalues. However, the addition of rvaluereferences and move semantics to C++11 added a twist to the plot.

BINDING RVALUES

C++11 lets you bind rvalue references to rvalues, effectively prolonging their lifetime as if they were lvalues:

```
//C++11
int && func2(){
return 17; //returns an rvalue
```

```
}
int main() {
int x=0;
int&& rr=func2();
cout<<rr<<endl;//ouput: 17
x=rr;// x=17 after the assignment
}
```

Using lvalue references where rvalue references are required is an error:

```
int& func2(){//compilation error: cannot bind
return 17;      //an lvalue reference to an rvalue
}
```

In C++03 copying the rvalue to an lvalue is the preferred choice (in some cases you can bind an lvalue reference to const to achieve a similar effect):

```
int func2(){ // an rvalue expression
return 17;
}
int m=func2(); // C++03-style copying
```

For fundamental types, the copy approach is reasonable. However, as far as class objects are concerned, spurious copying might incur performance overhead. Instead, C++11 encourages you to *move* objects. Moving means*pilfering* the resources of the source object, instead of copying it.

For further information about move semantics, read [C++11 Tutorial: Introducing the Move Constructor and the Move Assignment Operator].

```
//C++11 move semantics in action
string source ("abc"), target;
target=std::move(source);  //pilfer source
//source no longer owns the resource
cout<<"source: "<<source<<endl; //source:
cout<<"target: "<<target<<endl; //target: abc
```

How does move semantics affect the semantics of lvalues and rvalues?

THE SEMANTICS OF CHANGE

In C++03, all you needed to know was whether a value had identity. In C++11 you also have to examine another property: movability. The combination of identity and movability (i and m, respectively, with a minus sign indicating negation) produces five meaningful value categories in C++11— "a whole type-zoo," as one of my Twitter followers put it:

- **i-m: lvalues** are non-movable objects with identity. These are classic C++03 lvalues from the pre-move era. The expression *p, where p is a pointer to an object is an lvalue. Similarly, dereferencing a pointer to a function is an lvalue.
- **im: xvalues** (an "eXpiring" value) refers to an object near the end of its lifetime (before its resources are moved, for example). An xvalue is the result of certain kinds of expressions involving rvalue references, e.g.,std::move(mystr);
- **i: glvalues**, or generalized lvalues, are values with identity. These include lvalues and xvalues.
- **m: rvalues** include xvalues, temporaries, and values that have no identity.
- **-im: prvalues,** or *pure* RVALUES, are rvalues that are not xvalues.Prvalues include literals and function calls whose return type is not a reference.

A detailed discussion about the new value categories is available in section 3.10 of the C++11 standard.

It has often been said that the original semantics of C++03 lvalues and rvalues remains unchanged in C++11. However, the C++11 taxonomy isn't quite the same as that of C++03; In C++11, every expression belongs to exactly one of the value classifications lvalue, xvalue, or prvalue.

IN CONCLUSION

Fifty years after their inception, lvalues and rvalues are still relevant not only in C++ but in many contemporary programming languages. C++11 changed the semantics of rvalues, introducing xvalues and prvalues. Conceptually, you can tame this type-zoo by grouping the five value categories into supersets, where glvalues include lvalues and xvalues, and rvalues includexvalues and prvalues.

Still confused? It's not you. It's BCPL's heritage that exhibits unusual vitality in a world that's light years away from the punched card and 16 kilobyte RAM era.

" http://blog.smartbear.com/development/c11-tutorial-explaining-the-ever-elusive-lvalues-and-rvalues/

FUNCTION CALLS, ARGUMENTS AND PARAMETERS

In a function call, the expressions passed at the call site 1 are the function's *arguments*. The arguments are used to initialize the function's *parameters*. In the first call to someFunc above, the argument is wid. In the second call, the argument is std::move(wid). In both calls, the parameter is w. The distinction between arguments and parameters is important, because parameters are lvalues, but the arguments with which they are initialized may be rvalues or lvalues. This is especially relevant during the process of *perfect forwarding*, whereby an argument passed to a function is passed to a second function such that the original argument's rvalueness or lvalueness is preserved.

EXCEPTION-SAFE

"Well-designed functions are exception-safe, meaning they offer at least the basic exception safety guarantee (i.e., the basic guarantee). Such functions assure callers that even if an exception is thrown, program invariants remain intact (i.e., no data structures are corrupted) and no resources are leaked. Functions offering the strong exception safety guarantee (i.e., the strong guarantee) assure callers that if an exception is arises, the state of the program remains as it was prior to the call. " (Meyers, Effective Modern C++, 2014)

FUNCTION OBJECT

"When I refer to a function object, I usually mean an object of a type supporting an operator() member function. In other words, an object that acts like a function. Occasionally I use the term in a slightly more general sense to mean anything that can be invoked using the syntax of a non-member function call (i.e., "function- Name(arguments)"). This broader definition covers not just objects supporting operator(), but also functions and C-like function pointers. (The narrower definition comes from C++98, the broader one from C++11.) Generalizing further by adding member function pointers yields what are known as callable objects. You can generally ignore the fine distinctions and simply think of function objects and callable objects as things in C++ that can be invoked using some kind of function calling syntax." (Meyers, Effective Modern C++, 2014)

LAMBDA EXPRESSION, CLOSURES AND TEMPLATES

"Function objects created through lambda expressions are known as closures. It's seldom necessary to distinguish between lambda expressions and the closures they create, so I often refer to both as lambdas. Similarly, I rarely distinguish between function templates (i.e., templates that generate functions) and template functions (1 i.e., the functions generated from function templates). Ditto for class templates and template classes. " (Meyers, Effective Modern C++, 2014)

DECLARATIONS, DEFINITIONS AND SIGNATURES

“Many things in C++ can be both declared and defined. Declarations introduce names and types without giving details, such as where storage is located or how things are implemented:

```
extern int x; // object declaration

class Widget; // class declaration

bool func(const Widget& w); // function declaration

enum class Color; // scoped enum declaration
```

Definitions provide the storage locations or implementation details:

```
int x; // object definition

class Widget { // class definition

…

};

bool func(const Widget& w)

{ return w.size() < 10; } // function definition

enum class Color

{ Yellow, Red, Blue }; // scoped enum definition
```

A definition also qualifies as a declaration, so unless it’s really important that something is a definition, I tend to refer to declarations.

I define a function’s signature to be the part of its declaration that specifies parameter and return types. Function and parameter names are not part of the signature. In the example above, func’s signature is bool(const Widget&). Elements of a function’s declaration other than its parameter and return types (e.g., noexcept or constexpr, if present), are excluded.

The official definition of “signature” is slightly different from mine, but for this book, my definition is more useful. (The official definition sometimes omits return types.)” (Meyers, Effective Modern C++, 2014)

POINTERS

“I call built-in pointers, such as those returned from new, raw pointers. The opposite of a raw pointer is a smart pointer. Smart pointers normally overload the pointer-deferencing operators (operator-> and operator*), though std::weak_ptr is an exception. In source code comments, I sometimes abbreviate “constructor” as ctor and “destructor” as dtor.” (Meyers, Effective Modern C++, 2014),

OFFSET

“In computer science, an **offset** within an array or other data structure object is an integer indicating the distance (displacement) from the beginning of the object up until a given element or point, presumably within

the same object. The concept of a distance is valid only if all elements of the object are the same size (typically given in bytes or words).

For example, given an array of characters *A*, containing "abcdef", the element containing the character 'c' has an offset of two from the start of *A*." http://en.wikipedia.org/wiki/Offset_(computer_science)

STRONG AND WEAK TYPING

"In computer programming, programming languages are often colloquially referred to as **strongly typed** or **weakly typed**. In general, these terms do not have a precise definition, but in general a strongly typed language is more likely to generate an error or refuse to compile a program if the argument passed to a function does not closely match the expected type. A very weakly typed language may produce unpredictable results or may perform implicit type conversion instead. On the other hand, a very strongly typed language may not work at all or crash when data of an unexpected type is passed to a function.[1]"
http://en.wikipedia.org/wiki/Strong_and_weak_typing

DEDUCTING TYPES

- **Understand template type deduction**
  - During template type deduction, arguments that are references are treated as non-references, i.e., their reference-ness is ignored.
  - When deducing types for universal reference parameters, lvalue arguments get special treatment.
  - When deducing types for by-value parameters, const and /or volatile arguments are treated as non-const and non-volatile.
  - During template type deductions, arguments that are array or function names decay to pointers, unless they're used to initialize references.
- **Understand auto type deduction**
  - auto type deduction is usually the same as template type deduction, but auto type deduction assumes that a braced initializer represents a std:initializer_list, and template type deduction doesn't.
  - auto in a function return type or lambda parameter implies template type deduction, not auto type deduction.
- **Understand delctype**
  - Decltype almost always yields the type of a variable or expression without any modifications.
  - For lvalue expressions of type T other than names, decltype always reports a type for T&.
  - C++ 14 supports decltype(auto), which like auto, deduces a type from its initializer, but it performs the type deduction using decltype rules.
- **Know how to view deduced types**
  - Deduced types can often be seen using IDE editors, compiler error messages, and the Boost TypeIndex library.
  - The results of some tools may be neither helpful nor accurate, so an under standing of C++'s type deduction rules remains essential.

This chapter provides the information about type deduction that every C++ developer requires. It explains how template type deduction works, how `auto` builds on that, and how `decltype` goes its own way. It even explains how you can force compilers to make the results of their type deductions visible, thus enabling you to ensure that compilers are deducing the types you want them to.

REMOVE THE IMAGE FROM BOOK around page 12

TEMPLATE TYPE DEDUCTIONS

"Type deduction for templates is the basis for one of modern C++'s most compelling features: **auto**. to truly understand the aspects of template type deduction that auto builds on.

It's natural to expect that the type deduced for T is the same as the type of the argument passed to the function, i.e., that T is the type of expr. In the above example, that's the case: x is an int, and T is deduced to be int. But it doesn't always work that way. The type deduced for T is dependent not just on the type of expr, but also on the form of ParamType.

```
template<typename T>
void f(ParamType param);
f(expr); // deduce T and ParamType from expr
```

There are three cases:

- **ParamType** is a pointer or reference type, but not a universal 1 reference. (Universal references is that they exist and that they're not the same as lvalue references or rvalue references.)
    - Type deduction works like this:
        - If expr's type is a reference, ignore the reference part.
        - Then pattern-match expr's type against ParamType to determine T.
- **ParamType** is a universal reference.
    - In particular, when universal references are in use, type deduction distinguishes between lvalue arguments and rvalue arguments. That never happens for non universal references.
- **ParamType** is neither a pointer nor a reference.
    - pass-by-value

" (Meyers, Effective Modern C++, 2014)

AUTO TYPE DEDUCTION

Auto type decuction is a template type deduction.

AUTO

- **Prefer auto to explicit type declarations**

    - auto variables must be initialized, are generally immune to type mismatches that can lead to portability or efficiency problems, can ease the process of refactoring , and typically require less typing than variables with explicitly specified types.
    - auto-typed variables are subject to the pitfalls of auto type deductions and "invisible" proxy classes
- **Use the explicitly typed initializer idiom when auto deduces undesired types**
    - "Invisible" proxy types can cause auto to deduce the "wrong" type for an initializing expression.
    - The explicitly typed initializer idiom forces auto to deduce the type you want it to have.

C++11 TUTORIAL: LET YOUR COMPILER DETECT THE TYPES OF YOUR OBJECTS AUTOMATICALLY

"Imagine that your compiler could guess the type of the variables you declare as if by magic. In C++11, it's possible — well, almost. The new auto and decltype facilities detect the type of an object automatically, thereby paving the way for cleaner and more intuitive function declaration syntax, while ridding you of unnecessary verbiage and keystrokes. Find out how auto and decltype simplify the design of generic code, improve code readability, and reduce maintenance overhead.

Fact #1: Most declarations of C++ variables include an explicit initializer. Fact #2: The majority of those initializers encode a unique datatype. C++11 took advantage of these facts by introducing two new closely related keywords: `auto` and `decltype`.`auto` lets you

declare objects without specifying their types explicitly, while`decltype` captures the type of an object.

Together, `auto` and `decltype` make the design of generic code simpler and more robust while reducing the number of unnecessary keystrokes. Let's look closely at these new C++11 silver bullets.

AUTO DECLARATIONS

As stated above, most of the declarations in a C++ program have an explicit initializer (on a related note, you may find this article about class member initializers useful). Consider the following examples:

```
int x=0;

const double PI=3.14;

string curiosity_greeting ("hello universe");

bool dirty=true;

char *p=new char [1024];

long long distance=10000000000LL;

int (*pf)() = func;
```

Each of the seven initializers above is associated with a unique datatype. Let's examine some of them:

- The literal 0 is int.
- Decimal digits with a decimal point (e.g. 3.14) are taken to be double.
- The initializer new char [] implies char*.
- An integral literal value with the affix LL (or ll) is long long.

The compiler can extract this information and automatically assign types to the objects you're declaring. For that purpose, C++11 introduced a new syntactic mechanism. A declaration whose type is omitted begins with the new keyword auto. An AUTO DECLARATION must include an initializer with a distinct type (an empty pair of parentheses won't do). The rest is relegated to the compiler. How convenient!

NOTE: C++ ACTUALLY HAS TWO AUTO KEYWORDS: THE OLD AUTO AND THE NEW AUTO. THE OLD AUTO DATES BACK TO THE LATE 1960S, SERVING AS A STORAGE CLASS IN DECLARATIONS OF VARIABLES WITH AUTOMATIC STORAGE. SINCE AUTOMATIC STORAGE IS ALWAYS IMPLIED FROM THE CONTEXT IN C++, THE OLD AUTO WAS SELDOM USED, WHICH IS WHY MOST PROGRAMMERS DIDN'T EVEN KNOW IT EXISTED IN C++ UNTIL RECENTLY.*C++11 removed the old auto (though it still exists in C) in favor of the new* `auto`*, which is used in* `auto`*declarations. Some implementations may require that you turn on C++11 computability mode to support the new* `auto`*. In this article, I refer exclusively to the new* `auto`*.*

Here are the previous seven declarations transformed into auto declarations:

```
//C++11

auto x=0;

const auto PI=3.14;

auto curiosity_greeting= string ("hello universe");

auto dirty=true;
```

```
auto p=new char [1024];

auto distance=10000000000LL;

auto pf = func;
```

In the case of class objects such as `std::string`, you need to include the name of the class in the initializer to disambiguate the code. Otherwise, the quoted literal string "hello universe" would be construed as `const char[15]`.

CONST, ARRAYS, AND REFERENCE VARIABLES

If you want to declare cv-qualified objects, you have to provide the const and volatile qualifiers in the auto declaration itself. That's because the compiler can't tell whether the initializer 3.14 is plain double, const double, volatile double, or const volatile double:

```
volatile auto i=5; //volatile int

auto volatile const flag=true; //const volatile bool

auto const y='a'; //const char
```

To declare an auto array, enclose the initializers in braces. The compiler counts the number of initializers enclosed and uses that information to determine the size (and dimensions) of the array:

```
auto iarr={0,1,2,3}; //int [4]

auto arrarr={{0,1},{0,1}}; //int[2][2]
```

TIP: CERTAIN IMPLEMENTATIONS INSIST THAT YOU *#include* THE NEW STANDARD HEADER *<initializer_list>* WHEN YOU DECLARE AN AUTO ARRAY. TECHNICALLY, THE C++11 STANDARD REQUIRES THAT AN IMPLEMENTATION *#include <initializer_list>* IMPLICITLY WHENEVER THAT

HEADER IS NEEDED, SO YOU'RE NOT SUPPOSED TO WRITE ANY *#include <initializer_list>* DIRECTIVES, EVER. HOWEVER, NOT ALL IMPLEMENTATIONS ARE FULLY COMPLIANT IN THIS RESPECT. THEREFORE, IF YOU ENCOUNTER COMPILER ERRORS IN ARRAY AUTO DECLARATIONS, TRY TO *#include <initializer_list>*MANUALLY.

Declaring an auto array of pointers to functions (and pointers to member functions) follows the same principles:

```
int func1();

int func2();

auto pf_arr={func1, func2};//array  int (*[2])()



int (A::*pmf) (int)=&A::f; //pointer to member

auto pmf_arr={pmf,pmf}; //array int (A::*[2]) (int)
```

Reference variables are a slightly different case. Unlike with pointers, when the initializer is a reference type, auto still defaults to value semantics:

```
int& f();

auto x=f(); //x is int, not int&
```

To declare a reference variable using auto, add the & to the declaration explicitly:

```
auto& xref=f(); //xref is int&
```

Technically, you can declare multiple entities in a single auto expression:

```
auto x=5, d=0.54, flag=false, arr={1,2,3}, pf=func;
```

OPERATOR DECLTYPE

The new operator decltype captures the type of an expression. You can use decltype to store the type of an object, expression, or literal value. In this sense, decltype is the complementary operation of auto – whereas auto instructs the compiler to fill-in the omitted type, decltype "uncovers" the type of an object:

```
auto x=0;  //int

decltype (x)  z=x; //same as auto z=x;

typedef decltype (x) XTYPE; //XTYPE is a synonym for int

XTYPE y=5;
```

How does it work? The expression `decltype(x)` extracts the type of `x`, which is int. That type is assigned to the new typedef name XTYPE. XTYPE is then used in the declaration of a new variable, `y`, which has the same type as `x`.

Let's look at a more realistic example in which decltype extracts the iterator type from a generic expression. Suppose you want to capture the iterator type returned from a `vector<T>::begin()` call. You can do it like this:

```
vector<Foo> vf;

typedef decltype(vf.begin()) ITER;

for (ITER it=vf.begin(); it<vf.end(); it++)

  cout<<*it<<endl;
```

Notice that decltype takes an expression as its argument. Therefore, you can apply decltype not just to objects and function calls, but also to arithmetic expressions, literal values, and array objects:

```
typedef decltype(12) INT;  //a literal value

float f;

typedef decltype(f) FLOAT; //a variable

typedef decltype (std::vector<int>()) VI; //a temporary

typedef decltype(4.9*0.357) DOUBLE; //a math expression

auto arr={2,4,8};

typedef decltype(arr) int_array;//an array

int_array table;

table[0]=-10;
```

C++11 STYLE FUNCTION DECLARATIONS

C++11 introduced a new notation for declaring a function using auto and decltype. Recall that a traditional function declaration looks like this:

```
return_type name(params,...); //old style function declaration
```

In the new function declaration notation, the return type appears after the closing parenthesis of the parameter list and is preceded by a `->` sign. For example:

```
auto newfunc(bool b)->decltype(b)//C++11 style function declaration
```

```
{ return b;}
```

`newfunc()` takes a parameter of type bool and returns bool. This style is useful in template functions where the return type depends on the TEMPLATE-ID (the actual template instance). Thus, template functions can define a generic return type using the new notation and decltype. The actual return type is calculated automatically, depending on the type of the decltype expression:

```
//return type is vector<T>::iterator
template <class T> auto get_end (vector<T>& v) ->decltype(v.end());

{return v.end(); }

vector<Foo> vf;

get_end(vf); //returns vector<Foo>::iterator

const vector<Foo> cvf;

get_end(cvf); //returns vector<Foo>::const_iterator
```

The new function declaration notation overcomes parsing ambiguity problems that were associated with the traditional function declaration notation. Additionally, this notation is easier to maintain because the actual return type isn't hardcoded in the source file. As a bonus, it is said to be more readable.

IN CONCLUSION

auto and decltype fill in a gap that, until recently, forced programmers to use hacks and non-standard extensions, such as the GCC-specific typeof() operator. auto doesn't just eliminate unnecessary keystrokes. Rather, it also simplifies complex declarations and improves the design of generic code. decltype complements auto by letting you capture the type of complex expressions, literal values, and function calls without spelling out (or

even knowing) their type. Together with the new function declaration notation, auto and decltype make a C++11 programmer's job a tad easier.

" http://blog.smartbear.com/c-plus-plus/c11-tutorial-let-your-compiler-detect-the-types-of-your-objects-automatically/

C++11 TUTORIAL: LET YOUR COMPILER DETECT THE TYPES OF YOUR OBJECTS AUTOMATICALLY

"Imagine that your compiler could guess the type of the variables you declare as if by magic. In C++11, it's possible — well, almost. The new auto and decltype facilities detect the type of an object automatically, thereby paving the way for cleaner and more intuitive function declaration syntax, while ridding you of unnecessary verbiage and keystrokes. Find out how auto and decltype simplify the design of generic code, improve code readability, and reduce maintenance overhead.

Fact #1: Most declarations of C++ variables include an explicit initializer. Fact #2: The majority of those initializers encode a unique datatype. C++11 took advantage of these facts by introducing two new closely related keywords: `auto` and `decltype`.`auto` lets you declare objects without specifying their types explicitly, while`decltype` captures the type of an object.

Together, `auto` and `decltype` make the design of generic code simpler and more robust while reducing the number of unnecessary keystrokes. Let's look closely at these new C++11 silver bullets.

AUTO DECLARATIONS

As stated above, most of the declarations in a C++ program have an explicit initializer (on a related note, you may find this article about class member initializers useful). Consider the following examples:

```
int x=0;

const double PI=3.14;

string curiosity_greeting ("hello universe");

bool dirty=true;

char *p=new char [1024];

long long distance=10000000000LL;

int (*pf)() = func;
```

Each of the seven initializers above is associated with a unique datatype. Let's examine some of them:

- The literal 0 is int.
- Decimal digits with a decimal point (e.g. 3.14) are taken to be double.
- The initializer new char [] implies char*.
- An integral literal value with the affix LL (or ll) is long long.

The compiler can extract this information and automatically assign types to the objects you're declaring. For that purpose, C++11 introduced a new syntactic mechanism. A declaration whose type is omitted begins with the new keyword auto. An AUTO DECLARATION must include an initializer with a distinct type (an empty pair of parentheses won't do). The rest is relegated to the compiler. How convenient!

NOTE: C++ ACTUALLY HAS TWO AUTO KEYWORDS: THE OLD AUTO AND THE NEW AUTO. THE OLD AUTO DATES BACK TO THE LATE 1960S, SERVING AS A STORAGE CLASS IN DECLARATIONS OF VARIABLES WITH AUTOMATIC STORAGE. SINCE AUTOMATIC STORAGE IS ALWAYS IMPLIED FROM THE CONTEXT IN C++, THE OLD AUTO WAS SELDOM USED, WHICH IS WHY MOST PROGRAMMERS DIDN'T EVEN KNOW IT EXISTED IN C++ UNTIL RECENTLY.*C++11 removed the old auto (though it still exists in C) in favor of the new* `auto`*, which is used in* `auto`*declarations. Some implementations may require that you turn on C++11 computability mode to support the new* `auto`*. In this article, I refer exclusively to the new* `auto`*.*

Here are the previous seven declarations transformed into auto declarations:

```
//C++11

auto x=0;

const auto PI=3.14;

auto curiosity_greeting= string ("hello universe");

auto dirty=true;

auto p=new char [1024];

auto distance=10000000000LL;
```

```
auto pf = func;
```

In the case of class objects such as `std::string`, you need to include the name of the class in the initializer to disambiguate the code. Otherwise, the quoted literal string "hello universe" would be construed as `const char[15]`.

CONST, ARRAYS, AND REFERENCE VARIABLES

If you want to declare cv-qualified objects, you have to provide the const and volatile qualifiers in the auto declaration itself. That's because the compiler can't tell whether the initializer 3.14 is plain double, const double, volatile double, or const volatile double:

```
volatile auto i=5; //volatile int

auto volatile const flag=true; //const volatile bool

auto const y='a'; //const char
```

To declare an auto array, enclose the initializers in braces. The compiler counts the number of initializers enclosed and uses that information to determine the size (and dimensions) of the array:

```
auto iarr={0,1,2,3}; //int [4]

auto arrarr={{0,1},{0,1}}; //int[2][2]
```

TIP: CERTAIN IMPLEMENTATIONS INSIST THAT YOU *#include* THE NEW STANDARD HEADER *<initializer_list>* WHEN YOU DECLARE AN AUTO ARRAY. TECHNICALLY, THE C++11 STANDARD REQUIRES THAT AN IMPLEMENTATION *#include <initializer_list>* IMPLICITLY WHENEVER THAT HEADER IS NEEDED, SO YOU'RE NOT SUPPOSED TO WRITE ANY *#include <initializer_list>* DIRECTIVES, EVER. HOWEVER, NOT ALL IMPLEMENTATIONS ARE FULLY COMPLIANT IN THIS RESPECT. THEREFORE,

IF YOU ENCOUNTER COMPILER ERRORS IN ARRAY AUTO DECLARATIONS, TRY TO *#include <initializer_list>* MANUALLY.

Declaring an auto array of pointers to functions (and pointers to member functions) follows the same principles:

```
int func1();

int func2();

auto pf_arr={func1, func2};//array  int (*[2])()



int (A::*pmf) (int)=&A::f; //pointer to member

auto pmf_arr={pmf,pmf}; //array int (A::*[2]) (int)
```

Reference variables are a slightly different case. Unlike with pointers, when the initializer is a reference type, auto still defaults to value semantics:

```
int& f();

auto x=f(); //x is int, not int&
```

To declare a reference variable using auto, add the & to the declaration explicitly:

```
auto& xref=f(); //xref is int&
```

Technically, you can declare multiple entities in a single auto expression:

```
auto x=5, d=0.54, flag=false, arr={1,2,3}, pf=func;
```

OPERATOR DECLTYPE

The new operator decltype captures the type of an expression. You can use decltype to store the type of an object, expression, or literal value. In this sense, decltype is the complementary operation of auto – whereas auto instructs the compiler to fill-in the omitted type, decltype "uncovers" the type of an object:

```
auto x=0;  //int

decltype (x)  z=x; //same as auto z=x;

typedef decltype (x) XTYPE; //XTYPE is a synonym for int

XTYPE y=5;
```

How does it work? The expression `decltype(x)` extracts the type of `x`, which is int. That type is assigned to the new typedef name XTYPE. XTYPE is then used in the declaration of a new variable, `y`, which has the same type as `x`.

Let's look at a more realistic example in which decltype extracts the iterator type from a generic expression. Suppose you want to capture the iterator type returned from a `vector<T>::begin()` call. You can do it like this:

```
vector<Foo> vf;

typedef decltype(vf.begin()) ITER;

for (ITER it=vf.begin(); it<vf.end(); it++)

  cout<<*it<<endl;
```

Notice that decltype takes an expression as its argument. Therefore, you can apply decltype not just to objects and function calls, but also to arithmetic expressions, literal values, and array objects:

```
typedef decltype(12) INT;  //a literal value
```

```
float f;

typedef decltype(f) FLOAT; //a variable

typedef decltype (std::vector<int>()) VI; //a temporary

typedef decltype(4.9*0.357) DOUBLE; //a math expression

auto arr={2,4,8};

typedef decltype(arr) int_array;//an array

int_array table;

table[0]=-10;
```

C++11 STYLE FUNCTION DECLARATIONS

C++11 introduced a new notation for declaring a function using auto and decltype. Recall that a traditional function declaration looks like this:

```
return_type name(params,...); //old style function declaration
```

In the new function declaration notation, the return type appears after the closing parenthesis of the parameter list and is preceded by a `->` sign. For example:

```
auto newfunc(bool b)->decltype(b)//C++11 style function declaration

{ return b;}
```

`newfunc()` takes a parameter of type bool and returns bool. This style is useful in template functions where the return type depends on the TEMPLATE-ID (the actual template instance). Thus, template functions can define a generic return type using the

new notation and decltype. The actual return type is calculated automatically, depending on the type of the decltype expression:

```
//return type is vector<T>::iterator
template <class T> auto get_end (vector<T>& v) ->decltype(v.end());

{return v.end(); }

vector<Foo> vf;

get_end(vf); //returns vector<Foo>::iterator

const vector<Foo> cvf;

get_end(cvf); //returns vector<Foo>::const_iterator
```

The new function declaration notation overcomes parsing ambiguity problems that were associated with the traditional function declaration notation. Additionally, this notation is easier to maintain because the actual return type isn't hardcoded in the source file. As a bonus, it is said to be more readable.

IN CONCLUSION

auto and decltype fill in a gap that, until recently, forced programmers to use hacks and non-standard extensions, such as the GCC-specific typeof() operator. auto doesn't just eliminate unnecessary keystrokes. Rather, it also simplifies complex declarations and improves the design of generic code. decltype complements auto by letting you capture the type of complex expressions, literal values, and function calls without spelling out (or even knowing) their type. Together with the new function declaration notation, auto and decltype make a C++11 programmer's job a tad easier.

" http://blog.smartbear.com/c-plus-plus/c11-tutorial-let-your-compiler-detect-the-types-of-your-objects-automatically/

MODERN C++ FUNCTIONALITIES

- **Distinguish between () and {} when creating objects**
  - Braced initialization is the most widely usable initialization syntax, it prevents narrowing conversions, and it's immune to C++'s most vexing parse.
  - During constructor overload resolution, braced initializers are matched to `std::initializer_list` parameters if at all possible, even if other constructors offer seemingly better matches.
  - An example of where the choice between parentheses and braces can make a significant difference is creating a `std::vector<`*numeric type*`>` with two arguments.
  - Choosing between parentheses and braces for object creation inside templates can be challenging.
- **Prefer nullptr to 0 and NULL**
- Avoid overloading on integral and pointer types
- **Prefer alias declarations to typedefs**
  - typedefs do not support templatization, but **alias declarations** do
    - **using** UPtrMapSS = std::unique_ptr<std::unordered_map<std::string, std::string>>;
  - alias tempaltes avoid the "::type" suffix and, in templates, the "typename" prefix often required to refer to typedefs.
  - C++ 14 offers alias templates for all the C++11 type traits transformations.
- **Prefer scoped enums to unscoped enums**
  - C++ 98-style enums are now known as unscoped enums
  - Enumerators of scoped enums are visible only within the enum. They convert to other types only with a cast.
  - Both scoped and unscoped enums support specification of the underlying type. The default underlying type for scoped enums is int. Unscoped enums have no default underlying type.
  - Scoped enums may always be forward-declared. Unscoped enums may be forward declared only if their declaration specifies an underlying type.
- **Prefer *deleted* functions to private undefined ones.**
  - bool isLucky(int number); // original function
  - bool isLucky(char) **= delete**; // reject chars
  - bool isLucky(bool) **= delete**; // reject bools
  - bool isLucky(double) **= delete**; // reject doubles and Float
  - Any function may be deleted, including non-member functions and template instantiations.
- Declare overriding functions *override*
  - Member function reference qualifiers make it possible to treat lavalue and rvalue objects (*this) differently.
- Prefer const_iterators to iterators
  - In maximally generic code, prefer non-member versions of begin, end and rbegin etc., over their member function counterparts.
- Declare function noexcept if they won't emit exceptions
  - `noexcept` is part of a function's interface, and that means that callers may depend on it.
  - Noexcept function are more optimizable than non-except functions.
  - Noexcept is particularly valuable for the move operations, swap, memory and deallocation functions, and destructors-
  - Most functions are exception-neutral rather than noexcept.
- Use constexpr whenever possible

    - Constexpr objects are const and initialized with values known during compilation
    - Constexpr functions can produce compile-time results when called with arguments whose values are known during compilation.
    - Constexpr objects and functions may be used in a wider range of contexts than non-constexpr objects and functions.
    - Constexpr is part of an object's or function's interface.
- Make const member functions thread-safe
    - Make const member function thread safe unless you're certain they'll never be used in a concurrent context.
    - Use of std:atomic variables may offer better performance than a mutex, but they're suited for manipulation of only a single variable.
- Understand special member function generation
    - C++98 has four "special member functions": the default constructor, the destructor, the copy constructor and the copy assignment operator.
    - C++11 has two more options: the move constructor and the move assignment operator
    - The special member functions are those compilers may generate on their own: default constructor, destructor, copy operations, and move operations.
    - Move operations are generated only for classes lacking explicitly-declared move operations, copy operations, and a destructor.
    - The copy constructor is generated only for classes lacking an explicitly declared copy constructor, and it's deleted if a move operation is declared. The copy assignment operator is generated only for classes lacking an explicitly declared copy assignment operator, and it's deleted if a move operation is declared. Generation of the copy operations in classes with an explicitly-declared destructor is deprecated.
    - Member function templates never suppress generation of special member functions.

USE C++11 INHERITANCE CONTROL KEYWORDS TO PREVENT INCONSISTENCIES IN CLASS HIERARCHIES – OVERRIDE, FINAL

"FOR MORE THAN 30 YEARS, C++ GOT ALONG WITHOUT INHERITANCE CONTROL KEYWORDS. IT WASN'T EASY, TO SAY THE LEAST. DISABLING FURTHER DERIVATION OF A CLASS WAS POSSIBLE BUT TRICKY. TO PREVENT USERS FROM OVERRIDING A VIRTUAL FUNCTION IN A DERIVED CLASS YOU HAD TO LEAN OVER BACKWARDS. BUT NOT ANY MORE: TWO NEW CONTEXT-SENSITIVE KEYWORDS MAKE YOUR JOB A LOT EASIER. HERE'S HOW THEY WORK.

C++11 adds two inheritance control keywords: `override` and `final`.`override` ensures that an overriding virtual function declared in a derived class has the same signature as that of the base class. `final` blocks further derivation of a class and further overriding of a virtual function. Let's see how these watchdogs can eliminate design and implementation bugs in your class hierarchies.

VIRTUAL FUNCTIONS AND OVERRIDE

A derived class can override a member function that was declared virtual in a base class. This is a fundamental aspect of object-oriented design. However, things can go wrong even with such a trivial operation as overriding a function. Two common bugs related to overriding virtual functions are:

- Inadvertent overriding.
- Signature mismatch.

First, let's analyze the INADVERTENT OVERRIDING syndrome. You might inadvertently override a virtual function simply by declaring a member function that accidentally has the same name and signature as a base class's virtual member function. Compilers and human readers rarely detect this bug because they usually assume that the new function is meant to override the base class's function:

```
struct A
{
virtual void func();
};
struct B: A{};
struct F{};
struct D: A, F
{
  void func();//meant to declare a new function but
 //accidentally overrides A::func};
```

Reading the code listing above, you can't tell for sure whether the member function `D::func()` overrides `A::func()` deliberately. It could be an accidental overriding that occurred because the parameter lists and the names of both functions are identical by chance.

A SIGNATURE MISMATCH is a more commonplace scenario. It leads to the accidental creation of a new virtual function (instead of overriding an existing virtual function), as demonstrated in the following example:

```
struct G
{
 virtual void func(int);
};
struct H: G
{
 virtual void func(double); //accidentally creates a new virtual function
};
```

In this case, the programmer intended to override `G::func()` in class `H`. However, because `H::func()` has a different signature, the result is a new virtual function, not an override of a base class function. Not all compilers issue a warning in such cases, and those that do are sometimes configured to suppress this warning.

In C++11, you can eliminate these two bugs by using the new keyword`override`. `override` explicitly states that a function is meant to override a base class's virtual function. More importantly, it checks for signature mismatches between the base class virtual function and the overriding function in the derived classes. If the signatures don't match, the compiler issues an error message.

Let's see how `override` can eliminate the signature mismatch bug:

```
struct G
{
 virtual void func(int);
};
struct H: G
{
 virtual void func(double) override; //compilation error
};
```

When the compiler processes the declaration of `H::func()` it looks for a matching virtual function in a base class. Recall that "matching" in this context means:

- Identical function names.
- A `virtual` specifier in the first base class that declares the function.

- Identical parameter lists, return types (with one exception), cv qualifications etc., in both the base class's function and the derived class's overriding function.

If any of these three conditions isn't met, you get a compilation error. In our example, the parameter lists of the two functions don't match: `G::func()` takes `int` whereas `H::func()` takes `double`. Without the `override` keyword, the compiler would simply assume that the programmer meant to create a new virtual function in `H`.

Preventing the inadvertent overriding bug is trickier. In this case, it's the LACKof the keyword `override` that should raise your suspicion. If the derived class function is truly meant to override a base class function, it should include an explicit `override` specifier. Otherwise, assume that either `D::func()` is a new virtual function (a comment would be most appreciated in this case!), or that this may well be a bug.

FINAL FUNCTIONS AND CLASSES

The C++11 keyword `final` has two purposes. It prevents inheriting from classes, and it disables the overriding of a virtual function. Let's look at final classes first.

Certain classes that implement system services, infrastructure utilities, encryption etc., are often meant to be NON-SUBCLASSABLE: The implementers don't want clients to modify those classes by means of deriving new classes from them. Standard Library containers such as `std::vector` and `std::list` are another good example of non-subclassable types. These container classes don't have a virtual destructor or indeed, any virtual member functions.

And yet, every now and then, programmers insist on deriving from `std::vector` without realizing the risks involved. In C++11, non-subclassable types should be declared `final` like this:

```
class TaskManager final{/*..*/};
class PrioritizedTaskManager: public TaskManager {
};  //compilation error: base class TaskManager is final
```

In a similar vein, you can disable further overriding of a virtual function by declaring it `final`. If a derived class attempts to override a `final` function, the compiler issues an error message:

```
struct A
{
  virtual void func() const;
};
struct B: A
{
  void func() const override final; //OK
};
struct C: B
{
 void func()const; //error, B::func is final
};
```

It doesn't matter whether `C::func()` is declared `override`. Once a virtual function is declared `final`, derived classes cannot override it.

SYNTAX AND TERMINOLOGY

I have thus far avoided two side issues pertaining to `override` and `final`. The first one is their unique location. Unlike virtual, inline, explicit extern, and similar function specifiers, these two keywords appear after the closing parenthesis of a function's parameter list, or (in the case of non-subclassable classes) after the class name in a class declaration.

The peculiar location of these keywords is a consequence of another unusual property: `override` and `final` aren't ordinary keywords. In fact officially, they aren't keywords at all. C++11 considers them as identifiers that gain special meaning only when used in the specific contexts and locations as I have shown. In any other location or context, they are treated as identifiers. Consequently, the following listing makes perfectly valid C++11 code:

```
//valid C++11 code
int final=0;
bool override=false;
if (override==true){
 cout<<"override is: "<<override<<endl;}
struct D{} final;
struct A
{virtual bool func(); };
struct B:A
{ bool func() override final; };
```

It may seem surprising that `final` and `override` behave exactly like PL/1's context sensitive keywords (CSK). Since 1972, C and later C++ always avoided CSK, adhering instead to the reserved keywords approach.

So why did the committee make `final` and `override` an exception? The CSK choice was a compromise. Adding `override` and `final` as reserved keywords might have caused existing C++ code to break. If the committee had introduced new reserved keywords, they probably would have chosen funky strings such as `final_decl` or `_Override`, tokens that were less likely to clash with user-declared identifiers in legacy C++ code. However, no one likes such ugly keywords (ask C users what they think of C99's `_Bool` for example). That is why the CSK approach won eventually.

`override` and `final` become keywords in C++11, but only when used in specific contexts. Otherwise, they are treated as plain identifiers. The committee was reluctant to call `override` and `final` "context sensitive keywords" (which is what they truly are) though. Instead, they are formally referred to as "identifiers with special meaning." Special indeed!

IN CONCLUSION

The two new context-sensitive keywords override and final give you tighter control over hierarchies of classes, ridding you of some irritating inheritance-related bugs and design gaffes. `override` guarantees that an overriding virtual function matches its base class

counterpart. `final` blocks further derivation of a class or further overriding of a virtual function. With respect to compiler support, GCC 4.7, Intel's C++ 12, MSVC 11, and Clang 2.9 support these new keywords.

" http://blog.smartbear.com/c-plus-plus/use-c11-inheritance-control-keywords-to-prevent-inconsistencies-in-class-hierarchies/

INTERFACES DONE RIGHT

"Interfaces are one of the backbones of modern, object-oriented design. You need to use instances of different classes which share similarities in a uniform way? Use interfaces. You cannot easily test a class because it depends on details of other classes? Make your class depend on interfaces instead and introduce mock objects. You have a class with a single responsibility, but a respectable amount of methods which are intended for different types of callers? Make your class implement several interfaces, and let different callers refer to the ones which suit their needs.

As you can see, interfaces used correctly are pretty powerful tools, and many programming languages such as Java and C# provide dedicated interface keywords. This is not the case for C++. This article explains how to write clean and safe interfaces for your classes.

PLAYMATES FOR PROFESSIONAL MONSTER SLAYERS

Imagine you work in the computer games business and your team sets out to create the next big role-playing game. Marketing tells you it should be set in a cyber-punk science fiction medieval sorcery universe overrun with zombie unicorns to accomodate as many tastes as possible. Besides the D&D license you are craving for, the first thing you need is monsters.

Monsters share a few commonalities. They have names and they take damage. Usually they even fight back, but in the spirit of agile development we will only implement features for the VERY EASY difficulty mode in the first iteration. To express the commonalities of different kinds of monsters, we define the following monster interface:

```
1 // interface for all monsters
2 class monster {
3 public:
```

```
4     virtual ~monster();
5
6     // forbid copying
7     monster(monster const &) = delete;
8     monster & operator=(monster const &) = delete;
9
10    void receive_damage(double damage);
11    void interact_with_chainsaw();
12    std::string name() const;
13
14 protected:
15    // allow construction for child classes only
16    monster();
17
18 private:
19    virtual void do_receive_damage(double damage) = 0;
20    virtual void do_interact_with_chainsaw() = 0;
21    virtual std::string do_name() const = 0;
22 };
```

INTERFACE ANATOMY

Above snippet contains many details of note:

VIRTUAL DESCTRUCTOR

Since we intend monster to be implemented by child classes, we need to make the destructor virtual and declare it explicitly. If we forget this, child objects may not be cleaned up correctly if they are destroyed via pointers to the monster base class.

FORBID COPYING

When a derived class is copied by a reference to the base class, additional members of the child class would not be copied. This effect is known as slicing and must be avoided, and thus we forbid copying by deleting both the copy constructor and the copy assignment operator. In pre-C++11 code, you would declare these functions `private`and never implement them.

SIDE NOTE: If you really need copying, implement a `virtual clone()` method which returns a (smart) pointer to the base class.

PROTECTED CONSTRUCTOR

The constructor of `monster` has been made protected to highlight that this class can only be instantiated by child classes.

PUBLIC INTERFACE FOR END-USERS

A public interface is provided to all users of the class and its children. It consists of the methods `receive_damage()`, `name()`, and `interact_with_chainsaw()`. This is cyberpunk, after all. The public interface should be safe to use and very difficult to use incorrectly. Give as many guarantees as you can.

PURE PRIVATE INTERFACE FOR CHILD CLASS IMPLEMENTERS

Finally, pure `virtual` functions are provided which must be implemented by child classes. The purity is indicated by the `= 0` behind the prototypes. Please note that they are declared `private`. End users cannot call these functions directly, they can only use the `public` interface. Child class implementers, on the other hand, must only implement the `private` interface, i.e., the three methods `do_receive_damage()`,`do_interact_with_chainsaw()`, and `do_name()`.

All in all, above declaration makes it very clear for users and implementers alike what can be expected of this class.

INTERFACE IMPLEMENTATION

Let us have a look at the implementation of the `monster` interface:

```
1 monster::monster()
```

```
2  {
3  }
4
5  monster::~monster()
6  {
7  }
8
9  void monster::receive_damage(double damage)
10 {
11   do_receive_damage(damage);
12 }
13
14 void monster::interact_with_chainsaw()
15 {
16   do_interact_with_chainsaw();
17 }
18
19 std::string monster::name() const
20 {
21   return do_name();
22 }
```

Methods from the `public` interface call methods from the `private` interface. This layer of indirection, however, can be put to great use: The `receive_damage()` method could not only call `do_receive_damage()`, it could also perform additional tasks.

For example, it could check if the monster is still alive after being hit with a pointy stick. Furthermore, preconditions could be asserted, e.g., that the provided damage is a finite number in contrast to `NaN` or `Inf` (but you would use a type for that anyways, wouldn't you?). Once corresponding code is written in the base class, all child classes benefit from this check, no matter how sloppy their implementers are. In turn, this also relieves end users of

the monster interface from the burden of performing those checks themselves. They do not have to trust all potential child classes to come, they can trust your interface.
This separation of public end-user and private implementer interfaces is so useful that it is known as the NON-VIRTUAL INTERFACE (NVI) pattern and is widely used in modern C++ development.

SUBCLASSING THE INTERFACE

Let us now turn our attention to implementing a concrete monster class. The following snippet shows how the child class hobgoblin is declared:

```
1  class hobgoblin : public monster
2  {
3  public:
4      hobgoblin();
5      virtual ~hobgoblin();
6
7  private:
8      void do_receive_damage(double damage) final;
9      void do_interact_with_chainsaw() final;
10     std::string do_name() const final;
11
12     double health_;
13 };
```

Since this is intended as a concrete class, we provide a public constructor for users to call. We also declare a destructor to clean up members of hobgoblin. Though it is not required to repeat the virtual keyword, it highlights the destructor's nature.
In order to make hobgoblin concrete, i.e., instantiable, we need to implement (and declare, obviously) all pure virtual functions of the base class. Here we use two new C++11 features to protect us from past and future mistakes:

- override tells the compiler to complain if no virtual method with the exact same signature is present in any base class. This avoids accidently creating new methods with SLIGHTLY different signatures such as missing consts.
- final forbids child classes of hobgoblin to provide their own implementations of these virtual functions—a great feature for giving guarantees for this class and all its child classes. final implies override.
- One could add the virtual keyword at the start of the declaration. However, it does not provide a great deal of new information, since override guarantees that this function has been declared virtual in a base class.

For completeness, this is how hobgoblin would be implemented in a source file:

```
1  hobgoblin::hobgoblin() :
2      health_(100.0)
3  {
4  }
5
6  hobgoblin::~hobgoblin()
7  {
8  }
9
10 void hobgoblin::do_receive_damage(double damage)
11 {
12     health -= damage;
13 }
14
15 void hobgoblin::do_interact_with_chainsaw()
16 {
17     // imagine horrible, gory things here such as
18     // having to deal with a singleton
19 }
```

```
20
21 std::string hobgoblin::do_name() const
22 {
23     static std::string const name("Furry hobgoblin of nitwittery +5");
24     return name;
25 }
```

DEALING WITH MONSTERS

To really benefit from the interface, you should use it everywhere to address concrete monsters, e.g., example in free-standing functions like this:

```
1 std::string taunt_monster(monster const & tauntee)
2 {
3     return tauntee.name() + ", you are big and ugly and so is your mama!";
4 }
```

Concrete monster classes should only occur once in the (factory) code where concrete monster instances are created. Since copying monsters around is a problem, monsters are typically held ~~in cages~~ by smart pointers such as `std::shared_ptr<monster>` or `std::unique_ptr<monster>`. Smart pointers guarantee that monsters are kept alive as long as they are needed.

Enjoy populating your dungeon. Do not forget to add the fight-back feature, though…

” http://clean-cpp.org/interfaces-done-right/

DECLARE OVERRIDING FUNCTIONS *OVERRIDE*

VIRTUAL FUNCTION OVERRIDING

For overriding to occur, several requirements must be met:

- The base class function must be virtual.
- The base and derived function names must be identical (except in the case of destructors).
- The parameter types of the base and derived functions must be identical.
- The `const`ness of the base and derived functions must be identical.

- The return types and exception specifications of the base and derived functions must be compatible.
- New in C++11: The functions' *reference qualifiers* must be identical. Member 1 function reference qualifiers are one of C++11's less-publicized features, so don't be surprised if you've never heard of them. They make it possible to limit use of a member function to lvalues only or to rvalues only. Member functions need not be virtual to use them:

```
class Widget {
7 public:
8 …
9 void doWork() &; // this version of doWork applies
10 // only when *this is an lvalue
11 void doWork() &&; // this version of doWork applies
12 }; // only when *this is an rvalue
13 …
14 Widget makeWidget(); // factory function (returns rvalue)
15 Widget w; // normal object (an lvalue)
16 …
17 w.doWork(); // calls Widget::doWork for lvalues
18 // (i.e., Widget::doWork &)
19 makeWidget().doWork(); // calls Widget::doWork for rvalues
```

- 20 `// (i.e., Widget::doWork &&)`

The above code sample:
Simply note that if a virtual function in a base class has a reference qualifier, derived class overrides of that function must have exactly the same reference qualifier. If they don't, the declared functions will still exist in the derived class, but they won't override anything in the base class.

Because declaring derived class overrides is important to get right, but easy to get wrong, C++11 gives you a way to make explicit that a derived class function is supposed to override a base class version. Declare it `override`. Applying this to the example above would yield this derived class: 6

```
class Base {
7 public:
8 virtual void mf1() const;
9 virtual void mf2(int x);
10 virtual void mf3() &;
11 virtual void mf4() const;
12 };
13 class Derived: public Base {
14 public:
15 virtual void mf1() const override;
16 virtual void mf2(int x) override;
17 virtual void mf3() & override;
18 void mf4() const override; // adding "virtual" is OK,
19 }; // but not necessary
```

If we want to write a function that accepts only lvalue arguments, we declare a
17 non-`const` lvalue reference parameter:

```
18 void doSomething(Widget& w); // accepts only lvalue Widgets
```

19 If we want to write a function that accepts only rvalue arguments, we declare an

20 rvalue reference parameter:
21 `void doSomething(Widget&& w); // accepts only rvalue Widgets`

Applying `final` to a virtual function prevents the function from being overridden in derived classes. `final` may also be applied to a class, in which case the class is prohibited from being used as a base class.

DECLARE FUNCTION NOEXCEPT IF THEY WON'T EMIT EXCEPTIONS

The fact that swapping higher-level data structures can generally be noexcept only if swapping their lower-level constituents is noexcept should motivate you to offer noexcept swap functions whenever you can.

I noted at the beginning of this Item that noexcept is part of a function's interface, so you should declare a function noexcept only if you are willing to commit to a noexcept implementation over the long term. If you declare a function noexcept and later regret that decision, your options are bleak. You can remove noexcept from the function's declaration (i.e., change its interface), thus running the risk of breaking client code. You can change the implementation such that an exception could escape, yet keep the original (now incorrect) exception specification. If you do that, your program will be terminated if an exception tries to leave the function. Or you can resign yourself to your existing implementation, abandoning whatever kindled your desire to change the implementation in the first place. None of these options is appealing.

The fact of the matter is that most functions are exception-neutral. Such functions throw no exceptions themselves, but functions they call might emit one. When that happens, the exception-neutral function allows the emitted exception to pass through on its way to a handler further up the call chain. Exception-neutral functions are never noexcept, because they may emit such "just passing through" exceptions. Most functions, therefore, quite properly lack the noexcept designation. Some functions, however, have natural implementations that emit no exceptions, and for a few more—notably the move operations and swap—being noexcept can have such a significant payoff, it's worth implementing them in a noexcept manner if at all possible.† When you can honestly say that a function should never emit exceptions, you should definitely declare it noexcept.

Please note that I said some functions 1 have natural noexcept implementations. Twisting a function's implementation to permit a noexcept declaration is the tail wagging the dog. Is putting the cart before the horse. Is not seeing the forest for the trees. Is…choose your favorite metaphor. If a straightforward function implementation might yield exceptions (e.g., by invoking a function that might throw), the hoops you'll jump through to hide that from callers (e.g., catching all exceptions and replacing them with status codes or special return values) will not only complicate your function's implementation, it will typically complicate code at call sites, too. For example, callers may have to check for status codes or special return values. The runtime cost of those complications (e.g., extra branches, larger functions that put more pressure on instruction caches, etc.) could exceed any speedup you'd hope to achieve via noexcept, plus you'd be saddled with source code that's more difficult to comprehend and maintain. That'd be poor software engineering.

It's worth noting that some library interface designers distinguish functions with *wide contracts* from those with *narrow contracts*. A function with a wide contract has no preconditions. Such a function may be called regardless of the state of the program, and it imposes 1 no constraints on the arguments that callers pass it.† Functions with wide contracts never exhibit undefined behavior. Functions without wide contracts have narrow contracts. For such functions, if a precondition is violated, results are undefined.

CONTEXPR

“A constexpr function has some very rigid rules it must follow:

- It must consist of single return statement (with a few exceptions)
- It can call only other constexpr functions
- It can reference only constexpr global variables

Notice that one thing that isn't restricted is recursion. How can you do recursion if the function can only have a single return statement? By using the ternary operator (sometimes known as the question mark colon operator). For example, here's a function that computes the value of a specific factorial:

```
1   constexpr factorial (int n)
2   {
3       return n > 0 ? n * factorial( n - 1 ) : 1;
4   }
```

Now you can use factorial(2) and when the compiler sees it, it can optimize away the call and make the calculation entirely at compile time. In this way, by allowing more sophisticated calculations, constexpr behaves differently than a mere inline function. You can't inline a recursive function! In fact, any time the function argument is itself a constexpr, it can be computed at compile time.

WHAT ELSE CAN GO IN A CONSTEXPR FUNCTION?

A constexpr function can have only a single line of executable code, but it may contain typedefs, using declarations and directives, and static_asserts.

”http://www.cprogramming.com/c++11/c++11-compile-time-processing-with-constexpr.html

UNDERSTAND SPECIAL MEMBER FUNCTION GENERATION

C++11 two new “special member functions”:

class Widget {

public:

…

Widget(Widget&& rhs); // move constructor

Widget& operator=(Widget&& rhs); // move assignment operator

…

```
};
```

That means that the move constructor move-constructs each non-static data member of the class from the corresponding member of its parameter rhs, and the move assignment operator move-assigns each non-static data member from its parameter. The move constructor also move-constructs its base class parts (if there are any), and the move assignment operator move-assigns its base class parts.

The heart of each memberwise "move" is application of `std::move` to the object to be moved from, and the result is used during function overload resolution to determine whether a move or a copy should be performed. Simply remember that a memberwise move consists of move operations on data members and base classes that support move operations, but a copy operation for those that don't.

As is the case with the copy operations, the move operations aren't generated if you declare them yourself. However, the precise conditions under which they are generated differ a bit from those for the copy operations. The two copy operations are independent: declaring one doesn't prevent compilers from generating the other. So if you declare a copy constructor, but no copy assignment operator, then write code that requires copy assignment, compilers will generate the copy assignment operator for you. Similarly, if you declare a copy assignment operator, but no copy constructor, yet your code requires copy construction, compilers will generate the copy constructor for you. That was true in C++98, and it's still true in C++11.

The two move operations are not independent. If you declare either, that prevents compilers from generating the other. The rationale is that if you declare, say, a move constructor for your class, you're indicating that there's something about how move construction should be implemented that's different from the default memberwise move that compilers would generate. And if there's something wrong with memberwise move construction, there'd probably be something wrong with memberwise move assignment, too. So declaring a move constructor prevents a move assignment operator from being generated, and declaring a move assignment operator prevents compilers from generating a move constructor.

Furthermore, move operations won't be generated for any class that explicitly declares a copy operation. The justification is that declaring a copy operation (construction or assignment) indicates that the normal approach to copying an object (memberwise copy) isn't appropriate for the class, and compilers figure that if memberwise copy isn't appropriate for the copy operations, memberwise move probably isn't appropriate for the move operations.

This goes in the other direction, too. Declaring a move operation (construction or assignment) in a class causes compilers to disable the copy operations. (The copy operations are disabled by deleting them). After all, if memberwise move isn't the proper way to move an object, there's no reason to expect that memberwise copy is the proper way to copy it. This may sound like it could break C++98 code, because the conditions under which the copy operations are enabled are more constrained in C++11 than in C++98, but this is not the case. C++98 code can't have move operations, because there was no such thing as "moving" objects in C++98. The only way a legacy class can have user-declared move operations is if they were added for C++11, and classes that are modified to take advantage of move semantics have to play by the C++11 rules for special member function generation.

So move operations are generated for classes (when needed) only if these three things are true:

- No copy operations are declared in the class.

- No move operations are declared in the class.
- No destructor is declared in the class.

declaring a destructor has a potentially significant side effect: it prevents the move operations from being generated.

The simple act of adding a destructor to the class could thereby have introduced a significant performance problem! Had the copy and move operations been explicitly defined using "`= default`", the problem would not have arisen.

Default constructor: Same rules as C++98. Generated only if the class contains no user-declared constructors. **Destructor**: Essentially same rules as C++98; sole difference is that destructors are `noexcept` by default. As in C++98, virtual only if a base class destructor is virtual. **Copy constructor**: Same runtime behavior as C++98: memberwise copy construction of non-static data members. Generated only if the class lacks a user declared copy constructor. Deleted if the class declares a move operation. Generation of this function in a class with a user-declared copy assignment operator or destructor is deprecated. **Copy assignment operator**: Same runtime behavior as C++98: memberwise copy assignment of non-static data members. Generated only if the class lacks a user-declared copy assignment operator. Deleted if the class declares a move operation. Generation of this function in a class with a user-declared copy constructor or destructor is deprecated. **Move constructor** and **move assignment operator**: Each performs memberwise moving of non-static data members. Generated only if the class contains no user-declared copy operations, move operations, or destructor.

USING CONSTEXPR TO IMPROVE SECURITY, PERFORMANCE AND ENCAPSULATION IN C++

"*`constexpr`* IS A NEW C++11 KEYWORD THAT RIDS YOU OF THE NEED TO CREATE MACROS AND HARDCODED LITERALS. IT ALSO GUARANTEES, UNDER CERTAIN CONDITIONS, THAT OBJECTS UNDERGO STATIC INITIALIZATION. DANNY KALEV SHOWS HOW TO EMBED*`constexpr`* IN C++ APPLICATIONS TO DEFINE CONSTANT EXPRESSIONS THAT MIGHT NOT BE SO CONSTANT OTHERWISE.

The new C++11 keyword `constexpr` controls the evaluation time of an expression. By enforcing compile-time evaluation of its expression,`constexpr` lets you define true constant expressions that are crucial for time-critical applications, system programming, templates, and generally speaking, in any code that relies on compile-time constants.

NOTE: I USE THE PHRASES COMPILE-TIME EVALUATION AND STATIC INITIALIZATION IN THIS ARTICLE INTERCHANGEABLY, AS WELL AS DYNAMIC INITIALIZATION AND RUNTIME EVALUATION, RESPECTIVELY.

TIMING IS EVERYTHING

Before discussing `constexpr`, I need to clarify the difference between traditional `const` and the new `constexpr`.

As we all know, `const` guarantees that a program doesn't change a variable's value. However, `const` doesn't guarantee which type of initialization the variable undergoes. For instance, a `const` variable initialized by a function call requires dynamic initialization (the function is called at runtime; consequently, the constant is initialized at runtime). With certain functions, there's no justification for this restriction because the compiler can evaluate the function call statically, effectively replacing the call with a constant value. Consider:

```
const int mx = numeric_limits<int>::max(); // dynamic initialization
```

The function `max()` merely returns a literal value. However, because the initializer is a function call, `mx` undergoes dynamic initialization. Therefore, you can't use it as a *constant expression*:

```
int arr[mx]; //compilation error: "constant expression required"
```

A similar surprise occurs when the initializer of a class's `const static` data member comes "too late":

```
struct S {

static const int sz;

};
```

```
const int page_sz = 4 * S::sz; //OK, but dynamic initialization

const int S::sz = 256; //OK, but too late
```

Here the problem is that the initializer of `S::sz` appears after the initialization of <`page_sz`. Consequently, `page_sz` undergoes dynamic initialization. That isn't just slower than static initialization; it also disqualifies`S::sz` from being used as a CONSTANT INTEGRAL EXPRESSION, as the following example shows:

```
enum PAGE

{

 Regular=page_sz, //compilation error: "constant expression required"

 Large=page_sz*2 //compilation error: "constant expression required"

};
```

PROBLEMS WITH PRE-C++11 WORKAROUNDS

In pre-C++11 code, the common workaround for the late function call problem is a macro. However, macros are usually a bad choice for various reasons, including the lack of type-safety and debugging difficulties. C++ programmers thus far were forced to choose between code safety (i.e., calling a function and thereby sacrificing efficiency) and performance (i.e., using type-unsafe macros).

With respect to a `const static` class member, the common workaround is to move the initializer into the class body:

```
struct S {

static const int sz=256;
```

```
};

const int max_sz = 4 * S::sz; // static initialization

enum PAGE

{

 Regular=page_size, //OK

 Large=page_size*2 //OK

};
```

However, moving the initializer into the class body isn't always an option (for example, if `S` is a third-party class).

Another problem with `const` is that not all programmers are aware of its subtle rules of static and dynamic initialization. After all, the compiler usually doesn't tell you which type of initialization it uses for a `const` variable. For example, `mx` seems to be a constant expression when it isn't.

The aim of `constexpr` is to simplify the rules of creating constant expressions by guaranteeing that expressions declared `constexpr` undergo static initialization when certain conditions are fulfilled.

CONSTANT EXPRESSION FUNCTIONS

Let's look at `numeric_limits<int>::max()` again. My implementation defines this function like this:

```
#define INT_MAX (2147483647)
class numeric_limits<int>
{
```

```
public:
 inline static inline int max () { return INT_MAX; }
};
```

Technically, a call to `max()` could make a perfect constant expression because the function consists of a single return statement that returns a literal value. C++11 lets you do exactly that by turning `max()` into a CONSTANT EXPRESSION FUNCTION. A constant expression function is one that fulfills the following requirements:

- It returns a value (i.e., it isn't `void`).
- Its body consists of a single statement of the form

  `return exp;`

where `>expr` is a constant expression.

- The function is declared `constexpr`.

Let's examine a few examples of constant expression functions:

```
constexpr int max()
{return INT_MAX;} //OK
constexpr long long_max()
{ return 2147483647; } //OK
constexpr bool get_val()
{
  bool res=false;
  return res;
} //error, body isn't just a return statement
```

Put differently, a `constexpr` function is a named constant expression with parameters. It's meant to replace macros and hardcoded literals without sacrificing performance or type safety.

`constexpr` functions guarantee compile-time evaluation so long as their arguments are constant expressions, too. However, if any of the arguments isn't a constant expression, the `constexpr` function may be evaluated dynamically. Consider:

```
constexpr int square(int x)
```

```
{ return x * x; } //OK, compile time evaluation only if x is a constant
expression
const int res=square(5); //compile-time evaluation of square(5)
int  y=getval();
int n=square(y);//dynamic evaluation of square(y)
```

The automatic defaulting to dynamic initialization lets you define a single`constexpr` function that accepts both constant expressions and non-constant expressions. You should also note that, unlike ordinary functions, you can't call a constant expression function before it's defined.

CONSTANT EXPRESSION DATA

A CONSTANT-EXPRESSION VALUE is a variable or data member declared`constexpr`. It must be initialized with a constant expression or an rvalueconstructed by a constant expression constructor with constant expression arguments (I discuss constant expression constructors shortly). A`constexpr` value behaves as if it was declared `const`, except that it requires initialization before use and its initializer must be a constant expression. Consequently, a `constexpr` variable can always be used as part of another constant expression. For example:

```
struct S
{
private:
 static constexpr int sz; // constexpr variable
public:
 constexpr int two(); //constexpr function
 };
constexpr int S::sz = 256;
enum DataPacket
{
Small=S::two(), //error. S::two() called before it was defined
Big=1024
};
constexpr int S::two() { return sz*2; }
```

```
constexptr S s;
int arr[s.two()]; //OK, s.two() called after its definition
```

CONSTANT EXPRESSION CONSTRUCTORS

By default, an object with a nontrivial constructor undergoes dynamic initialization. However, under certain conditions, C++11 lets you declare a class's constructor `constexpr`. A `constexpr` constructor allows the compiler to initialize the object at compile-time, provided that the constructor's arguments are all constant expressions.

Formally, a CONSTANT EXPRESSION CONSTRUCTOR is one that meets the following criteria:

- It's declared `constexpr` explicitly.
- It can have a member initialization list involving only potentially constant expressions (if the expressions used aren't constant expressions then the initialization of that object will be dynamic).
- Its body must be empty.

An object of a user-defined type constructed with a constant expression constructor and constant expression arguments is called a USER-DEFINED LITERAL. Consider the following class `complex`:

```
struct complex
{
 //a constant expression constructor
  constexpr complex(double r, double i) : re(r), im(i) { }//empty body
//constant expression functions
 constexpr double real() { return re;}
  constexpr double imag() { return im;}
private:
 double re;
 double im;
};
constexpr complex COMP(0.0, 1.0); // creates a literal complex
```

As you can see, a constant expression constructor is a private case of a constant expression function except that it doesn't have a return value (because constructors don't return values). Typically, the memory layout of`COMP` is similar to that of an array of two `double`s. One of the advantages of user-defined literals with a small memory footprint is that an implementation can store them in the system's ROM. Without a `constexpr` constructor, the object would require dynamic initialization and therefore wouldn't be ROM-able.

As with ordinary constant expression functions, the `constexpr` constructor may accept arguments that aren't constant expressions. In such cases, the initialization is dynamic:

```
double x = 1.0;
constexpr complex cx1(x, 0); // error: x isn't a constant expression
const complex cx2(x, 1); //OK, dynamic initialization
constexpr double xx = COMP.real(); // OK
constexpr double imaglval=COMP.imag(); //OK, static init
complex cx3(2, 4.6); //dynamic initialization
```

Notice that if the initializer is a constant that isn't declared `constexpr`, the implementation is free to choose between static and dynamic initialization. However, if the initializer is declared `constexpr`, the object undergoes static initialization.

IN CONCLUSION

`constexpr` is an effective tool for ensuring compile-time evaluation of function calls, objects and variables. Compile-time evaluation of expressions often leads to more efficient code and enables the compiler to store the result in the system's ROM. Additionally, `constexpr` can be used wherever a constant expression is required, such as the size of arrays and bit-fields, as an enumerator's initializer, or in the forming of another constant expression. With respect to compiler support, GCC 4.6, Intel's C++ 13, IBM's XLC++ 12.1, and Clang 3.1 already support `constexpr`.

" http://blog.smartbear.com/c-plus-plus/using-constexpr-to-improve-security-performance-and-encapsulation-in-c/

C++11 TUTORIAL: NEW CONSTRUCTOR FEATURES THAT MAKE OBJECT INITIALIZATION FASTER AND SMOOTHER

"CONSTRUCTORS IN C++11 STILL DO WHAT THEY'VE ALWAYS DONE: INITIALIZE AN OBJECT. HOWEVER, TWO NEW FEATURES, NAMELY DELEGATING CONSTRUCTORS AND CLASS MEMBER INITIALIZERS, MAKE CONSTRUCTORS SIMPLER, EASIER TO MAINTAIN, AND GENERALLY MORE EFFICIENT. LEARN HOW TO COMBINE THESE NEW FEATURES IN YOUR C++ CLASSES.

C++11 introduced several constructor-related enhancements including:

- Delegating constructors
- Class member initializers

The interaction between these two features enables you to tailor your classes' constructors to achieve safer and simpler initialization. In this article, I demonstrate the use of delegating constructors, explain what class member initializers are, and show how to combine both of them in your classes.

CONSTRUCTIVE TALKS

Earlier variants of C++ lack a mechanism for calling a constructor from another constructor of the same class (constructors of the same class are known as SIBLING CONSTRUCTORS). This limitation is chiefly noticeable in cases where default arguments aren't an option. Consequently, the class's maintainer has to write multiple constructors that repeat similar initialization code segments.

Thus far, the common workaround has been to define a separate initialization member function, and call it from every constructor:

```
class C
```

```
{
 void init();//comon initialization tasks
 int s;
 T t; //T has a conversion operator to double
public:
 C(): s(12), t(4.5) {init();}
 C(int i) : s(i), t(4.5) {init();}
 C(T& t1) : s(12), t(t1) {init();}
};
```

This approach has two drawbacks. First, delegating the initialization task to `init()` is inefficient because by the time `init()` executes, the data members of C have already been constructed. In other words, `init()` merely REASSIGNS new values to C's data members. It doesn't really initialize them. (By contrast, initialization by means of a member initialization list takes place before the constructor's body executes.)

The second problem is obvious: Every constructor repeats identical bits of initialization code. Not only is it wearisome, it's also a recipe for future maintenance problems. Every time you modify the class's members or its members' initialization protocol, you'd need to update multiple constructors accordingly.

INTRODUCING TARGET CONSTRUCTORS

C++11 solves this problem by allowing a constructor (known as theDELEGATING CONSTRUCTOR) to call another sibling constructor (the TARGET CONSTRUCTOR) from the delegating constructor's member initialization list. For example:

```
class C //a C11 class with a target constructor
{
 int s;
 T t;  //has a conversion operator to double
public:
//the target constructor
 C(int i, T& t1): s(i), t(t1) {/*more common init code*/}
//three delegating ctors that call the target ctor
 C(): C(12, 4.5) {/*specific post-init code*/}
 C(int i): C(i, 4.5) {/*specific post-init code*/}
 C(T& t1): C(12, t1) {/*specific post-init code*/}
};
```

The modified class C now defines a fourth constructor: the target constructor. The three delegating constructors call it from their member initialization list to perform the common initialization procedure. Once the target constructor returns, the body of the delegating constructor is executed.

Notice that there are no special syntactic markers for target and delegating constructors. A target constructor is defined as such if another constructor of the same class calls it. Likewise, a delegating constructor is a constructor that calls a sibling constructor in its member initialization list.

From this you can infer that a target constructor can also be a delegating constructor if it calls another target constructor. However, if a constructor delegates to itself directly or indirectly, the program is ill-formed. The following example demonstrates a target constructor that calls another target constructor:

```
class Z //C++11, a target ctor that's also a delegating ctor

{

 int j;

 bool flag;

public:

 Z(): Z(5) {} //#1

 explicit Z(int n) :Z(n, false) {} //#2

 Z(int n, bool b): j(n), flag(b) {} //#3

};
```

Constructor #1 calls the target constructor #2. Constructor #2 is also a delegating constructor since it calls the target constructor #3.

Technically, the use of default arguments could make two of the three constructors of class Z redundant:

```
class Z //using default arguments to minimize ctors

{

 int j;
```

```
 bool flag;

public:

 Z(int n=5, bool b=false): j(n), flag(b) {}

};
```

However, default arguments aren't always desirable (or even possible) for various reasons. Therefore, delegating constructors that call a target constructor are a convenient and more efficient alternative to `init()`.

CLASS MEMBER INITIALIZERS

C++11 introduced another related feature called CLASS MEMBER INITIALIZERSthat could make the design and implementation of your constructors even simpler.

Note: Class member initializers are also called in-class initializers. I use both terms interchangeably in this article.

In this case too, C++11 follows other programming languages that let you initialize a data member directly in its declaration:

```
class M //C++11

{

 int j=5; //in-class initializer

 bool flag (false); //another in-class initializer

public:

 M();
```

};

M m1; //m1.j = 5, m1.flag=false

Under the hood, the compiler transforms every class member initializer (such as `int j=5;`) into a constructor's member initializer. Therefore, the definition of class M above is semantically equivalent to the following C++03 class definition:

```
class M2
{
 int j;
 bool flag;
public:
 M2(): j(5), flag(false) {}
};
```

If the constructor includes an explicit member initializer for a member that also has an in-class initializer, the constructor's member initializer takes precedence, effectively overriding the in-class initializer for that particular constructor. Consider:

```
class M2
{
 int j=7;
public:
```

```
 M2(); //j=7

 M2(int i): j(i){}//overrides j's in-class initializer

};

M2 m2; //j=7,

M2 m3(5); //j=5
```

This property of class member initializers makes them useful for defining a "default" value that constructors may override individually. In other words, a constructor that needs to override the default value for a certain member can use an explicit member initializer for it. Otherwise, the in-class initializer takes effect.

Class member initializers can simplify the design of a class's constructor. You can specify the initializer for a given member directly with its declaration instead of using a target constructor for other constructors to call. If, however, a certain member requires a more complex initialization procedure (for example, its value depends on other data members' values, or if its value depends on a specific constructor), a combination of traditional constructor member initializers and target constructors does the trick.

Remember that if a target constructor appears in a constructor's member initialization list, that list shall contain nothing else. No other base or member initializers are allowed in this case.

Let's look again at the constructors of class C to see how to combine class member initializers and delegating constructors. Recall that the original constructor set of C looks like this:

```
//a target constructor

C(int i, T& t1): s(i), t(t1) {/*common init code*/}
```

//three delegating ctors invoke the target

C(): C(12, 4.5) {/*specific post-init code*/}

C(int i): C(i, 4.5) {/*specific post-init code*/}

C(T& t1): C(12, t1) {/*specific post-init code*/}

The member s is initialized to 12 in two out of four constructors. You can therefore use a class member initializer for it, while leaving constructor-dependent initializers in member initializer lists or in the single target constructor:

```
class C //C11, combine in-class init with a target ctor
```

class C //C11, combine in-class init with a target ctor

{

 int s=12;

 T t;

public:

//a target constructor

 C(int i, T& t1): s(i), t(t1) {}

 C(): t(4.5) {} //s=12, t=4.5

 C(int i): C(i, 4.5) {}//using a target ctor. s=i, t=4.5

 C(T& t1): t(t1) {} //s=12, t=t1

```
};
```

Of course, you can use a different combination such as an in-class initializer for the member t instead of s, or use in-class initializers for both. There is no right or wrong here. Rather, the aim is to reduce the amount of repeated initialization code and to minimize the complexity of the constructors while ensuring that data members are initialized properly. Most of all, the aim is to get rid of the `init()` hack.

Apart from making your code more readable and easier to maintain, delegating constructors and class member initializers offer a hidden bonus. They also improve your constructors' performance because they literally initialize data members, as opposed to reassigning a new value to them (which is what `init()`-like member functions actually do).

Therefore, it might be a good idea to go through your existing pre-C++11 code and refactor it with the following guidelines in mind: Simple initializers that occur frequently should become class member initializers in general, whereas ad-hoc initializers should be dealt with by target constructors and member initializer lists.

" http://blog.smartbear.com/how-to/c11-tutorial-new-constructor-features-that-make-object-initialization-faster-and-smoother/

SMART POINTERS

- **Use std::unique_ptr for exclusive-ownership re-source management**
  - std::unique_ptr is a small, fast, move-only smart pointer for managing resources with exclusive-ownership semantics.
  - By default, resource destruction takes place via delete, but custom deleters can be specified. Stateful deleters and function pointers as deleters increase the size of std::unique_ptr objects.
  - Converting a std::unique_ptr to a std::shared_ptr is easy.
- **Use std::sharePtr for shared-ownership resource management.**
  - std::shared_ptrs offer convenience approaching that of garbage collection for the shared lifetime management of arbitrary resources.
  - Compared to std::unique_ptr, std::shared_ptr objects are twice as big, incur overhead for control blocks, and require atomic reference count manipulations.
  - Default resource destruction is via delete, but custom deleters are supported. The type of the delete has no effect on the type of the std::shared_prt.
  - Avoid creating std::shared_ptrs from variables of raw pointer type.
- **Use std::weak_ptr for std::share_ptr like pointers that can dangle**

- Use std_weak_ptr for std_shared_ptr –like pointers that can dangle
- Potential use cases for std::weak_ptr include caching, observer list, and the prevention of std:shared_ptr cycles.

- **Prefer std::make_unique and std::make_shared to direct user of new**
  - Compared to direct use of new, make functions eliminate source code duplication, improve exception safety, and, for std::make_shared and std::allocate_shared, generate code that's smaller and faster.
  - Situations where to use make functions is inappropriate include the need to specify custom deleters and desire to pass braced initializers.
  - For std::shared_ptrs, additional situations where make functions may be ill-advised include (1) classes with custom memory management and (2) systems with memory concerns, very large objects, and std::weak_ptrs that outlive the corresponding std::shared_ptrs.
- **When using the Pimpl Idion, define special member functions in the implementation file**
  - The Pimpl Idiom decreases build times by reducing compilation dependencies between class clients and class implementations.
  - For `std::unique_ptr pImpl` pointers, declare special member functions in the class header, but implement them in the implementation file. Do this even if the default function implementations are acceptable.
  - The above advice applies to `std::unique_ptr`, but not to `std::shared_ptr`.

Smart pointers are one way to address the raw pointers issues. Smart pointers are wrappers around raw pointers that act much like the raw pointers they wrap, but that avoid many of their pitfalls. You should therefore prefer smart pointers to raw pointers. Smart pointers can do virtually everything raw pointers can, but with far fewer opportunities for error.

There are four smart pointers in C++11: `std::auto_ptr`, `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`. All are designed to help manage the life times of dynamically allocated objects, i.e., to avoid resource leaks by ensuring that such objects are destroyed in the appropriate manner at the appropriate time (including in the event of exceptions).

`std::unique_ptr` does everything `std::auto_ptr` does, plus more. It does it as efficiently, and it does it without warping what it means to copy an object. It's better than `std::auto_ptr` in every way. The only legitimate use case for `std::auto_ptr` is a need to compile code with C++98 compilers. Unless you have that constraint, you should replace `std::auto_ptr` with `std::unique_ptr` and never look back.

C++11 SMART POINTERS

"Introduction
Ooops. Yet another article on smart pointers of C++11. Nowadays I hear a lot of people talking about the new C++ standard which is nothing but C++0x/C++11. I went through some of the

language features of C++11 and it's really an amazing work. I'll focus only on the smart pointers section of C++11.

Background

What are the issues with normal/raw/naked pointers?

Let's go one by one.

People refrain from using pointers as they give a lot of issues if not handled properly. That's why newbie programmers dislike pointers. Many issues are involved with pointers like ensuring the lifetime of objects referred to by pointers, dangling references, and memory leaks.

Dangling reference is caused if a memory block is pointed by more than one pointer variable and if one of the pointers is released without letting know the other pointer. As all of you know, memory leaks occur when a block of memory is fetched from the heap and is not released back.

People say, I write clean and error proof code, why should I use smart pointers? And a programmer asked me, "Hey, here is my code. I fetched the memory from the heap, manipulated it, and after that I released it properly. What is the need of a smart pointer? "

```
void Foo( )
{
   int* iPtr = new int[5];

   //manipulate the memory block
   .
   .
   .

   delete[ ] iPtr;
}
```

The above code works fine and memory is released properly under ideal circumstances. But think of the practical environment of code execution. The instructions between memory allocation and releasing can do nasty things like accessing an invalid memory location, dividing by zero, or say another programmer pitching into your program to fix a bug and adding a
premature return statement based on some condition.

In all the above cases, you will never reach the point where the memory is released. This is because the first two cases throw an exception whereas the third one is a premature return. So the memory gets leaked while the program is running.

The one stop solution for all of the above issues is Smart Pointers [if they are really smart enough].

What is a smart pointer?

Smart pointer is a RAII modeled class to manage dynamically allocated memory. It provides all the interfaces provided by normal pointers with a few exceptions. During construction, it owns the memory and releases the same when it goes out of scope. In this way, the programmer is free about managing dynamically allocated memory.

C++98 has introduced the first of its kind called auto_ptr.

auto_ptr

Let's see the use of auto_ptr and how smart it is to resolve the above issues.

```
class Test
{
public:
 Test(int a = 0 ) : m_a(a)
 {
 }
 ~Test( )
 {
```

```
 cout<<"Calling destructor"<<endl;
 }
public:
 int m_a;
};

void main( )
{
 std::auto_ptr<Test> p( new Test(5) );
 cout<<p->m_a<<endl;
}
```

The above code is smart to release the memory associated with it. What we did is, we fetched a memory block to hold an object of type Test and associated it with auto_ptr p. So when p goes out of scope, the associated memory block is also released.

```
//*************************************************************
class Test
{
public:
 Test(int a = 0 ) : m_a(a)
 {
 }
 ~Test( )
 {
  cout<<"Calling destructor"<<endl;
 }
public:
 int m_a;
};
//*************************************************************
void Fun( )
{
 int a = 0, b= 5, c;
 if( a ==0 )
 {
  throw "Invalid divisor";
 }
 c = b/a;
 return;
}
//*************************************************************
void main( )
{
 try
 {
  std::auto_ptr<Test> p( new Test(5) );
  Fun( );
  cout<<p->m_a<<endl;
 }
 catch(...)
 {
```

```
  cout<<"Something has gone wrong"<<endl;
 }
}
```
In the above case, an exception is thrown but still the pointer is released properly. This is because of stack unwinding which happens when an exception is thrown. As all local objects belonging to the try block are destroyed, p goes out of scope and it releases the associated memory.

Issue 1: So far auto_ptr is smart. But it has more fundamental flaws over its smartness. auto_ptr transfers the ownership when it is assigned to another auto_ptr. This is really an issue while passing theauto_ptr between the functions. Say, I have an auto_ptr in Foo( ) and this pointer is passed another function say Fun( ) from Foo. Now once Fun( ) completes its execution, the ownership is not returned back to Foo.

```
//*************************************************************
class Test
{
public:
 Test(int a = 0 ) : m_a(a)
 {
 }
 ~Test( )
 {
  cout<<"Calling destructor"<<endl;
 }
public:
 int m_a;
};


//*************************************************************
void Fun(auto_ptr<Test> p1 )
{
 cout<<p1->m_a<<endl;
}
//*************************************************************
void main( )
{
 std::auto_ptr<Test> p( new Test(5) );
 Fun(p);
 cout<<p->m_a<<endl;
}
```
The above code causes a program crash because of the weird behavior of auto_ptr. What happens is that, p owns a memory block and when Fun is called, p transfers the ownership of its associated memory block to the auto_ptr p1 which is the copy of p. Now p1 owns the memory block which was previously owned by p. So far it is fine. Now fun has completed its execution, and p1 goes out of scope and the memory blocked is released. How about p? p does not own anything, that is why it causes a crash when the next line is executed which accesses p thinking that it owns some resource.

Issue 2: Yet another flaw. auto_ptr cannot be used with an array of objects. I mean it cannot be used with the operator new[].

```
//*************************************************************
void main( )
{
```

```
 std::auto_ptr<Test> p(new Test[5]);
}
```
The above code gives a runtime error. This is because when auto_ptr goes out of scope, delete is called on the associated memory block. This is fine if auto_ptr owns only a single object. But in the above code, we have created an array of objects on the heap which should be destroyed using delete[ ] and not delete.

Issue 3: auto_ptr cannot be used with standard containers like vector, list, map, etc.

As auto_ptr is more error prone and it will be deprecated, C++ 11 has come with a new set of smart pointers, each has its own purpose.

- shared_ptr
- unique_ptr
- weak_ptr

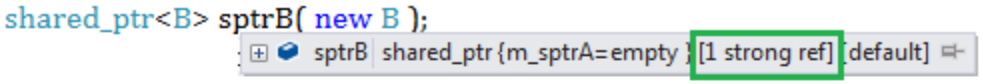
shared_ptr

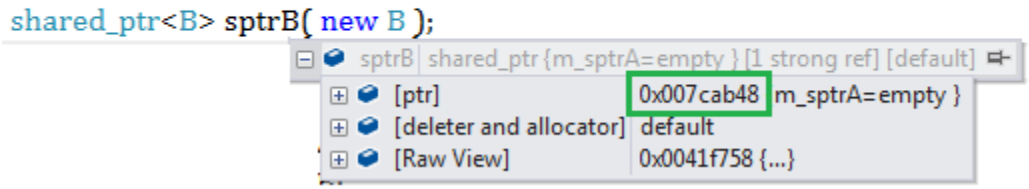
OK, get ready to enjoy the real smartness. The first of its kind is shared_ptr which has the notion called shared ownership. The goal of shared_ptr is very simple: Multiple shared pointers can refer to a single object and when the last shared pointer goes out of scope, memory is released automatically.

Creation:

```
void main( )
{
 shared_ptr<int> sptr1( new int );
}
```
Make use of the make_shared macro which expedites the creation process. As shared_ptr allocates memory internally, to hold the reference count, make_shared( ) is implemented in a way to do this job effectively.
```
void main( )
{
 shared_ptr<int> sptr1 = make_shared<int>(100);
}
```
The above code creates a shared_ptr which points to a memory block to hold an integer with value 100 and reference count 1. If another shared pointer is created out of sptr1, the reference count goes up to 2. This count is known as *strong reference*. Apart from this, the shared pointer has another reference count known as *weak reference*, which will be explained while visiting weak pointers.

You can find out the number of shared_ptrs referring to the resource by just getting the reference count by calling use_count( ). And while debugging, you can get it by watching the stong_ref of theshared_ptr.





Destruction:

shared_ptr releases the associated resource by calling delete by default. If the user needs a different destruction policy, he/she is free to specify the same while constructing the shared_ptr. The following code is a source of trouble due to the default destruction policy:

```
class Test
{
public:
 Test(int a = 0 ) : m_a(a)
 {
 }
 ~Test( )
 {
  cout<<"Calling destructor"<<endl;
 }
public:
       int m_a;
};
void main( )
{
 shared_ptr<Test> sptr1( new Test[5] );
}
```

Because shared_ptr owns an array of objects, it calls delete when it goes out of scope.
Actually, delete[ ] should have been called to destroy the array. The user can specify the custom deallocator by a callable object, i.e., a function, lambda expression, function object.

```
void main( )
{
 shared_ptr<Test> sptr1( new Test[5],
      [ ](Test* p) { delete[ ] p; } );
}
```

The above code works fine as we have specified the destruction should happen via delete[].

Interface

shared_ptr provides dereferencing operators *, -> like a normal pointer provides. Apart from that it provides some more important interfaces like:

- get( ) : To get the resource associated with the shared_ptr.
- reset( ) : To yield the ownership of the associated memory block. If this is the last shared_ptr owning the resource, then the resource is released automatically.
- unique: To know whether the resource is managed by only this shared_ptr instance.
- operator bool: To check whether the shared_ptr owns a memory block or not. Can be used with an if condition.

OK, that is all about shared_ptrs. But shared_ptrs too have a few issues:.

Issues:

1. If a memory is block is associated with shared_ptrs belonging to a different group, then there is an error. All shared_ptrs sharing the same reference count belong to a group. Let's see an example.

```
void main( )
{
 shared_ptr<int> sptr1( new int );
 shared_ptr<int> sptr2 = sptr1;
 shared_ptr<int> sptr3;
 sptr3 = sptr2;
}
```

The below table gives you the reference count values for the above code.

	Reference count
shared_ptr<int> sptr1(new int)	1
shared_ptr<int> sptr2 = sptr1	2
sptr3 = sptr2	3
When main ends -> sptr3 goes out of scope	2
When main ends -> sptr2 goes out of scope	1
When main ends -> sptr1 goes out of scope	0 -> The resource is released as the count goes down to zero.

All shared_ptrs share the same reference count hence belonging to the same group. The above code is fine. Let's see another piece of code.

```
void main( )
{
 int* p = new int;
 shared_ptr<int> sptr1( p);
 shared_ptr<int> sptr2( p );
}
```

The above piece of code is going to cause an error because two shared_ptrs from different groups share a single resource. The below table gives you a picture of the root cause.

	Reference count
shared_ptr<int> sptr1(p)	1
shared_ptr<int> sptr2(p)	1
main ends -> sptr1 goes out of scope	0 -> memory block pointed by sptr1 or p is destroyed as ref count is 0.
main ends -> sptr2 goes out of scope	0 -> **Crashhhhhhh!!!!!**. Because this tries to release memory block associated with sptr2, which is already being destroyed.

To avoid this, better not create the shared pointers from a naked pointer.

2. There is another issue involved with creating a shared pointer from a naked pointer. In the above code, consider that only one shared pointer is created using p and the code works fine. Consider by mistake if a programmer deletes the naked pointer p before the scope of the shared pointer ends. Oooppss!!! Yet another crash..

3. Cyclic Reference: Resources are not released properly if a cyclic reference of shared pointers are involved. Consider the following piece of code.

```
class B;
class A
{
public:
 A( ) : m_sptrB(nullptr) { };
 ~A( )
 {
  cout<<" A is destroyed"<<endl;
 }
 shared_ptr<B> m_sptrB;
};
class B
{
public:
 B( ) : m_sptrA(nullptr) { };
 ~B( )
 {
  cout<<" B is destroyed"<<endl;
 }
 shared_ptr<A> m_sptrA;
};
//************************************************************
void main( )
{
 shared_ptr<B> sptrB( new B );
 shared_ptr<A> sptrA( new A );
 sptrB->m_sptrA = sptrA;
 sptrA->m_sptrB = sptrB;
}
```

The above code has cyclic reference. I mean class A holds a shared pointer to B and class B holds a shared pointer to A. In this case, the resource associated with both sptrA and sptrB are not released. Refer to the below table.

	Reference count
shared_ptr<B> sptrB(new B)	1
shared_ptr<A> sptrA(new A)	1
sptrB->m_sptrA = sptrA	Ref count of sptrA -> 2
sptrA->m_sptrB = sptrB	Ref count of sptrB ->2
main ends -> sptrA goes out of scope	Ref count of sptrA ->1
main ends -> sptrB goes out of scope	Ref count of sptrB ->1

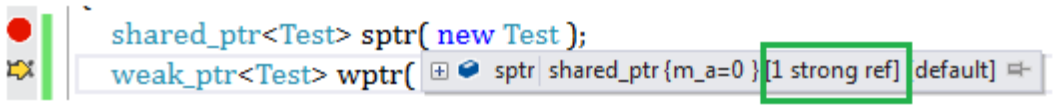
Reference counts for both sptrA and sptrB go down to 1 when they go out of scope and hence the resources are not released!!!!!
To resolve the cyclic reference, C++ provides another smart pointer class called weak_ptr.
Weak_Ptr
A weak pointer provides sharing semantics and not owning semantics. This means a weak pointer can share a resource held by a shared_ptr. So to create a weak pointer, some body should already own the resource which is nothing but a shared pointer.
A weak pointer does not allow normal interfaces supported by a pointer, like calling *, ->. Because it is not the owner of the resource and hence it does not give any chance for the programmer to mishandle it. Then how do we make use of a weak pointer?
The answer is to create a shared_ptr out of a weak _ptr and use it. Because this makes sure that the resource won't be destroyed while using by incrementing the strong reference count. As the reference count is incremented, it is sure that the count will be at least 1 till you complete using the shared_ptr created out of the weak_ptr. Otherwise what may happen is while using the weak_ptr, the resource held by theshared_ptr goes out of scope and the memory is released which creates chaos.

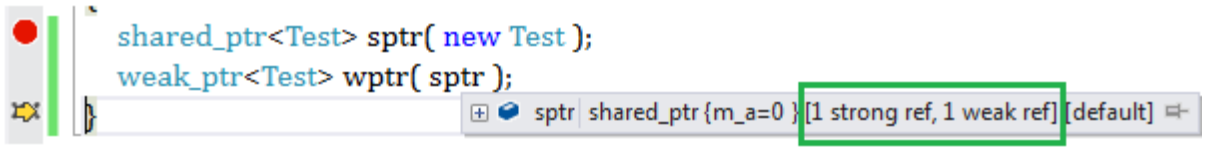
Creation

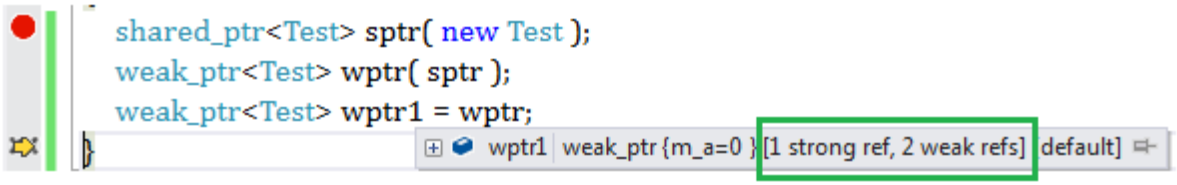
A weak pointer constructor takes a shared pointer as one of its parameters. Creating a weak pointer out of a shared pointer increases the *weak reference* counter of the shared pointer. This means that the shared pointer shares it resource with another pointer. But this counter is not considered to release the resource when the shared pointer goes out of scope. I mean if the strong reference of the shared pointer goes to 0, then the resource is released irrespective of the weak reference value.

```
void main( )
{
 shared_ptr<Test> sptr( new Test );
 weak_ptr<Test> wptr( sptr );
 weak_ptr<Test> wptr1 = wptr;
}
```

We can watch the reference counters of the shared/weak pointer.







Assigning a weak pointer to another weak pointer increases the weak reference count.
So what happens when a weak pointer points to a resource held by the shared pointer and the shared pointer destroys the associated resource when it goes out of scope? The weak pointer gets expired.
How to check whether the weak pointer is pointing to a valid resource? There are two ways:

1. Call the use_count( ) method to know the count. Note that this method returns the strong reference count and not the weak reference.
2. Call the expired( ) method. This is faster than calling use_count( ).

To get a shared_ptr from a weak_ptr call lock( ) or directly casting the weak_ptr to shared_ptr.

```
void main( )
{
 shared_ptr<Test> sptr( new Test );
 weak_ptr<Test> wptr( sptr );
 shared_ptr<Test> sptr2 = wptr.lock( );
}
```

Getting the shared_ptr from the weak_ptr increases the strong reference as said earlier.

Now let's see how the cyclic reference issue is resolved using the weak_ptr.

```
class B;
class A
{
public:
 A(  ) : m_a(5)  { };
 ~A( )
 {
  cout<<" A is destroyed"<<endl;
 }
 void PrintSpB( );
 weak_ptr<B> m_sptrB;
 int m_a;
};
class B
{
public:
 B(  ) : m_b(10) { };
 ~B( )
 {
  cout<<" B is destroyed"<<endl;
 }
 weak_ptr<A> m_sptrA;
 int m_b;
};

void A::PrintSpB( )
{
 if( !m_sptrB.expired() )
 {
  cout<< m_sptrB.lock( )->m_b<<endl;
 }
}

void main( )
{
 shared_ptr<B> sptrB( new B );
 shared_ptr<A> sptrA( new A );
 sptrB->m_sptrA = sptrA;
```

```
 sptrA->m_sptrB = sptrB;
 sptrA->PrintSpB( );
}
```

	Reference Count
shared_ptr<B> sptrB(new B)	Strong Ref = 1
shared_ptr<A> sptrA(new A)	Strong Ref = 1
sptrB->m_sptrA = sptrA	Strong Ref = 1, Weak Ref = 1
sptrA->m_sptrB = sptrB	Strong Ref = 1, Weak Ref = 1
Main ends -> sptrA goes out of scope	Strong Ref = 0 , Weak Ref = 1 As strong reference goes to 0, the resource is released.
Main ends -> sptrB goes out of scope	Strong Ref = 0 , Weak Ref = 1 As strong reference goes to 0, the resource is released.

Unique_ptr

This is almost a kind of replacement to the error prone auto_ptr. unique_ptr follows the exclusive ownership semantics, i.e., at any point of time, the resource is owned by only one unique_ptr. When auto_ptrgoes out of scope, the resource is released. If the resource is overwritten by some other resource, the previously owned resource is released. So it guarantees that the associated resource is released always.

Creation

unique_ptr is created in the same way as shared_ptr except it has an additional facility for an array of objects.

unique_ptr<int> uptr(new int);

The unique_ptr class provides the specialization to create an array of objects which calls delete[ ] instead of delete when the pointer goes out of scope. The array of objects can be specified as a part of the template parameter while creating the unique_ptr. In this way, the programmer does not have to provide a custom deallocator, as unique_ptr does it.

unique_ptr<int[ ]> uptr(new int[5]);

Ownership of the resource can be transferred from one unique_ptr to another by assigning it. Keep in mind that unique_ptr does not provide you copy semantics [copy assignment and copy construction is not possible] but move semantics.

In the above case, if upt3 and uptr5 owns some resource already, then it will be destroyed properly before owning a new resource.

Interface

The interface that unique_ptr provides is very similar to the ordinary pointer but no pointer arithmetic is allowed.

unique_ptr provides a function called release which yields the ownership. The difference between release( ) and reset( ), is release just yields the ownership and does not destroy the resource whereasreset destroys the resource.

Which one to use?

It purely depends upon how you want to own a resource. If shared ownership is needed then go for shared_ptr, otherwise unique_ptr.
Apart from that, shared_ptr is a bit heavier than unique_ptr because internally it allocates memory to do a lot of book keeping like strong reference, weak reference, etc. But unique_ptr does not need these counters as it is the only owner for the resource.
"http://www.codeproject.com/Articles/541067/Cplusplus-Smart-Pointers

---

PREFER STD::MAKE_UNIQUE AND STD::MAKE_SHARED TO DIRECT USER OF NEW

"The difference is that `std::make_shared` performs one heap-allocation, whereas calling the `std::shared_ptr` constructor performs two.

WHERE DO THE HEAP-ALLOCATIONS HAPPEN?

`std::shared_ptr` manages two entities,

- the control block (stores meta data such as ref-counts, type-erased deleter, etc)
- the object being managed

`std::make_shared` performs a single heap-allocation accounting for the space necessary for both the control block and the data. In the other case, `new Obj("foo")` invokes a heap-allocation for the managed data and the `std::shared_ptr` constructor performs another one for the control block.
For further information, check out the **implementation notes** at cppreference.

UPDATE I: EXCEPTION-SAFETY

Since the OP seem to be wondering about the exception-safety side of things, I've updated my answer.

Consider this example,

```
void F(const std::shared_ptr<Lhs> &lhs, const std::shared_ptr<Rhs> &rhs) { /* ... */ }

F(std::shared_ptr<Lhs>(new Lhs("foo")),
  std::shared_ptr<Rhs>(new Rhs("bar")));
```

Because C++ allows arbitrary order of evaluation of inner expressions, one possible ordering is:

1. `new Lhs("foo"))`
2. `new Rhs("bar"))`
3. `std::shared_ptr<Lhs>`
4. `std::shared_ptr<Rhs>`

Now, suppose we get an exception thrown at step 2 (eg. out of memory exception, `Rhs` constructor threw some exception). We then lose memory allocated at step 1, since nothing will have had a chance to clean it up. The core of the problem here is that the raw pointer didn't get passed to the `std::shared_ptr`constructor immediately.
One way to fix this is to do them on separate lines so that this arbitary ordering cannot occur.

```
auto lhs = std::shared_ptr<Lhs>(new Lhs("foo"));
```

```
auto rhs = std::shared_ptr<Rhs>(new Rhs("bar"));
F(lhs, rhs);
```

The preferred way to solve this of course is to use `std::make_shared` instead.

```
F(std::make_shared<Lhs>("foo"), std::make_shared<Rhs>("bar"));
```

UPDATE II: DISADVANTAGE OF `STD::MAKE_SHARED`

Quoting Casey's comments:

Since there there's only one allocation, the pointee's memory cannot be deallocated until the control block is no longer in use. A `weak_ptr` can keep the control block alive indefinitely.

WHY DO INSTANCES OF `WEAK_PTR`S KEEP THE CONTROL BLOCK ALIVE?

There must be a way for `weak_ptr`s to determine if the managed object is still valid (eg. for `lock`). They do this by checking the number of `shared_ptr`s that own the managed object, which is stored in the control block. The result is that the control blocks are alive until the `shared_ptr` count and the `weak_ptr`count both hit 0.

BACK TO `STD::MAKE_SHARED`

Since `std::make_shared` makes a single heap-allocation for both the control block and the managed object, there is no way to free the memory for control block and the managed object independently. We must wait until we can free both the control block and the managed object, which happens to be until there are no `shared_ptr`s or `weak_ptr`s alive.

Suppose we instead performed two heap-allocations for the control block and the managed object via `new`and `shared_ptr` constructor. Then we free the memory for the managed object (maybe earlier) when there are no `shared_ptr`s alive, and free the memory for the control block (maybe later) when there are no `weak_ptr`s alive.

” http://stackoverflow.com/questions/20895648/difference-in-make-shared-and-normal-shared-ptr-in-c

---

QUESTIONS AND ANSWERS ABOUT SMART POINTERS

“

PROBLEM

JG QUESTION

1. When should you use shared_ptr vs. unique_ptr? List as many considerations as you can.

GURU QUESTION

2. Why should you almost always use make_shared to create an object to be owned byshared_ptrs? Explain.

3. Why should you almost always use make_unique to create an object to be initially owned by aunique_ptr? Explain.
4. What's the deal with auto_ptr?

SOLUTION

1. WHEN SHOULD YOU USE SHARED_PTR VS. UNIQUE_PTR?

When in doubt, prefer unique_ptr by default, and you can always later move-convert toshared_ptr if you need it. If you do know from the start you need shared ownership, however, go directly to shared_ptr via make_shared (see #2 below).

There are three major reasons to say "when in doubt, prefer unique_ptr."

First, use the simplest semantics that are sufficient: Choose the right smart pointer to most directly express your intent, and what you need (now). If you are creating a new object and don't know that you'll eventually need shared ownership, use unique_ptr which expresses unique ownership. You can still put it in a container (e.g., vector<unique_ptr<widget>>) and do most other things you want to do with a raw pointer, only safely. If you later need shared ownership, you can always move-convert the unique_ptr to a shared_ptr.

Second, a unique_ptr is more efficient than a shared_ptr. A unique_ptr doesn't need to maintain reference count information and a control block under the covers, and is designed to be just about as cheap to move and use as a raw pointer. When you don't ask for more than you need, you don't incur overheads you won't use.

Third, starting with unique_ptr is more flexible and keeps your options open. If you start with aunique_ptr, you can always later convert to a shared_ptr via move, or to another custom smart pointer (or even to a raw pointer) via .get() or .release().

Guideline: Prefer to use the standard smart pointers, **unique_ptr** by default and **shared_ptr** if sharing is needed. They are the common types that all C++ libraries can understand. Use other smart pointer types only if necessary for interoperability with other libraries, or when necessary for custom behavior you can't achieve with deleters and allocators on the standard pointers.

2. WHY SHOULD YOU ALMOST ALWAYS USE MAKE_SHARED TO CREATE AN OBJECT TO BE OWNED BY SHARED_PTRS? EXPLAIN.

Note: If you need to create an object using a custom allocator, which is rare, you can useallocate_shared. Note that even though its name is slightly different, allocate_shared should be viewed as "just the flavor of make_shared that lets you specify an allocator," so I'm mainly going to talk about them both as make_shared here and not distinguish much between them.

There are two main cases where you can't use make_shared (or allocate_shared) to create an object that you know will be owned by shared_ptrs: (a) if you need a custom deleter, such as because of using shared_ptrs to manage a non-memory resource or an object allocated in a nonstandard memory area, you can't use make_shared because it doesn't support specifying a deleter; and (b) if you are adopting a raw pointer to an object being handed to you from other (usually legacy) code, you would construct a shared_ptr from that raw pointer directly.

Guideline: Use **make_shared** (or, if you need a custom allocator, **allocate_shared**) to create an object you know will be owned by **shared_ptr**s, unless you need a custom deleter or are adopting a raw pointer from elsewhere.

So, why use make_shared (or, if you need a custom allocator, allocate_shared) whenever you can, which is nearly always? There are two main reasons: simplicity, and efficiency. First, with make_shared the code is simpler. Write for clarity and correctness first.

Second, using make_shared is more efficient.

The shared_ptr implementation has to maintain housekeeping information in a control block shared by all shared_ptrs and weak_ptrs referring to a given object. In particular, that housekeeping information has to include not just one but two reference counts:

- A "strong reference" count to track the number of shared_ptrs currently keeping the object alive. The shared object is destroyed (and possibly deallocated) when the last strong reference goes away.
- A "weak reference" count to track the number of weak_ptrs currently observing the object. The shared housekeeping control block is destroyed and deallocated (and the shared object is deallocated if it was not already) when the last weak reference goes away.

If you allocate the object separately via a raw new expression, then pass it to a shared_ptr, theshared_ptr implementation has no alternative but to allocate the control block separately, as shown in Example 2(a) and Figure 2(a).

```
// Example 2(a): Separate allocation
auto sp1 = shared_ptr<widget>{ new widget{} };
auto sp2 = sp1;
```

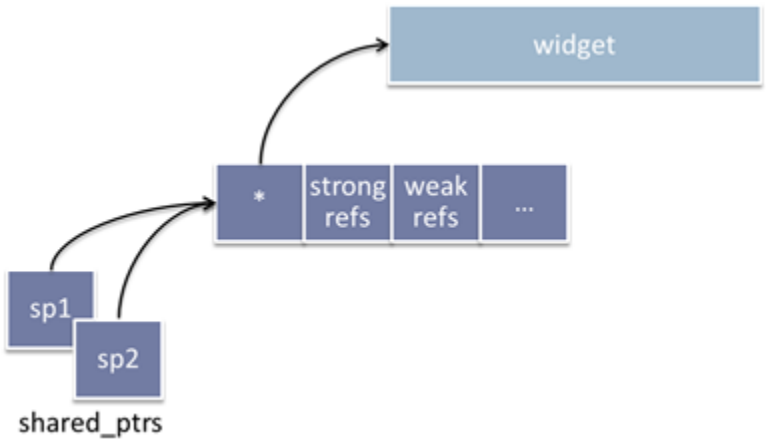


Figure 2(a): Approximate memory layout for Example 2(a).

We'd like to avoid doing two separate allocations here. If you use make_shared to allocate the object and the shared_ptr all in one go, then the implementation can fold them together in a single allocation, as shown in Example 2(b) and Figure 2(b).

```
// Example 2(b): Single allocation
auto sp1 = make_shared<widget>();
auto sp2 = sp1;
```

Figure 2(b): Approximate memory layout for Example 2(b).

Note that combining the allocations has two major advantages:

- It reduces allocation overhead, including memory fragmentation. First, the most obvious way it does this is by reducing the number of allocation requests, which are typically more expensive operations. This also helps reduce contention on allocators (some allocators don't scale well). Second, using only one chunk of memory instead of two reduces the per-allocation overhead. Whenever you ask for a chunk of memory, the system must give you at least that many bytes, and often gives you a few more because of using fixed-size pools or tacking on housekeeping information per allocation. So by using a single chunk of memory, we tend to reduce the total extra overhead. Finally, we also naturally reduce the number of "dead" extra in-between gaps that cause fragmentation.
- It improves locality. The reference counts are frequently used with the object, and for small objects are likely to be on the same cache line, which improves cache performance (as long as

there isn't some thread copying the smart pointer in a tight loop; don't do that).

As always, when you can express more of what you're trying to achieve as a single function call, you're giving the system a better chance to figure out a way to do the job more efficiently. This is just as true when inserting 100 elements into a vector using a single range-insert call tov.insert(first, last) instead of 100 calls to v.insert(value) as it is when using a single call tomake_shared instead of separate calls to new widget() and shared_ptr(widget*).

There are two more advantages: Using make_shared avoids explicit new and avoids an exception safety issue. Both of these also apply to make_unique, so we'll cover them under #3.

3. WHY SHOULD YOU ALMOST ALWAYS USE MAKE_UNIQUE TO CREATE AN OBJECT TO BE INITIALLY OWNED BY A UNIQUE_PTR? EXPLAIN.

As with make_shared, there are two main cases where you can't use make_unique to create an object that you know will be owned (at least initially) by a unique_ptr: if you need a custom deleter, or if you are adopting a raw pointer.

Otherwise, which is nearly always, prefer make_unique.

Guideline: Use **make_unique** to create an object that isn't shared (at least not yet), unless you need a custom deleter or are adopting a raw pointer from elsewhere.

Besides symmetry with make_shared, make_unique offers at least two other advantages. First, you should prefer use make_unique<T>() instead of the more-verbose unique_ptr<T>{ new T{} }because you should avoid explicit new in general:

Guideline: Don't use explicit **new**, **delete**, and owning * pointers, except in rare cases encapsulated inside the implementation of a low-level data structure.

Second, it avoids some known exception safety issues with naked new. Here's an example:

```
void sink( unique_ptr<widget>, unique_ptr<gadget> );
```

```
sink( unique_ptr<widget>{new widget{}},
      unique_ptr<gadget>{new gadget{}} ); // Q1: do you see
the problem?
```

Briefly, if you allocate and construct the new widget first, then get an exception while allocating or constructing the new gadget, the widget is leaked. You might think: "Well, I could just change new widget{} to make_unique<widget>() and this problem would go away, right?" To wit:

```
sink( make_unique<widget>(),
      unique_ptr<gadget>{new gadget{}} );         // Q2: is
this better?
```

The answer is no, because C++ leaves the order of evaluation of function arguments unspecified and so either the new widget or the new gadget could be performed first. If thenew gadget is allocated and constructed first, then the make_unique<widget> throws, we have the same problem. But while just changing one of the arguments to use make_unique doesn't close the hole, changing them both to make_unique really does completely eliminate the problem:

```
sink( make_unique<widget>(), make_unique<gadget>() );  //
exception-safe
```

This exception safety issue is covered in more detail in GotW #56.

Guideline: To allocate an object, prefer to write **make_unique** by default, and write**make_shared** when you know the object's lifetime is going to be managed by using**shared_ptr**s.

4. WHAT'S THE DEAL WITH AUTO_PTR?

auto_ptr is most charitably characterized as a valiant attempt to create a unique_ptr before C++ had move semantics. auto_ptr is now deprecated, and should not be used in new code.
If you have auto_ptr in an existing code base, when you get a chance try doing a global search-and-replace of auto_ptr to unique_ptr; the vast majority of uses will work the same, and it might expose (as a compile-time error) or fix (silently) a bug or two you didn't know you had.
" http://herbsutter.com/2013/05/29/gotw-89-solution-smart-pointers/

POINTS ON POINTERS

"Sydius outlined the types fairly well:

- **Normal pointers** are just that - they point to some thing in memory somewhere. Who owns it? Only the comments will let you know. Who frees it? Hopefully the owner at some point.
- **Smart pointers** are a blanket term that cover many types; I'll assume you meant scoped pointer which uses the RAII pattern. It is a stack-allocated object that wraps a pointer; when it goes out of scope, it calls delete on the pointer it wraps. It "owns" the contained pointer in that it is in charge of deleteing it at some point. They allow you to get a raw reference to the pointer they wrap for passing to other methods, as well as RELEASING the pointer, allowing someone else to own it. Copying them does not make sense.
- **Shared pointers** is a stack-allocated object that wraps a pointer so that you don't have to know who owns it. When the last shared pointer for an object in memory is destructed, the wrapped pointer will also be deleted.

How about when you should use them? You will either make heavy use of scoped pointers or shared pointers. How many threads are running in your application? If the answer is "potentially a lot", shared pointers can turn out to be a performance bottleneck if used everywhere. The reason being that creating/copying/destructing a shared pointer needs to be an atomic operation, and this can hinder performance if you have many threads running. However, it won't always be the case - only testing will tell you for sure.

There is an argument (that I like) against shared pointers - by using them, you are allowing programmers to ignore who owns a pointer. This can lead to tricky situations with circular references (Java will detect these, but shared pointers cannot) or general programmer laziness in a large code base.

There are two reasons to use scoped pointers. The first is for simple exception safety and cleanup operations - if you want to guarantee that an object is cleaned up no matter what in the face of exceptions, and you don't want to stack allocate that object, put it in a scoped pointer. If the

operation is a success, you can feel free to transfer it over to a shared pointer, but in the meantime save the overhead with a scoped pointer.

The other case is when you want clear object ownership. Some teams prefer this, some do not. For instance, a data structure may return pointers to internal objects. Under a scoped pointer, it would return a raw pointer or reference that should be treated as a weak reference - it is an error to access that pointer after the data structure that owns it is destructed, and it is an error to delete it. Under a shared pointer, the owning object can't destruct the internal data it returned if someone still holds a handle on it - this could leave resources open for much longer than necessary, or much worse depending on the code.

“http://stackoverflow.com/questions/417481/pointers-smart-pointers-or-shared-pointers

LESSON #4: SMART POINTERS

“One big change to modern C++ style that comes with C++11 is that you should never need to manuallydelete (or free) anymore, thanks to the new classes shared_ptr, unique_ptr and weak_ptr.

Note that before C++11, C++ did have one smart pointer class – auto_ptr. This was unsafe and is now deprecated, with unique_ptr replacing it.

To use these classes, you'll need to #include <memory> (and also add using namespace std; or prefix withstd::).

unique_ptr

unique_ptr simply holds a pointer, and ensures that the pointer is deleted on destruction. unique_ptrobjects cannot be copied.

It thus behaves very much like the now-deprecated auto_ptr behaved – the problem with auto_ptr was that it was aiming to work what unique_ptr does, but unable to do so properly when it was defined, back before C++11 was invented with features like move-constructors, and thus it was unsafe.

As example of how unique_ptr is used is as follows – say we have the following:

```
struct MyClass {

  MyClass(const char* s);

  void methodA();

};

void someMethod(MyClass* m);


void test() {

  unique_ptr<MyClass> ptr1(new MyClass("obj1"));
```

```
  // can use -> (and *) on the unique_ptr just like with a normal pointer
  ptr1->methodA();

  // to get a plain pointer from the unique_ptr, we use the get() method
  someMethod(ptr1.get());

  // use std::move to transfer ownership to ptr2 - ptr1 now holds no pointer
  unique_ptr<MyClass> ptr2(std::move(ptr1));

  // assign a new pointer to ptr1
  ptr1.reset(new MyClass("obj2"));

  // assign a new pointer to ptr2 - "obj1" will now automatically be deleted
  ptr2.reset("obj3");

  // set ptr1 to contain nothing - "obj2" will now automatically be deleted
  ptr1.reset();

  // "obj3" will automatically deleted at the end of this function, as ptr2 goes out of scope
}
```

The following parts will describe three useful ways in which unique_ptr is often used …

unique_ptr for class members

A simple use of unique_ptr is for replacing the use of pointers in storing class members. For example, say you have the following class which simply stores a pointer to an int:

```
class A {
public:
  int* i;
```

```
  A():i(new int(0)) { }


  ~A() {
    if(i) {
      delete i;
    }
  }


private:
   // we need to explicitly disable value-copying, as it's not safe!
   A(const A&);
   A& operator=(const A&);
};
```

Using unique_ptr, this class simplifies to:

```
class B {
public:
  unique_ptr<int> i;


  B():i(new int(0)) { }
};
```

The destructor is no longer needed, and value-copying will automatically be disabled, as unique_ptr is not copyable. On top of that, this unique_ptr version also gets a move constructor defined automatically, thanks to unique_ptr being moveable (whereas the former would have to additionally define a move constructor to get that).

This code is not only more concise, but by eliminating the need to remember boilerplate code, it also eliminates two dangers which are common causes of memory leaks and memory corruption:

- forgetting to delete all class-member pointers in the destructor.
- forgetting to disable value-copying (or to define a suitable copy-constructor + operator= to handle it safely).

Making unique_ptr class members work with Forward Declarations

Say you're declaring *ClassA* to have a class member variable that is a pointer to *ClassB*, and you want to store that pointer using a unique_ptr object. And say you don't want to #include *ClassB.h* in *ClassA.h*(rather you want to forward declare *ClassB* there in order to save compilation time etc).

When you try to do this, you may sometimes encounter compilation problems, depending upon the compiler. In which case, to fix this you may need to ensure that the constructor and/or destructor are defined in the corresponding .cpp file, rather than being defined in the .h file or being not defined at all and thus left to default implementations.

Note that you would in any case generally have needed to do exactly this before when using a plain pointer and forward declaration. It's only now that using unique_ptr that you may have been able to avoid writing a destructor and just relied on the default constructor and/or destructor, as in the above example where the destructor was no longer needed).

This same caveat also applies to using shared_ptr for class member variables, although it again depends on the compiler.

unique_ptr for local variables within functions

A second example of the use of unique_ptr is for local variables within functions. Say you have the following function:

```
void methodA() {

  int* buf = new int[256];



  int result = fillBuf(buf))

  if(result == -1) {

    return;

  }

  printf("Result: %d", result);



  delete[] buf;

}
```

The above code has a couple of major safety issues:

- it is forgetting to delete buf when return is called, and thus will leak memory.
- if any exception is thrown by fillBuf(buf), it will again fail to delete `buf and thus leak memory.

A safer form of this function while still using a plain pointer would thus be something like:

```
void methodA() {
  int* buf = new int[256];

  try {
    int result = fillBuf(buf))
    if(result == -1) {
      delete[] buf;
      return;
    }
    printf("Result: %d", result);
  }
  catch(...) {
    delete[] buf;
    throw;
  }
  delete[] buf;
}
```

This should now be safe, however it's a lot of code to repeat and get right every time.

With unique_ptr, you avoid all this boilerplate – you just do the following, and the code is perfectly safe:

```
void methodA() {
  unique_ptr<int> buf(new int[256]);

  int result = fillBuf(buf))
  if(result == -1) {
    return;
  }
  printf("Result: %d", result);
}
```

unique_ptr in STL collections

When using STL collections in 'old C++', you have the choice of:

- storing objects by value, which can add a lot of overhead quickly if the objects are large and thus being copied around.
- storing objects with pointers, which suddenly adds a whole lot of memory management issues:
    - whenever an item is erased from the collection, the user has to remember to also the object it points to
    - when the collection itself is destroyed, the user has to first remember to iterate the entire collection in order to remove+erase any objects it still contains.

C++11 solves this above conundrum with move constructors (see my post on move semantics). These allow any object which defines cheap move constructors to be stored in collections by value rather than pointer, with little/no performance overhead as before (in fact potentially with performance gains due to less indirection).

unique_ptr also defines such move constructors, thus although it is not copyable, it is moveable, and so the collections still have a way to move it around – transferring the pointer that it points to betweenunique_ptr objects.

Thus for objects which don't define inexpensive move constructors, unique_ptr provides a safe way to store them in STL collections, with no difference in performance from storing plain pointers, i.e.:

```
std::vector<unique_ptr<MyClass>> v;


v.push_back(unique_ptr<MyClass>("hello world"));

v.emplace_back(new MyClass("hello world"));


MyClass* m = v.get();
```

This example also demonstrates the use of the new emplace_back in C++11, reducing a little verbosity otherwise attached with using unique_ptr to wrap the objects in the vector. emplace_back works likepush_back, but allows you to simply pass the arguments you would otherwise pass to the constructor for the object which you're storing in the vector, rather than passing the object itself. This method can also be more efficient than using the old push_back, so should be generally preferred.

shared_ptr

shared_ptr functions the same way as unique_ptr – holding a pointer, providing the same basic interface for construction and using the pointer, and ensuring that the pointer is deleted on destruction.

Unlike unique_ptr, it also allows copying of the shared_ptr object to another shared_ptr, and then ensures that the pointer is still guaranteed to always be deleted once (but not before) all shared_ptrobjects that were holding it are destroyed (or have released it).

It does this using reference counting – it keeps a shared count of how many shared_ptr objects are holding the same pointer. This reference counting is done using atomic functions and is thus threadsafe.

An example of its use is as follows:

```
struct MyClass {

  MyClass(const char* s);

  void methodA();

};

void someMethod(MyClass* m);


auto ptr = make_shared<MyClass>("obj1");


ptr->methodA();


someMethod(ptr.get());


shared_ptr<MyClass> anotherPtr = ptr; // now anotherPtr + ptr are both pointing to the "obj1" object


ptr.reset(new MyClass("obj2"); // now ptr switches to pointing to "obj2", but the "obj1"

                        // object is not deleted as anotherPtr is still holding it


anotherPtr.reset(); // now no shared_ptr object is referencing the "obj1" MyClass*, so it is deleted


// "obj2" will be automically deleted when ptr goes out of scope
```

Note the use of make_shared in the above – while it is possible to create a shared_ptr using the same syntax as for unique_ptr (passing a pointer to its constructor), make_shared should be always preferred, as it also happens to be more efficient (only requiring one memory allocation rather than two).

shared_ptr and thread-safety

One place where the use of shared_ptr can really shine (and the main case where you would need to use shared_ptr rather than unique_ptr is in multi-threaded applications. When you're not sure which thread will finish needing an object last, you can simply give each thread a shared_ptr that references the object.

However, note here that I said that you give *each thread* one such object. The shared_ptr class is *not*thread-safe for the case that two threads try to access the same shared_ptr object concurrently. Rather, thread-safety is ensured as long as each thread has their own shared_ptr objects (which may in turn all share+reference the same pointer).

weak_ptr

A weak_ptr is used to hold a non-owning reference to a pointer that is managed by a shared_ptr (or multiple shared_ptr objects). It must be converted to shared_ptr in order to access the referenced pointer.

weak_ptr models temporary ownership: when an object needs to be accessed only if it exists, and it may be deleted at any time by someone else, weak_ptr is used to track the object, and it is converted to std::shared_ptr to assume temporary ownership. If the original shared_ptr is destroyed at this time, the object's lifetime is extended until the temporary shared_ptr is destroyed as well.

weak_ptr is primarily useful in the rare cases where it is needed in order to break 'circular references' – a problem with the use of reference counting in shared_ptr. An example of this would be a doubly-linked-list containing two nodes – *Node A* and *Node B* – here *Node A* will contain a forward-reference to*Node B*, and *Node B* will contain a back-reference to *Node A*. If these references are shared_ptr objects, then their reference counts will never reach zero, as even once everything else that refers to *Node A*and *Node B* is destroyed, they will still be referencing each other.

```
auto sp = std::make_shared<string>("hello world");


std::weak_ptr<string> wp = sp;


if(auto spt = wp.lock()) { // to be safe, have to be copy into a shared_ptr before usage
  cout << *spt << endl;
}
else {
  cout << "sp was deleted already\n";
}
```

General Memory Management Guidelines – What to use when?

Method 1: just use shared_ptr everywhere

One simple approach is to use shared_ptr everywhere (together with weak_ptr in the rare case where you have cycles, but most programmers will never encounter this problem).

This has the advantage of being both safe and simple. And this is precisely the approach that the Objective-C language has taken with it's Automatic Reference Counting – in it's latest iteration it automatically adds reference counting to all objects, making this transparent so that the user is still just allocating and using

pointers, but internally these are being reference counted. It's also similar to the approach many other languages take with garbage collection.

The argument against this

There are however are a couple of issues with this approach:

- if your code is being used by other code (i.e. if you're creating a library), then if you're requiringshared_ptr pointers to be passed to your functions for example, you're imposing this (your) style of memory management on anybody using your code, which is both not neccessary and not appropriate.

- shared_ptr does incur a little overhead in terms of the atomic operations used in the reference counting, and in that it has to allocate a little extra memory to store the reference count (which is only really an issue if having say millions of pointers to very small objects).

- by consistently using unique_ptr, shared_ptr and plain-pointer where appropriate (i.e. the following guidelines), the ownership+memory-management+lifetime of objects is clear to anyone looking at code – at a class or method.

    - i.e. when you use shared_ptr everywhere, then looking at a method that takes a shared_ptr, you have no idea what that method will do with it – whether it will retain it or not.

    - such complex or unclear ownership models tend to lead to difficult to follow couplings of different parts of the application through shared state that may not be easily trackable.

- C++ gurus also seem to be strongly against this idea, i.e. Bjarne Stroustrop argues that "shared_ptr should be used as a last resort when an object's life-time is uncertain" (mainly it appears for the previous reason – making ownership+lifetime of resources clear)

A matter of personal preference perhaps

You might argue that resource lifetime+ownership is all stuff that one ideally shouldn't need to care about, now that you have shared_ptr and can just use it everywhere. You could also make similar arguments against the *Final guideline* (see below) of avoiding dynamic memory + pointers whenever possible, arguing that this also requires too much thinking about the performance tradeoffs and understanding of the implementations of the classes you're using – how much data is being copied by the objects' move constructors, and are you sure that they have move constructors defined? And that it's easer to simply not bother with defining move constructors correctly for all your classes, and cleaner and simpler just to use shared_ptr everywhere.

I tend to agree that simply using shared_ptr generally is a pretty safe way to do things for many kinds of projects (but I'd be interested to hear comments from those who disagree). I'd say especially for relative beginners, that you could do much worse than to just use shared_ptr everywhere. Also, basically all the other popular programming languages out there are now happily using either automatic reference counting or garbage collection these days, so using C++ effectively in the same way – by using shared_ptr everywhere – can't be such a horrible idea.

Using shared_ptr everywhere will also hurt C++'s performance a little. And I guess people could argue that if you don't care about performance so much, and don't want to think about object lifecycle and such, then there's many other languages out there for you to use (in fact, pretty much every other language out there).

I don't totally buy this argument either though – using shared_ptr everywhere is really not a significant performance cost most of the time (nowhere near as bad as switching to Ruby for example), and I'd like to see C++ as a good tool for many kinds of programming needs, not only a tool to be used by programmers that like/need to invest extra time in understanding and thinking more about memory managment and object ownership+lifetimes in order to squeeze final couple of % of performance out of their programs.

- note also that you can avoid unnecessary performance penalties from using shared_ptr by minimizing copying of the shared_ptr objects (as that incurs atomic increments and decrements which are the main overhead)

Method 2: some general guidelines

These general guidelines are a little more complicated than above method of just using shared_ptr. They require some thinking about the concept of 'ownership' of your objects – who is the owner of this object, and is there one owner or many? Generally there should just be one owner, however in some cases ownership will need to be shared (and thus in these cases, but only in these cases,shared_ptr is to used). Plain pointers, when used, should always be perceived as non-owning pointers that you must guarantee never outlive the object they points to.

The following guidelines apply:

- when an object is allocated, it should generally be immediately assigned to a shared_ptr orunique_ptr that is the 'owner':
    - typically this shared_ptr/unique_ptr should act as the 'owner' of the object
    - plain pointers are thus never used to act as the 'owners' of objects
    - the exception to this immediate assignment rule is things like factory methods that return a plain pointer to the object they create – in this case however, the callee still should generally be immediately assigning this returned object to a shared_ptr / unique_ptr
    - also, note here that some argue that even in such cases, the object should still be immediately assigned to a unique_ptr, which these methods should return, and the object then gets transferred (moved) from this unique_ptr to another unique_ptr / shared_ptr for ownership – this all being done in order to ensure that the object is at all times 'owned' by a smart pointer so that there is no risk of memory leaks in the case of an exception being thrown etc
- methods should take plain-pointers in their arguments, unless they plan to store a reference to a passed object beyond the life of the method (or pass it to another function that does this and is thus also requiring a shared_ptr), in which case they should take shared_ptr.
- methods should always return plain-pointers when it is up to the caller to handle memory management of the object, i.e. to assume ownership of it with a shared_ptr/unique_ptr or to pass to someone else to own it. Examples of such methods would be:
    - factory methods to create an instance of some class
    - as mentioned earlier, some people argue that unique_ptr should be used in this case, in order that there is always an owner (in case some exception is thrown at any point and thus there's risk of a memory leak)

- methods that are accessor methods to an owned object that is a unique_ptr should return a plain pointer. Examples of such methods would be:
    - a singleton getInstance() method.
- methods that are accessor methods to an owned object that is a shared_ptr should generally return a shared_ptr object (though this should be a rare case I would think).
- use unique_ptr whenever shared_ptr is not neccessary, that is, whenever shared-ownership is not neccessary. It should relatively rarely be that case that shared-ownership is really needed, but cases where it really can be needed include
    - when shared-ownership is required due to sharing an object between multiple threads.
    - for modeling parent-child relations using shared_ptr together with weak_ptr

Final guideline – simply avoid dynamic memory + pointers whenever possible

With C++11 it is generally preferable to simply avoid dynamic allocation, thus avoiding the consequent need for any corresponding form of memory management.

With the new move constructors in C++11 (see my post on move semantics), most objects' classes should have move constructors defined, and thus objects should generally be moveable cheaply (for example STL classes should now all be), thus such objects:

- can be stored in containers by value without the need to first wrap them in pointers or smart-pointers to avoid costly copies. Thus for example, instead of vector<unique_ptr<string>> orvector<shared_ptr<string>> or vector<string*>, the simple vector<string> should generally be preferred now.
- can generally passed to functions by value (rather than by reference or as pointers), without any extra cost
- can generally be returned by functions by value (when returning a temporary at least) without any extra cost

Discussion

I would love to hear all opinions about whether there's any good arguement against someone preferring to simply use smart_ptr everywhere – it is not what I see the experts recommending, and it is not something I use in my work as it generally involves extremely performance critical code, however it would seem like a relatively safe braindead approach, giving the programmer less to worry about for relatively little performance cost.

Also also any discussion about the 'general guidelines' on when to use unique_ptr vs plain pointer vsshared_ptr is welcome.

” http://mbevin.wordpress.com/2012/11/18/smart-pointers/

POINTS ON RAW POINTERS

"Its declaration doesn't indicate whether it points to a single object or to an array.

2. Its declaration reveals nothing about whether you should destroy what it points to when you're done using it, i.e., if the pointer *owns* the thing it points to.

3. If you determine that you should destroy what the pointer points to, there's no way to tell how. Should you use delete, or is there a different destruction mechanism (e.g., a dedicated destruction function the pointer should be passed to)?

4. If you manage to find out that delete is the way to go, Reason 1 means it may not be possible to know whether to use the single-object form ("delete") or the array form ("delete []"). If you use the wrong form, results are undefined.

5. Assuming you ascertain that the pointer owns what it points to and you discover how to destroy it, it's difficult to ensure that you perform the destruction *exactly once* along every path in your code (including those due to exceptions). Missing a path leads to resource leaks, and doing the destruction more than once leads to undefined behavior.

6. There's typically no way to tell if the pointer dangles, i.e., points to memory that no longer holds the object the pointer is supposed to point to. Dangling pointers arise when objects are destroyed while pointers still point to them.

Raw pointers are powerful tools, to be sure, but decades of experience have demonstrated that with only the slightest lapse in concentration or discipline, these tools can turn on their ostensible masters.

Smart pointers are one way to address these issues. Smart pointers are wrappers around raw pointers that act much like the raw pointers they wrap, but that avoid many of their pitfalls. You should therefore prefer smart pointers to raw pointers. Smart pointers can do virtually everything raw pointers can, but with far fewer opportunities for error.

There are four smart pointers in C++11: std::auto_ptr, std::unique_ptr,

std::shared_ptr, and std::weak_ptr. All are designed to help manage the lifetimes

of dynamically allocated objects, i.e., to avoid resource leaks by ensuring that

such objects are destroyed in the appropriate manner at the appropriate time (including

in the event of exceptions).

" (Meyers, Effective Modern C++, 2014)

USE **STD::UNIQUE_PTR** FOR EXCLUSIVE-OWNERSHIPRESOURCE MANAGEMENT.

"When you reach for a smart pointer, std::unique_ptr should generally be the one closest at hand. It's reasonable to assume that, by default, std::unique_ptrs are the same size as raw pointers, and for most operations (including dereferencing), they execute exactly the same instructions. This means you can use them even in situa-tions where memory and cycles are tight. If a raw pointer is small enough and fast enough for you, a std::unique_ptr almost certainly is, too.

std::unique_ptr embodies *exclusive ownership* semantics. A non-null std::unique_ptr always owns what it points to. Moving a std::unique_ptr transfers ownership from the source pointer to the destination pointer. (The source pointer is set to null.) Copying a std::unique_ptr isn't allowed, because if you could copy a std::unique_ptr, you'd end up with two std::unique_ptrs to the same resource, each thinking it owned (and should therefore destroy) that resource. std::unique_ptr is thus a *move-only type*. Upon destruction, a non-null std::unique_ptr destroys its resource. By default, resource destruction is accomplished by applying delete to the raw pointer inside the std::unique_ptr."

USE **STD::SHARED_PTR** FOR SHARED-OWNERSHIPRESOURCE MANAGEMENT

"std::shared_ptr is the C++11 way of binding these worlds together. An object

accessed via std::shared_ptrs has its lifetime managed by those pointers through

shared ownership. No specific std::shared_ptr owns the object. Instead, all

std::shared_ptrs pointing to it collaborate to ensure its destruction at the point

where it's no longer needed. When the last std::shared_ptr pointing to an object

stops pointing there (e.g., because the std::shared_ptr is destroyed or made to

point to a different object), that std::shared_ptr destroys the object it points to. As

with garbage collection, clients need not concern themselves with managing the lifetime

of pointed-to objects, but as with destructors, the timing of the objects' destruction

is deterministic.

A std::shared_ptr can tell whether it's the last one pointing to a resource by consulting the resource's *reference count*, a value associated with the resource that keeps track of how many std::shared_ptrs point to it. std::shared_ptr constructors increment this count (usually—see below), std::shared_ptr destructors decrement it, and copy assignment operators do both. (If sp1 and sp2 are std::shared_ptrs to different objects, the assignment "sp1 = sp2;" modifies sp1 such that it points to the object pointed to by sp2. The net effect of the assignment is that the reference count for the object originally pointed to by sp1 is decremented, while that for the object pointed to by sp2 is incremented.) If a std::shared_ptr sees a reference count of zero after performing a decrement, no more std::shared_ptrs point to the resource, so the std::shared_ptr destroys it.

The existence of the reference count has performance implications:

std::shared_ptrs are twice the size of a raw pointer, because they internally contain a raw pointer to the resource as well as a raw pointer to the resource's reference count.2

· **Memory for the reference count must be dynamically allocated**. Conceptually, the reference count is associated with the object being pointed to, but pointed-to objects know nothing about this. They thus have no place to store a reference count. (A pleasant implication is that any object—even those of built-in types—may be managed by std::shared_ptrs.) Item 21 explains that the cost of the dynamic allocation is avoided when the std::shared_ptr is created by std::make_shared, but there are situations where std::make_shared can't be used. Either way, the reference count is stored as dynamically allocated data.

· **Increments and decrements of the reference count must be atomic**, because there can be simultaneous readers and writers in different threads. For example,

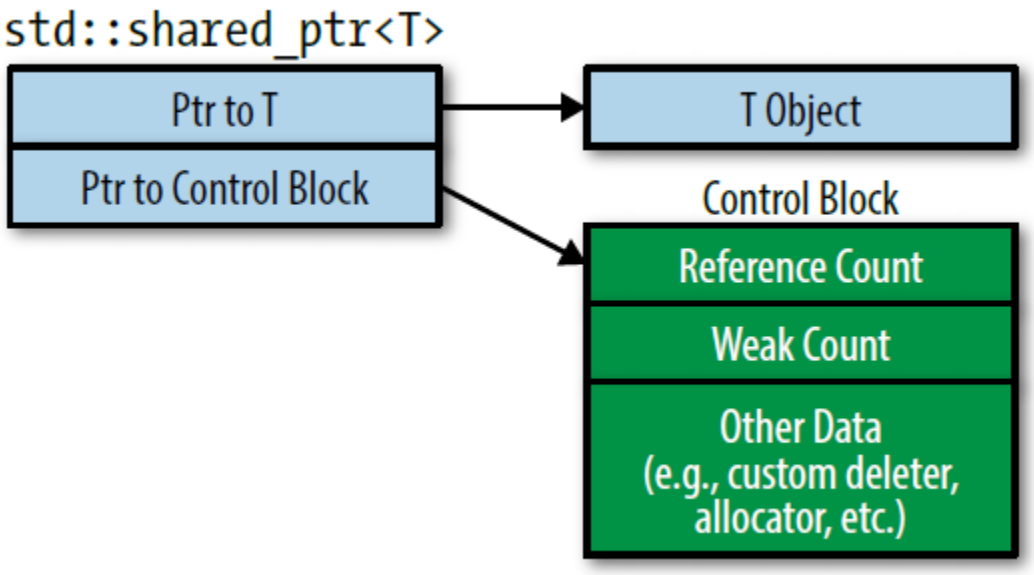
a std::shared_ptr pointing to a resource in one thread could be executing its destructor (hence decrementing the reference count for the resource it points to), while, in a different thread, a std::shared_ptr to the same object could be copied (and therefore incrementing the same reference count). Atomic operations are typically slower than non-atomic operations, so even though reference counts are usually only a word in size, you should assume that reading and writing them is comparatively costly.

Did I pique your curiosity when I wrote that std::shared_ptr constructors only "usually" increment the reference count for the object they point to? Creating a std::shared_ptr pointing to an object always yields one more std::shared_ptr pointing to that object, so why mustn't we *always* increment the reference count? Move construction, that's why. Move-constructing a std::shared_ptr from another std::shared_ptr sets the source std::shared_ptr to null, and that means that the old std::shared_ptr stops pointing to the resource at the moment the new std::shared_ptr starts. As a result, no reference count manipulation is required. Moving std::shared_ptrs is therefore faster than copying them: copying requires incrementing the reference count, but moving doesn't. This is as true for assignment as for construction, so move construction is faster than copy construction, and move assignment is faster than copy assignment.

Like std::unique_ptr (see Item 18), std::shared_ptr uses delete as its default resource-destruction mechanism, but it also supports custom deleters.
The design of this support differs from that for std::unique_ptr.
In another difference from std::unique_ptr, specifying a custom deleter doesn't change the size of a std::shared_ptr object. Regardless of deleter, a std::shared_ptr object is two pointers in size. That's great news, but it should

make you vaguely uneasy. Custom deleters can be function objects, and function objects can contain arbitrary amounts of data. That means they can be arbitrarily large. How can a std::shared_ptr refer to a deleter of arbitrary size without using any more memory?

It can't. It may have to use more memory. However, that memory isn't part of the std::shared_ptr object. It's on the heap or, if the creator of the std::shared_ptr took advantage of std::shared_ptr support for custom allocators, it's wherever the memory managed by the allocator is located. I remarked earlier that a std::shared_ptr object contains a pointer to the reference count for the object it points to. That's true, but it's a bit misleading, because the reference count is part of a larger data structure known as the *control block*. There's a control block for each object managed by std::shared_ptrs. The control block contains, in addition to the reference count, a copy of the custom deleter, if one has been specified. If a custom allocator was specified, the control block contains a copy of that, too. The control block may also contain additional data, including, as Item 21 explains, a secondary reference count known as the weak count, but we'll ignore such data in this Item. We can envision the memory associated with a std::shared_ptr<T> object as looking like this:



An object's control block is set up by the function creating the first std::shared_ptr to the object. At least that's what's supposed to happen. In general, it's impossible for a function creating a std::shared_ptr to an object to know whether some other std::shared_ptr already points to that object, so the following rules for control block creation are used:

· **std::make_shared (see Item 21) always creates a control block.** It manufactures a new object to point to, so there is certainly no control block for that object at the time std::make_shared is called.

· **A control block is created when a std::shared_ptr is constructed from a unique-ownership pointer (i.e., a std::unique_ptr or std::auto_ptr).** Unique-ownership pointers don't use control blocks, so there should be no control block for the pointed-to object. (As part of its construction, the std::shared_ptr assumes ownership of the pointed-to object, so the uniqueownership pointer is set to null.)

· **When a std::shared_ptr constructor is called with a raw pointer, it creates a control block.** If you wanted to create a std::shared_ptr from an object that already had a control block, you'd presumably pass a std::shared_ptr or a

std::weak_ptr (see Item 20) as a constructor argument, not a raw pointer.

std::shared_ptr constructors taking std::shared_ptrs or std::weak_ptrs as constructor arguments don't create new control blocks, because they can rely on the smart pointers passed to them to point to any necessary control blocks.

A consequence of these rules is that constructing more than one std::shared_ptr from a single raw pointer gives you a complimentary ride on the particle accelerator of undefined behavior, because the pointed-to object will have multiple control blocks. Multiple control blocks means multiple reference counts, and multiple reference counts means the object will be destroyed multiple times (once for each reference count). That means that code like this is bad, bad, bad:

```
auto pw = new Widget; // pw is raw ptr
…
std::shared_ptr<Widget> spw1(pw, loggingDel); // create control
// block for *pw
…
std::shared_ptr<Widget> spw2(pw, loggingDel); // create 2nd
// control block
// for *pw!
```

There are at least two lessons regarding std::shared_ptr use here. First, try to avoid passing raw pointers to a std::shared_ptr constructor. The usual alternative is to use std::make_shared (see Item 21), but in the example above, we're using custom deleters, and that's not possible with std::make_shared. Second, if you must pass a raw pointer to a std::shared_ptr constructor, pass the result of new directly instead of going through a raw pointer variable.

A control block is typically only a few words in size, although custom deleters and allocators may make it larger. The usual control block implementation is more sophisticated than you might expect. It makes use of inheritance, and there's even a virtual function. (It's used to ensure that the pointed-to object is properly destroyed.)

That means that using std::shared_ptrs also incurs the cost of the machinery for the virtual function used by the control block.

Under typical conditions, where the default deleter and default allocator are used and where the std::shared_ptr is created by std::make_shared, the control block is only about three words in size, and its allocation is essentially free. (It's incorporated into the memory allocation for the object being pointed to. For details, see Item 21.)

Dereferencing a std::shared_ptr is no more expensive than dereferencing a raw pointer. Performing an operation requiring a reference count manipulation (e.g., copy construction or copy assignment, destruction) entails one or two atomic operations, but these operations typically map to individual machine instructions, so although they may be expensive compared to non-atomic instructions, they're still just single instructions. The virtual function machinery in the control block is generally used only once per object managed by std::shared_ptrs: when the object is destroyed.

In exchange for these rather modest costs, you get automatic lifetime management of dynamically allocated resources. Most of the time, using std::shared_ptr is vastly preferable to trying to manage the lifetime of an object with shared ownership by hand. If you find yourself doubting whether you can afford use of std::shared_ptr, reconsider whether you really need shared ownership. If exclusive ownership will do or even *may* do, std::unique_ptr is a better choice. Its performance profile is close to that for raw pointers, and "upgrading" from std::unique_ptr to std::shared_ptr is easy, because a std::shared_ptr can be created from a std::unique_ptr.

The reverse is not true. Once you've turned lifetime management of a resource over to a std::shared_ptr, there's no changing your mind. Even if the reference count is one, you can't reclaim ownership of the resource in order to, say, have a

std::unique_ptr manage it. The ownership contract between a resource and the std::shared_ptrs that point to it is of the 'til-death-do-us-part variety. No divorce, no annulment, no dispensations."

USE **STD::WEAK_PTR** FOR **STD::SHARED_PTR**LIKEPOINTERS THAT CAN DANGLE

"Paradoxically, it can be convenient to have a smart pointer that acts like a std::shared_ptr (see Item 19), but that doesn't participate in the shared ownership of the pointed-to resource. In other words, a pointer like std::shared_ptr that doesn't affect an object's reference count. This kind of smart pointer has to contend with a problem unknown to std::shared_ptrs: the possibility that what it points to has been destroyed. A truly smart pointer would deal with this problem by tracking when it *dangles*, i.e., when the object it is supposed to point to no longer exists. That's precisely the kind of smart pointer std::weak_ptr is.

You may be wondering how a std::weak_ptr could be useful. You'll probably wonder even more when you examine the std::weak_ptr API. It looks anything but smart. std::weak_ptrs can't be dereferenced, nor can they be tested for nullness. That's because std::weak_ptr isn't a standalone smart pointer. It's an augmentation of std::shared_ptr."

PREFER STD::MAKE_UNIQUE AND STD::MAKE_SHARED TO DIRECT USE OF NEW

"Let's begin by leveling the playing field for std::make_unique and std::make_shared. std::make_shared is part of C++11, but, sadly, std::make_unique isn't. It joined the Standard Library as of C++14. If you're using C++11, never fear, because a basic version of std::make_unique is easy to write yourself. Here, look:

```
template<typename T, typename... Ts>
std::unique_ptr<T> make_unique(Ts&&... params)
```

```
{
return std::unique_ptr<T>(new T(std::forward<Ts>(params)...));
}
```

As you can see, make_unique just perfect-forwards its parameters to the constructor of the object being created, constructs a std::unique_ptr from the raw pointer new produces, and returns the std::unique_ptr so created. This form of the function doesn't support arrays or custom deleters (see Item 18), but it demonstrates that with only a little effort, you can create make_unique if you need to.3 Just remember not to put your version in namespace std, because you won't want it to clash with a vendorprovided version when you upgrade to a C++14 Standard Library implementation.

std::make_unique and std::make_shared are two of the three *make functions*: functions that take an arbitrary set of arguments, perfect-forward them to the constructor for a dynamically allocated object, and return a smart pointer to that object.

The third make function is std::allocate_shared. It acts just like std::make_shared, except its first argument is an allocator object to be used for the dynamic memory allocation.

Even the most trivial comparison of smart pointer creation using and not using a make function reveals the first reason why using such functions is preferable. Consider:

```
auto upw1(std::make_unique<Widget>()); // with make func
std::unique_ptr<Widget> upw2(new Widget); // without make func
auto spw1(std::make_shared<Widget>()); // with make func
std::shared_ptr<Widget> spw2(new Widget); // without make func
```

I've highlighted the essential difference: the versions using new repeat the type being created, but the make functions don't. Repeating types runs afoul of a key tenet of software engineering: code duplication should be avoided.

That's because std::make_shared allocates a single chunk of memory to hold both the Widget object and the control block. This optimization reduces the static size of the program, because the code contains only one memory allocation call, and it increases the speed of the executable code, because memory is allocated only once.

The arguments for preferring make functions over direct use of new are strong ones. Despite their software engineering, exception safety, and efficiency advantages, however, this Item's guidance is to *prefer* the make functions, not to rely on them exclusively. That's because there are circumstances where they can't or shouldn't be used.

The size and speed advantages of std::make_shared vis-a-vis direct use of new stem from std::shared_ptr's control block being placed in the same chunk of memory as the managed object. When that object's reference count goes to zero, the object is destroyed (i.e., its destructor is called). However, the memory it occupies can't be released until the control block has also been destroyed, because the same chunk of dynamically allocated memory contains both.

As I noted, the control block contains bookkeeping information beyond just the reference count itself. The reference count tracks how many std::shared_ptrs refer to the control block, but the control block contains a second reference count, one that tallies how many std::weak_ptrs refer to the control block. This second reference count is known as the *weak count*.4 When a std::weak_ptr checks to see if it has expired (see Item 19), it does so by examining the reference count (not the weak count) in the control block that it refers to. If the reference count is zero (i.e., if the pointed-to object has no std::shared_ptrs referring to it and has thus been destroyed), the std::weak_ptr has expired. Otherwise, it hasn't."

WHEN USING THE PIMPL IDIOM, DEFINE SPECIALMEMBER FUNCTIONS IN THE IMPLEMENTATION FILE

"If you've ever had to combat excessive build times, you're familiar with the *Pimpl* ("pointer to implementation") *Idiom*. That's the technique whereby you replace the

data members of a class with a pointer to an implementation class (or struct), put the data members that used to be in the primary class into the implementation class, and

access those data members indirectly through the pointer. For example, suppose

Widget looks like this:

class Widget { // in header "widget.h"

public:

Widget();

…

private:

std::string name;

std::vector<double> data;

Gadget g1, g2, g3; // Gadget is some user-

};"

RVALUE REFERENCES, MOVE SEMANTICS AND PERFECT FORWARDING

- **Understand std::move and std::forward**
  - Std::move performs an unconditional cast to an rvalue. In and of itself, it does not move anything.
  - Std::forward casts its argument to an rvalue only if that argument is bound to an rvalue.
  - Neither std:move nor std::forward do anything at run time
- **Distinguish universal references from rvalue references**
  - If a function template parameter has type T&& for a deduced type T, or if an object is declared using auto&&, the parameter or object is a universal reference.
  - If the form of the type declaration isn't precisely *type*&&, or if type deduction does not occur, *type*&& denotes an rvalue reference.
  - Universal references correspond to rvalue references if they are initialized with rvalues. They correspond to lvalue references if they are initialized with lvalues.
- **Use std::move on rvalue references, std::forward on universal references**
  - Apply std::move to rvalue references and std::forward to universal references the last time each is used.
  - Do the same thing for rvalue references and universal references being returned from functions that return by value.
  - Never apply the std:move or std::forward to local objects if they would otherwise be eligible for the return optimization.
- **Avoid overloading on universal references**
  - Overloading on universal references almost always leads to the universal reference overload being called more frequently than expected.

    - Perfect-forwarding constructors are especially problematic, because they're typically better matches than copy constructors for non-const lvalues, and they can hijack derived class calls to base class copy and move constructors.
- **Familiarize yourself with alternatives to overloading on universal references**
    - Alternatives to the combination of universal references and overloading include the use of distinct function names, passing parameters by (lvalue)- reference-to-const, passing parameters by value, and using tag dispatch.
    - Constraining templates via std::enable_if permits the use of universal references and overloading together, but it controls the conditions under which compilers may use the universal reference overloads.
    - Universal reference parameters often have efficiency advantades, but they typically have usability disadvantages.
- **Understand reference collapsing**
    - Reference collapsing occurs in four contexts: template instantiation, auto type generation, creation and use of typedefs and alias declarations, and decltype.
    - When compilers generate a reference to a reference in a reference collapsing context, the result becomes a single reference. If either of the original references is an lvalue reference, the result is an lvalue reference. Otherwise it's an rvalue reference.
    - Universal references are rvalue references in contexts where type deduction distinguishes lvalues from rvalues and where references-collapsing occurs.
- **Familiarize yourself with perfect forwarding failure cases**
    - Perfect forwarding fails when template type deduction fails or when it deduces the wrong type.
    - The kinds of arguments that lead to perfect forwarding failure are braced initializers, null pointers expressed as 0 or NULL, declaration-only integral const static data members, template and overloaded function names, and bitfields.

"When you first learn about them, move semantics and perfect forwarding seem pretty straightforward:

· **Move semantics** makes it possible for compilers to replace expensive copying operations with less expensive moves. In the same way that copy constructors and copy assignment operators give you control over what it means to copy objects, move constructors and move assignment operators offer control over the semantics of moving. Move semantics also enables the creation of move-only types, such as std::unique_ptr, std::future, and std::thread.

· **Perfect forwarding** makes it possible to write function templates that take arbitrary arguments and forward them to other functions such that the target functions receive exactly the same arguments as were passed to the forwarding

functions.

Rvalue references are the glue that ties these two rather disparate features together. They're the underlying language mechanism that makes both move semantics and perfect forwarding possible.

The more experience you have with these features, the more you realize that your initial impression was based on only the metaphorical tip of the proverbial iceberg. The world of move semantics, perfect forwarding, and rvalue references is more nuanced than it appears. std::move doesn't move anything, for example, and perfect forwarding is imperfect. Move operations aren't always cheaper than copying; when they are, they're not always as cheap as you'd expect; and they're not always called in a context where moving is valid. The construct "*type*&&" doesn't always represent an rvalue reference."

C++11 TUTORIAL: INTRODUCING THE MOVE CONSTRUCTOR AND THE MOVE ASSIGNMENT OPERATOR

"COPY CONSTRUCTORS SOUNDS LIKE A TOPIC FOR AN ARTICLE FROM 1989. AND YET, THE CHANGES IN THE NEW C++ STANDARD AFFECT THE DESIGN OF A CLASS' SPECIAL MEMBER FUNCTIONS FUNDAMENTALLY. FIND OUT MORE ABOUT THE IMPACT OF MOVE SEMANTICS ON OBJECTS' BEHAVIOR AND LEARN HOW TO IMPLEMENT THE MOVE CONSTRUCTOR AND THE MOVE ASSIGNMENT OPERATOR IN C++11.

C++11 is the informal name for ISO/IEC 14882:2011, the new C++ standard that was published in September 2011. It includes the TR1 libraries and a large number of new core features (a detailed discussion about these new C++11 features is available here; also see The Biggest Changes in C++11 (and Why You Should Care)):

- Initializer lists
- Uniform initialization notation
- Lambda functions and expressions

- Strongly-typed enumerations
- Automatic type deduction in declarations
- `__thread_local` storage class
- Control and query of object alignment
- Static assertions
- Type `long long`
- Variadic templates

Important as these features may be, the defining feature of C++11 is RVALUE REFERENCES.

THE RIGHT TIME FOR RVALUE REFERENCES

Rvalue references are a new category of reference variables that can bind to RVALUES. Rvalues are slippery entities, such as temporaries and literal values; up until now, you haven't been able to bind these safely to reference variables.

Technically, an rvalue is an unnamed value that exists only during the evaluation of an expression. For example, the following expression produces an rvalue:

```
x+(y*z); // A C++ expression that produces a temporary
```

C++ creates a temporary (an rvalue) that stores the result of `y*z`, and then adds it to `x`. Conceptually, this rvalue evaporates by the time you reach the semicolon at the end of the full expression.

A declaration of an rvalue reference looks like this:

```
std::string&& rrstr; //C++11 rvalue reference variable
```

The traditional reference variables of C++ e.g.,

```
std::string& ref;
```

are now called LVALUE REFERENCES.

Rvalue references occur almost anywhere, even if you don't use them directly in your code. They affect the semantics and lifetime of objects in C++11. To see how exactly, it's time to talk about MOVE SEMANTICS.

GET TO KNOW MOVE SEMANTICS

Hitherto, copying has been the only means for transferring a state from one object to another (an object's state is the collective set of its non-static data members' values). Formally, copying causes a target object `t` to end up with the same state as the source `s`, without modifying `s`. For example, when you copy a string `s1` to `s2`, the result is two identical strings with the same state as `s1`.

And yet, in many real-world scenarios, you don't copy objects but MOVEthem. When my landlord cashes my rent check, he moves money from my account into his. Similarly, removing the SIM card from your mobile phone and installing it in another mobile is a move operation, and so are cutting-and-pasting icons on your desktop, or borrowing a book from a library.

Notwithstanding the conceptual difference between copying and moving, there's a practical difference too: Move operations tend to be faster than copying because they transfer an existing resource to a new destination, whereas copying requires the creation of a new resource from scratch. The efficiency of moving can be witnessed among the rest in functions that return objects by value. Consider:

```
string func()
{
string s;
//do something with s
return s;
}
string mystr=func();
```

When `func()` returns, C++ constructs a temporary copy of `s` on the caller's stack memory. Next, `s` is destroyed and the temporary is used for copy-constructing `mystr`.

After that, the temporary itself is destroyed. Moving achieves the same effect without so many copies and destructor calls along the way.

Moving a string is almost free; it merely assigns the values of the source's data members to the corresponding data members of the target. In contrast, copying a string requires the allocation of dynamic memory and copying the characters from the source.

MOVE SPECIAL MEMBER FUNCTIONS

C++11 introduces two new special member functions: the MOVE CONSTRUCTORand the MOVE ASSIGNMENT OPERATOR. They are an addition to the fabulous four you know so well:

- Default constructor
- Copy constructor
- Copy assignment operator
- Destructor

If a class doesn't have any user-declared special member functions (save a default constructor), C++ declares its remaining five (or six) special member functions implicitly, including a move constructor and a move assignment operator. For example, the following class

```
class S{};
```

doesn't have any user-declared special member functions. Therefore, C++ declares all of its six special member functions implicitly. Under certain conditions, implicitly declared special member functions become implicitly defined as well. The implicitly-defined move special member functions move their sub-objects and data members in a member-wise fashion. Thus, a move constructor invokes its sub-objects' move constructors, recursively. Similarly, a move assignment operator invokes its sub-objects' move assignment operators, recursively.

What happens to a moved-from object? The state of a moved-from object is unspecified. Therefore, always assume that a moved-from object no longer owns any

resources, and that its state is similar to that of an empty (as if default-constructed) object. For example, if you move a string `s1` to `s2`, after the move operation the state of `s2` is identical to that of `s1` before the move, whereas `s1` is now an empty (though valid) string object.

DESIGNING A MOVE CONSTRUCTOR

A move constructor looks like this:

```
C::C(C&& other); //C++11 move constructor
```

It doesn't allocate new resources. Instead, it PILFERS `other`'s resources and then sets `other` to its default-constructed state.

Let's look at a concrete example. Suppose you're designing a `MemoryPage`class that represents a memory buffer:

```
class MemoryPage
{
size_t size;
char * buf;
public:
explicit MemoryPage(int sz=512):
size(sz), buf(new char [size]) {}
~MemoryPage( delete[] buf;}
//typical C++03 copy ctor and assignment operator
MemoryPage(const MemoryPage&);
MemoryPage& operator=(const MemoryPage&);
};
```

A typical move constructor definition would look like this:

```
//C++11
MemoryPage(MemoryPage&& other): size(0), buf(nullptr)
```

```
{
// pilfer other's resource
size=other.size;
buf=other.buf;
// reset other
other.size=0;
other.buf=nullptr;
}
```

The move constructor is much faster than a copy constructor because it doesn't allocate memory nor does it copy memory buffers.

DESIGNING A MOVE ASSIGNMENT OPERATOR

A move assignment operator has the following signature:

```
C& C::operator=(C&& other);//C++11 move assignment operator
```

A move assignment operator is similar to a copy constructor except that before pilfering the source object, it releases any resources that its object may own. The move assignment operator performs four logical steps:

- Release any resources that `*this` currently owns.
- Pilfer `other`'s resource.
- Set `other` to a default state.
- Return `*this`.

Here's a definition of `MemoryPage`'s move assignment operator:

```
//C++11
MemoryPage& MemoryPage::operator=(MemoryPage&& other)
{
if (this!=&other)
{
// release the current object's resources
```

```
delete[] buf;
size=0;
// pilfer other's resource
size=other.size;
buf=other.buf;
// reset other
other.size=0;
other.buf=nullptr;
}
return *this;
}
```

OVERLOADING FUNCTIONS

The overload resolution rules of C++11 were modified to support rvalue references. Standard Library functions such as `vector::push_back()` now define two overloaded versions: one that takes `const T&` for lvalue arguments as before, and a new one that takes a parameter of type `T&&` for rvalue arguments. The following program populates a vector with`MemoryPage` objects using two `push_back`() calls:

```
#include <vector>
using namespace std;
int main()
{
vector<MemoryPage> vm;
vm.push_back(MemoryPage(1024));
vm.push_back(MemoryPage(2048));
}
```

Both `push_back()` calls resolve as `push_back(T&&)` because their arguments are rvalues. `push_back(T&&)` moves the resources from the argument into `vector`'s internal `MemoryPage` objects using `MemoryPage`'s move constructor. In older versions of C++, the same program WOULDgenerate copies of the argument since the copy constructor of `MemoryPage`would be called instead.

As I said earlier, `push_back(const T&)` is called when the argument is an lvalue:

```
#include <vector>
using namespace std;
int main()
{
vector<MemoryPage> vm;
MemoryPage mp1(1024);//lvalue
vm.push_back(mp); //push_back(const T&)
}
```

However, you can enforce the selection of `push_back(T&&)` even in this case by casting an lvalue to an rvalue reference using `static_cast`:

```
//calls push_back(T&&)
```

vm.push_back(static_cast<MemoryPage&&>(mp));

Alternatively, use the new standard function `std::move()` for the same purpose:

```
vm.push_back(std::move(mp));//calls push_back(T&&)
```

It may seem as if `push_back(T&&)` is always the best choice because it eliminates unnecessary copies. However, remember that `push_back(T&&)` empties its argument. If you want the argument to retain its state after a `push_back()` call, stick to copy semantics. Generally speaking, don't rush to throw away the copy constructor and the copy assignment operator. In some cases, the same class could be used in a context that requires pure copy semantics, whereas in other contexts move semantics would be preferable.

IN CONCLUSION

C++11 is a different and better C++. Its rvalue references and move-oriented Standard Library eliminate many unnecessary copy operations, thereby improving performance

significantly, with minimal, if any, code changes. The move constructor and the move assignment operator are the vehicles of move operations. It takes a while to internalize the principles of move semantics – and to design classes accordingly. However, the benefits are substantial. I would dare predicting that other programming languages will soon find ways to usher-in move semantics too.

Danny Kalev IS A CERTIFIED SYSTEM ANALYST BY THE ISRAELI CHAMBER OF SYSTEM ANALYSTS AND SOFTWARE ENGINEER SPECIALIZING IN C++. KALEV HAS WRITTEN SEVERAL C++ TEXTBOOKS AND CONTRIBUTES C++ CONTENT REGULARLY ON VARIOUS SOFTWARE DEVELOPERS' SITES. HE WAS A MEMBER OF THE C++ STANDARDS COMMITTEE AND HAS A MASTER'S DEGREE IN GENERAL LINGUISTICS.

"

RULE OF THREE (C++ PROGRAMMING)

"The **rule of three**, **rule of five**, and **rule of 0** are rules of thumb in C++ for the building of exception-safe code & for formalizing rules on resource management. It accomplishes this by prescribing how the default members of aclass should be used to accomplish this task in a systematic manner.

CONTENTS

1 Rule of Three

2 Rule of 5

3 Rule Of 0

4 Example in C++

5 See also

6 References

RULE OF THREE

The **rule of three** (also known as the Law of The Big Three or The Big Three) is a rule of thumb in C++ (prior to C++11) that claims that if a class defines one of the following it should probably explicitly define all three:[1]

- destructor
- copy constructor
- copy assignment operator

These three functions are special member functions. If one of these functions is used without first being declared by the programmer it will be implicitly implemented by the compiler with the default semantics of performing the said operation on all the members of the class. The default semantics are:

- **Destructor** - Call the destructors of all the object's class-type members
- **Copy constructor** - Construct all the object's members from the corresponding members of the copy constructor's argument, calling the copy constructors of the object's class-type members, and doing a plain assignment of all non-class type (e.g., *int* or pointer) data members
- **Copy assignment operator** - Assign all the object's members from the corresponding members of the assignment operator's argument, calling the copy assignment operators of the object's class-type members, and doing a plain assignment of all non-class type (e.g., *int* or pointer) data members.

The Rule of Three claims that if one of these had to be defined by the programmer, it means that the compiler-generated version does not fit the needs of the class in one case and it will probably not fit in the other cases either. The term "Rule of three" was coined by Marshall Cline in 1991.[2]

An amendment to this rule is that if Resource Acquisition Is Initialization (RAII) is used for the class members, the destructor may be left undefined (also known as The Law of The Big Two[3]).

Because implicitly-generated constructors and assignment operators simply copy all class data members,[4] one should define explicit copy constructors and copy assignment operators for classes that encapsulate complex data structures or have external references such as pointers, since only the pointer gets copied, not the object it points to. In the case that this default behavior is actually the intended behavior, an explicit declaration can prevent ambiguity.

RULE OF 5

With the advent of C++11 the rule of three probably needs to be broadened to *the rule of five* as C++11 implements *move semantics*,[5] allowing destination objects to *grab* (or *steal*) data from temporary objects. The following example also shows the new moving members: move constructor

and move assignment operator. Consequently, for *the rule of five* we have the following *special members*:

- destructor
- copy constructor
- move constructor
- copy assignment operator
- move assignment operator

Note also that situations exist where classes may need destructors, but cannot sensibly implement copy and move constructors and copy and move assignment operators. This happens, e.g., when the base class does not support these latter *Big Four* members, but the derived class's constructor allocates memory for its own use[citation needed]. In C++11, this can be simplified by explicitly specifying the 5 members as default [6]

RULE OF 0

There's a proposal by R. Martinho Fernandes to simplify all of the above into a Rule of 0 for C++ (primarily for C++11 & newer).[7] The rule of 0 states that if you specify any of the default members, then your class must deal exclusively with a single resource. Furthermore, it must define all default members to handle that resource (or delete the default member as appropriate). Thus such classes must follow the **rule of 5** described above. A resource can be anything: memory that gets allocated, a file descriptor, database transaction etc.

Any other class must not allocate any resources directly. Furthermore, they must omit the default members (or explicitly assign all of them to default via **= default**). Any resources should be used indirectly by using the single-resource classes as member/local variables. This lets such classes inherit the default members from the union of member variables, thereby auto-forwarding the movability/copyability of the union of all underlying resource. Since ownership of 1 resource is owned by exactly 1 member variable, exceptions in the constructor cannot leak resources due to RAII. Fully initialized variables will have their destructors called & uninitialized variables could not have owned any resources to begin with.

Since the majority of classes don't deal with ownership as their sole concern, the majority of classes can omit the default members. This is where the rule-of-0 gets its name.

EXAMPLE IN C++

```
#include <cstring>
#include <iostream>
```

```
class Foo
{
public:
    /** Constructor */
    Foo() :
        data (new char[14])
    {
        std::strcpy (data, "Hello, World!");
    }

    /** Copy Constructor */
    Foo (const Foo& other) :
        data (new char[std::strlen (other.data) + 1])
    {
        std::strcpy (data, other.data);
    }

    /** Move Constructor */
    Foo (Foo&& other) noexcept : /* noexcept needed to enable
optimizations in containers */
        data(other.data)
    {
        other.data = nullptr;
    }

    /** Destructor */
    ~Foo() noexcept /* explicitly specified destructors should be
annotated noexcept as best-practice */
    {
        delete[] data;
    }

    /** Copy Assignment Operator */
    Foo& operator= (const Foo& other)
    {
        Foo tmp(other); // re-use copy-constructor
        *this = std::move(tmp); // re-use move-assignment
        return *this;
    }

    /** Move Assignment Operator */
    Foo& operator= (Foo&& other) noexcept
    {
        // simplified move-constructor that also protects against move-to-
self.
```

```
        // see http://scottmeyers.blogspot.com/2014/06/the-drawbacks-of-
implementing-move.html
        // for how to implement a more-optimized version of this.
        using namespace std;
        swap(data, other.data); // repeat for all elements
        return *this;
    }

private:
    friend std::ostream& operator<< (std::ostream& os, const Foo& foo)
    {
        os << foo.data;
        return os;
    }

    char* data;
};

int main()
{
    const Foo foo;
    std::cout << foo << std::endl;

    return 0;
}
```

” http://en.wikipedia.org/wiki/Rule_of_three_(C%2B%2B_programming)

UNIVERSAL REFERENCE

“UNIVERSAL REFERENCES IN C++11

T&& DOESN’T ALWAYS MEAN “RVALUE REFERENCE”

by Scott Meyers

Related materials:

- A video of Scott’s C&B talk based on this material is available on Channel 9.
- A black-and-white PDF version of this article is available in Overload 111.

Perhaps the most significant new feature in C++11 is rvalue references; they're the foundation on which move semantics and perfect forwarding are built. (If you're unfamiliar with the basics of rvalue references, move semantics, or perfect forwarding, you may wish to read Thomas Becker's overview before continuing.) Syntactically, rvalue references are declared like "normal" references (now known as LVALUE REFERENCES), except you use two ampersands instead of one. This function takes a parameter of type rvalue-reference-to-`Widget`:

```
void f(Widget&& param);
```

Given that rvalue references are declared using "`&&`", it seems reasonable to assume that the presence of "`&&`" in a type declaration indicates an rvalue reference. That is not the case:

```
Widget&& var1 = someWidget;      // here, "&&" means rvalue reference

auto&& var2 = var1;              // here, "&&" does not mean rvalue reference

template<typename T>
void f(std::vector<T>&& param);  // here, "&&" means rvalue reference

template<typename T>
void f(T&& param);               // here, "&&"does not mean rvalue reference
```

In this article, I describe the two meanings of "`&&`" in type declarations, explain how to tell them apart, and introduce new terminology that makes it possible to unambiguously communicate which meaning of "`&&`" is intended. Distinguishing the different meanings is important, because if you think "rvalue reference" whenever you see "`&&`" in a type declaration, you'll misread a lot of C++11 code. The essence of the issue is that "`&&`" in a type declaration sometimes means rvalue reference, but sometimes it means EITHER rvalue reference OR lvalue reference. As such, some occurrences of "`&&`" in source code may actually have the meaning of "`&`", i.e., have the syntactic APPEARANCE of an rvalue reference ("`&&`"), but the MEANING of an lvalue reference ("`&`"). References where this is possible are more flexible than either lvalue references or rvalue references. Rvalue references may bind only to rvalues, for example, and lvalue references, in addition to being able to bind to lvalues, may bind to rvalues only under restricted circumstances.[1] In contrast, references declared with "`&&`" that may be either lvalue references or rvalue references may bind to ANYTHING. Such

unusually flexible references deserve their own name. I call them UNIVERSAL REFERENCES.

The details of when "`&&`" indicates a universal reference (i.e., when "`&&`" in source code might actually mean "`&`") are tricky, so I'm going to postpone coverage of the minutiae until later. For now, let's focus on the following rule of thumb, because that is what you need to remember during day-to-day programming:

If a variable or parameter is declared to have type **`T&&`** for some **deduced type** `T`, that variable or parameter is a UNIVERSAL REFERENCE.

The requirement that type deduction be involved limits the situations where universal references can be found. In practice, almost all universal references are parameters to function templates. Because the type deduction rules for `auto`-declared variables are essentially the same as for templates, it's also possible to have `auto`-declared universal references. These are uncommon in production code, but I show some in this article, because they are less verbose in examples than templates. In the Nitty Gritty Details section of this article, I explain that it's also possible for universal references to arise in conjunction with uses of `typedef` and `decltype`, but until we get down to the nitty gritty details, I'm going to proceed as if universal references pertained only to function template parameters and `auto`-declared variables.

The constraint that the form of a universal reference be `T&&` is more significant than it may appear, but I'll defer examination of that until a bit later. For now, please simply make a mental note of the requirement.

Like all references, universal references must be initialized, and it is a universal reference's initializer that determines whether it represents an lvalue reference or an rvalue reference:

- If the expression initializing a universal reference is an lvalue, the universal reference becomes an lvalue reference.
- If the expression initializing the universal reference is an rvalue, the universal reference becomes an rvalue reference.

This information is useful only if you are able to distinguish lvalues from rvalues. A precise definition for these terms is difficult to develop (the C++11

standard generally specifies whether an expression is an lvalue or an rvalue on a case-by-case basis), but in practice, the following suffices:

- If you can take the address of an expression, the expression is an lvalue.
- If the type of an expression is an lvalue reference (e.g., `T&` or `const T&`, etc.), that expression is an lvalue.
- Otherwise, the expression is an rvalue. Conceptually (and typically also in fact), rvalues correspond to temporary objects, such as those returned from functions or created through implicit type conversions. Most literal values (e.g., `10` and `5.3`) are also rvalues.

Consider again the following code from the beginning of this article:

```
1. Widget&& var1 = someWidget;
2. auto&& var2 = var1;
```

You can take the address of `var1`, so `var1` is an lvalue. `var2`'s type declaration of `auto&&` makes it a universal reference, and because it's being initialized with `var1` (an lvalue), `var2` becomes an lvalue reference. A casual reading of the source code could lead you to believe that `var2` was an rvalue reference; the "`&&`" in its declaration certainly suggests that conclusion. But because it is a universal reference being initialized with an lvalue, `var2` becomes an lvalue reference. It's as if `var2` were declared like this:

```
1. Widget& var2 = var1;
```

As noted above, if an expression has type lvalue reference, it's an lvalue. Consider this example:

```
1. std::vector<int> v;
2. ...
3. auto&& val = v[0];           // val becomes an lvalue reference (see below)
```

`val` is a universal reference, and it's being initialized with `v[0]`, i.e., with the result of a call to `std::vector<int>::operator[]`. That function returns an lvalue reference to an element of the vector.[2] Because all lvalue references are lvalues, and because this lvalue is used to initialize `val`, `val` becomes an lvalue reference, even though it's declared with what looks like an rvalue reference.

I remarked that universal references are most common as parameters in template functions. Consider again this template from the beginning of this article:

```
template<typename T>
void f(T&& param);              // "&&" might mean rvalue reference
```

Given this call to f,

```
f(10);                          // 10 is an rvalue
```

`param` is initialized with the literal `10`, which, because you can't take its address, is an rvalue. That means that in the call to `f`, the universal reference `param` is initialized with an rvalue, so `param`becomes an rvalue reference – in particular, `int&&`.

On the other hand, if `f` is called like this,

```
int x = 10;
f(x);                           // x is an lvalue
```

`param` is initialized with the variable `x`, which, because you can take its address, is an lvalue. That means that in this call to `f`, the universal reference param is initialized with an lvalue, and param therefore becomes an lvalue reference – `int&`, to be precise.

The comment next to the declaration of `f` should now be clear: whether `param`'s type is an lvalue reference or an rvalue reference depends on what is passed when `f` is called. Sometimes param becomes an lvalue reference, and sometimes it becomes an rvalue reference. `param` really is a UNIVERSAL REFERENCE.

Remember that "`&&`" indicates a universal reference ONLY WHERE TYPE DEDUCTION TAKES PLACE. Where there's no type deduction, there's no universal reference. In such cases, "`&&`" in type declarations always means rvalue reference. Hence:

```
template<typename T>
void f(T&& param);              // deduced parameter type ⇒ type deduction;
                                // && ≡ universal reference

template<typename T>
class Widget {
    ...
    Widget(Widget&& rhs);       // fully specified parameter type ⇒ no type deduction;
    ...                         // && ≡ rvalue reference
};

template<typename T1>
class Gadget {
    ...
    template<typename T2>
    Gadget(T2&& rhs);           // deduced parameter type ⇒ type deduction;
    ...                         // && ≡ universal reference
};

void f(Widget&& param);         // fully specified parameter type ⇒ no type deduction;
```

```
21.                     // && ≡ rvalue reference
```

There's nothing surprising about these examples. In each case, if you see `T&&` (where `T` is a template parameter), there's type deduction, so you're looking at a universal reference. And if you see "`&&`" after a particular type name (e.g., `Widget&&`), you're looking at an rvalue reference.

I stated that the form of the reference declaration must be "`T&&`" in order for the reference to be universal. That's an important caveat. Look again at this declaration from the beginning of this article:

```
1.  template<typename T>
2.  void f(std::vector<T>&& param);     // "&&" means rvalue reference
```

Here, we have both type deduction and a "`&&`"-declared function parameter, but the form of the parameter declaration is not "`T&&`", it's "`std::vector<t>&&`". As a result, the parameter is a normal rvalue reference, not a universal reference. Universal references can only occur in the form "`T&&`"! Even the simple addition of a `const` qualifier is enough to disable the interpretation of "`&&`" as a universal reference:

```
1.  template<typename T>
2.  void f(const T&& param);            // "&&" means rvalue reference
```

Now, "`T&&`" is simply the required FORM for a universal reference. It doesn't mean you have to use the name `T` for your template parameter:

```
1.  template<typename MyTemplateParamType>
2.  void f(MyTemplateParamType&& param);  // "&&" means universal reference
```

Sometimes you can see `T&&` in a function template declaration where `T` is a template parameter, yet there's still no type deduction. Consider this `push_back` function in `std::vector`:[3]

```
1.  template <class T, class Allocator = allocator<T> >
2.  class vector {
3.  public:
4.     ...
5.     void push_back(T&& x);      // fully specified parameter type ⇒ no type deduction;
6.     ...                         // && ≡ rvalue reference
7.  };
```

Here, `T` is a template parameter, and `push_back` takes a `T&&`, yet the parameter is not a universal reference! How can that be?

The answer becomes apparent if we look at how `push_back` would be declared outside the class. I'm going to pretend that `std::vector`'s `Allocator` parameter doesn't exist, because it's irrelevant to the discussion, and it just clutters up the

code. With that in mind, here's the declaration for this version of `std::vector::push_back`:

```
template <class T>
void vector<T>::push_back(T&& x);
```

`push_back` can't exist without the class `std::vector<T>` that contains it. But if we have a class `std::vector<T>`, we already know what `T` is, so there's no need to deduce it.

An example will help. If I write

```
Widget makeWidget();           // factory function for Widget
std::vector<Widget> vw;
...
Widget w;
vw.push_back(makeWidget());    // create Widget from factory, add it to vw
```

my use of `push_back` will cause the compiler to instantiate that function for the class `std::vector<Widget>`. The declaration for that `push_back` looks like this:

```
void std::vector<Widget>::push_back(Widget&& x);
```

See? Once we know that the class is `std::vector<Widget>`, the type of `push_back`'s parameter is fully determined: it's `Widget&&`. There's no role here for type deduction.

Contrast that with `std::vector`'s `emplace_back`, which is declared like this:

```
template <class T, class Allocator = allocator<T> >
class vector {
public:
  ...
  template <class... Args>
  void emplace_back(Args&&... args); // deduced parameter types ⇒ type deduction;
  ...                                // && ≡ universal references
};
```

Don't let the fact that `emplace_back` takes a variable number of arguments (as indicated by the ellipses in the declarations for `Args` and `args`) distract you from the fact that a type for each of those arguments must be deduced. The function template parameter `Args` is independent of the class template parameter `T`, so even if we know that the class is, say, `std::vector<Widget>`, that doesn't tell us the type(s) taken by `emplace_back`. The out-of-class declaration for `emplace_back` for `std::vector<Widget>` makes that clear (I'm continuing to ignore the existence of the `Allocator`parameter):

```
template<class... Args>
void std::vector<Widget>::emplace_back(Args&&... args);
```

Clearly, knowing that the class is `std::vector<Widget>` doesn't eliminate the need for the compiler to deduce the type(s) passed to `emplace_back`. As a result, `std::vector::emplace_back`'s parameters are universal references, unlike the parameter to the version of `std::vector::push_back` we examined, which is an rvalue reference.

A final point is worth bearing in mind: the lvalueness or rvalueness of an expression is independent of its type. Consider the type `int`. There are lvalues of type `int` (e.g., variables declared to be `int`s), and there are rvalues of type `int` (e.g., literals like `10`). It's the same for user-defined types like `Widget`. A `Widget` object can be an lvalue (e.g., a `Widget` variable) or an rvalue (e.g., an object returned from a `Widget`-creating factory function). The type of an expression does not tell you whether it is an lvalue or an rvalue.

Because the lvalueness or rvalueness of an expression is independent of its type, it's possible to have LVALUES whose type is RVALUE REFERENCE, and it's also possible to have RVALUES of the type RVALUE REFERENCE:

```
Widget makeWidget();                    // factory function for Widget

Widget&& var1 = makeWidget()            // var1 is an lvalue, but
                                        // its type is rvalue reference (to Widget)

Widget var2 = static_cast<Widget&&>(var1); // the cast expression yields an rvalue, but
                                        // its type is rvalue reference  (to Widget)
```

The conventional way to turn lvalues (such as `var1`) into rvalues is to use `std::move` on them, so `var2` could be defined like this:

```
Widget var2 = std::move(var1);          // equivalent to above
```

I initially showed the code with `static_cast` only to make explicit that the type of the expression was an rvalue reference (`Widget&&`).

Named variables and parameters of rvalue reference type are lvalues. (You can take their addresses.) Consider again the `Widget` and `Gadget` templates from earlier:

```
template<typename T>
class Widget {
    ...
    Widget(Widget&& rhs);       // rhs's type is rvalue reference,
    ...                         // but rhs itself is an lvalue
};

template<typename T1>
class Gadget {
    ...
    template <typename T2>
```

```
12.     Gadget(T2&& rhs);          // rhs is a universal reference whose type will
13.     ...                        // eventually become an rvalue reference or
14. };                             // an lvalue reference, but rhs itself is an lvalue
```

In `Widget`'s constructor, `rhs` is an rvalue reference, so we know it's bound to an rvalue (i.e., an rvalue was passed to it), but `rhs` itself is an lvalue, so we have to convert it back to an rvalue if we want to take advantage of the rvalueness of what it's bound to. Our motivation for this is generally to use it as the source of a move operation, and that's why the way to convert an lvalue to an rvalue is to use `std::move`. Similarly, `rhs` in `Gadget`'s constructor is a universal reference, so it might be bound to an lvalue or to an rvalue, but regardless of what it's bound to, `rhs` itself is an lvalue. If it's bound to an rvalue and we want to take advantage of the rvalueness of what it's bound to, we have to convert `rhs` back into an rvalue. If it's bound to an lvalue, of course, we don't want to treat it like an rvalue. This ambiguity regarding the lvalueness and rvalueness of what a universal reference is bound to is the motivation for `std::forward`: to take a universal reference lvalue and convert it into an rvalue only if the expression it's bound to is an rvalue. The name of the function ("`forward`") is an acknowledgment that our desire to perform such a conversion is virtually always to preserve the calling argument's lvalueness or rvalueness when passing – FORWARDING – it to another function.

But `std::move` and `std::forward` are not the focus of this article. The fact that "`&&`" in type declarations may or may not declare an rvalue reference is. To avoid diluting that focus, I'll refer you to the references in the Further Information section for information on `std::move` and `std::forward`.

NITTY GRITTY DETAILS

The true core of the issue is that some constructs in C++11 give rise to references to references, and references to references are not permitted in C++. If source code explicitly contains a reference to a reference, the code is invalid:

```
1. Widget w1;
2. ...
3. Widget& & w2 = w1;          // error! No such thing as "reference to reference"
```

There are cases, however, where references to references arise as a result of type manipulations that take place during compilation, and in such cases, rejecting the

code would be problematic. We know this from experience with the initial standard for C++, i.e., C++98/C++03.

During type deduction for a template parameter that is a universal reference, lvalues and rvalues of the same type are deduced to have slightly different types. In particular, lvalues of type `T` are deduced to be of type `T&` (i.e., lvalue reference to `T`), while rvalues of type `T` are deduced to be simply of type `T`. (Note that while lvalues are deduced to be lvalue references, rvalues are not deduced to be rvalue references!) Consider what happens when a template function taking a universal reference is invoked with an rvalue and with an lvalue:

```
template<typename T>
void f(T&& param);

...

int x;

...

f(10);                  // invoke f on rvalue
f(x);                   // invoke f on lvalue
```

In the call to `f` with the rvalue `10`, `T` is deduced to be `int`, and the instantiated `f` looks like this:

```
void f(int&& param);          // f instantiated from rvalue
```

That's fine. In the call to `f` with the lvalue `x`, however, `T` is deduced to be `int&`, and `f`'s instantiation contains a reference to a reference:

```
void f(int& && param);        // initial instantiation of f with lvalue
```

Because of the reference-to-reference, this instantiated code is PRIMA FACIE invalid, but the source code– "`f(x)`" – is completely reasonable. To avoid rejecting it, C++11 performs "reference collapsing" when references to references arise in contexts such as template instantiation.

Because there are two kinds of references (lvalue references and rvalue references), there are four possible reference-reference combinations: lvalue reference to lvalue reference, lvalue reference to rvalue reference, rvalue reference to lvalue reference, and rvalue reference to rvalue reference. There are only two reference-collapsing rules:

- An rvalue reference to an rvalue reference becomes (“collapses into”) an rvalue reference.
- All other references to references (i.e., all combinations involving an lvalue reference) collapse into an lvalue reference.

Applying these rules to the instantiation of f on an lvalue yields the following valid code, which is how the compiler treats the call:

```
1.  void f(int& param);          // instantiation of f with lvalue after reference collapsing
```

This demonstrates the precise mechanism by which a universal reference can, after type deduction and reference collapsing, become an lvalue reference. The truth is that a universal reference is really just an rvalue reference in a reference-collapsing context.

Things get subtler when deducing the type for a variable that is itself a reference. In that case, the reference part of the type is ignored. For example, given

```
1.  int x;
2.
3.  ...
4.
5.  int&& r1 = 10;              // r1's type is int&&
6.
7.  int& r2 = x;                // r2's type is int&
```

the type for both `r1` and `r2` is considered to be `int` in a call to the template `f`. This reference-stripping behavior is independent of the rule that, during type deduction for universal references, lvalues are deduced to be of type `T&` and rvalues of type `T`, so given these calls,

```
1.  f(r1);
2.
3.  f(r2);
```

the deduced type for both `r1` and `r2` is `int&`. Why? First the reference parts of `r1`’s and `r2`’s types are stripped off (yielding `int` in both cases), then, because each is an lvalue, each is treated as `int&`during type deduction for the universal reference parameter in the call to `f`.

Reference collapsing occurs, as I’ve noted, in “contexts such as template instantiation.” A second such context is the definition of `auto` variables. Type deduction for `auto` variables that are universal references is essentially identical to type deduction for function template parameters that are universal references,

so lvalues of type `T` are deduced to have type `T&`, and rvalues of type `T` are deduced to have type `T`. Consider again this example from the beginning of this article:

```
Widget&& var1 = someWidget;      // var1 is of type Widget&& (no use of auto here)

auto&& var2 = var1;              // var2 is of type Widget& (see below)
```

`var1` is of type `Widget&&`, but its reference-ness is ignored during type deduction in the initialization of `var2`; it's considered to be of type `Widget`. Because it's an lvalue being used to initialize a universal reference (`var2`), its deduced type is `Widget&`. Substituting `Widget&` for `auto` in the definition for `var2` yields the following invalid code,

```
Widget& && var2 = var1;          // note reference-to-reference
```

which, after reference collapsing, becomes

```
Widget& var2 = var1;             // var2 is of type Widget&
```

A third reference-collapsing context is `typedef` formation and use. Given this class template,

```
template<typename T>
class Widget {
   typedef T& LvalueRefType;
   ...
};
```

and this use of the template,

```
Widget<int&> w;
```

the instantiated class would contain this (invalid) typedef:

```
typedef int& & LvalueRefType;
```

Reference-collapsing reduces it to this legitimate code:

```
typedef int& LvalueRefType;
```

If we then use this `typedef` in a context where references are applied to it, e.g.,

```
void f(Widget<int&>::LvalueRefType&& param);
```

the following invalid code is produced after expansion of the `typedef`,

```
void f(int& && param);
```

but reference-collapsing kicks in, so `f`'s ultimate declaration is this:

```
void f(int& param);
```

The final context in which reference-collapsing takes place is the use of `decltype`. As is the case with templates and `auto`, `decltype` performs type deduction on expressions that yield types that are either `T` or `T&`, and `decltype` then applies C++11's reference-collapsing rules. Alas, the type-

deduction rules employed by `decltype` are not the same as those used during template or `auto` type deduction. The details are too arcane for coverage here (the Further Information section provides pointers to, er, further information), but a noteworthy difference is that `decltype`, given a named variable of non-reference type, deduces the type `T` (i.e., a non-reference type), while under the same conditions, templates and `auto` deduce the type `T&`. Another important difference is that `decltype`'s type deduction depends only on the `decltype` expression; the type of the initializing expression (if any) is ignored. Ergo:

```
1.  Widget w1, w2;
2.
3.  auto&& v1 = w1;              // v1 is an auto-based universal reference being
4.                               // initialized with an lvalue, so v1 becomes an
5.                               // lvalue reference referring to w1.
6.
7.  decltype(w1)&& v2 = w2;      // v2 is a decltype-based universal reference, and
8.                               // decltype(w1) is Widget, so v2 becomes an rvalue reference.
9.                               // w2 is an lvalue, and it's not legal to initialize an
10.                              // rvalue reference with an lvalue, so
11.                              // this code does not compile.
```

SUMMARY

In a type declaration, "&&" indicates either an rvalue reference or a UNIVERSAL REFERENCE – a reference that may resolve to either an lvalue reference or an rvalue reference. Universal references always have the form `T&&` for some deduced type `T`.

REFERENCE COLLAPSING is the mechanism that leads to universal references (which are really just rvalue references in situations where reference-collapsing takes place) sometimes resolving to lvalue references and sometimes to rvalue references. It occurs in specified contexts where references to references may arise during compilation. Those contexts are template type deduction, `auto` type deduction, `typedef` formation and use, and `decltype` expressions.

" http://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers

RVALUE REFERENCES

"If X is any type, then X&& is called an *rvalue reference* to X. For better distinction, the ordinary reference X& is now also called an *lvalue reference*.

An rvalue reference is a type that behaves much like the ordinary reference `X&`, with several exceptions. The most important one is that when it comes to function overload resolution, lvalues prefer old-style lvalue references, whereas rvalues prefer the new rvalue references:

```
void foo(X& x); // lvalue reference overload
void foo(X&& x); // rvalue reference overload

X x;
X foobar();

foo(x); // argument is lvalue: calls foo(X&)
foo(foobar()); // argument is rvalue: calls foo(X&&)
```

So the gist of it is:

> Rvalue references allow a function to branch at compile time (via overload resolution) on the condition "Am I being called on an lvalue or an rvalue?"

It is true that you can overload *any* function in this manner, as shown above. But in the overwhelming majority of cases, this kind of overload should occur only for copy constructors and assignment operators, for the purpose of achieving move semantics:

```
X& X::operator=(X const & rhs); // classical implementation
X& X::operator=(X&& rhs)
{
  // Move semantics: exchange content between this and rhs
  return *this;
}
```

Implementing an rvalue reference overload for the copy constructor is similar.

> **Caveat:** As it happens so often in C++, what looks just right at first glance is still a little shy of perfect. It turns out that in some cases, the simple exchange of content between `this` and `rhs` in the implementation of the copy assignment operator above is not quite good enough. We'll come back to this in Section 4, "Forcing Move Semantics" below.

> **Note:** If you implement `void foo(X&);` but not `void foo(X&&);` then of course the behavior is unchanged: `foo` can be called on l-values, but not on r-values. If you implement `void foo(X const &);` but not `void foo(X&&);`

then again, the behavior is unchanged: `foo` can be called on l-values and r-values, but it is not possible to make it distinguish between l-values and r-values. That is possible only by implementing

```
void foo(X&&);
```

as well. Finally, if you implement

```
void foo(X&&);
```

but neither one of

```
void foo(X&);
```

and

```
void foo(X const &);
```

then, according to the final version of C++11, `foo` can be called on r-values, but trying to call it on an l-value will trigger a compile error.

A good example is the std library function `swap`. As before, let X be a class for which we have overloaded the copy constructor and copy assignment operator to achieve move semantics on rvalues.

```
template<class T>
void swap(T& a, T& b)
{
  T tmp(a);
  a = b;
  b = tmp;
}

X a, b;
swap(a, b);
```

There are no rvalues here. Hence, all three lines in `swap` use non-move semantics. But we know that move semantics would be fine: wherever a variable occurs as the source of a copy construction or assignment, that variable is either not used again at all, or else it is used only as the target of an assignment.

In C++11, there is an std library function called `std::move` that comes to our rescue. It is a function that turns its argument into an rvalue without doing anything else. Therefore, in C++11, the std library function `swap` looks like this:

```
template<class T>
void swap(T& a, T& b)
{
  T tmp(std::move(a));
  a = std::move(b);
  b = std::move(tmp);
}

X a, b;
swap(a, b);
```

Now all three lines in `swap` use move semantics. Note that for those types that do not implement move semantics (that is, do not overload their copy constructor and assignment operator with an rvalue reference version), the new `swap` behaves just like the old one.

`std::move` is a very simple function. Unfortunately, though, I cannot show you the implementation yet. We'll come back to it later.

Using `std::move` wherever we can, as shown in the `swap` function above, gives us the following important benefits:

- For those types that implement move semantics, many standard algorithms and operations will use move semantics and thus experience a potentially significant performance gain. An important example is inplace sorting: inplace sorting algorithms do hardly anything else but swap elements, and this swapping will now take advantage of move semantics for all types that provide it.
- The STL often requires copyability of certain types, e.g., types that can be used as container elements. Upon close inspection, it turns out that in many cases, moveability is enough. Therefore, we can now use types that are moveable but not copyable (`unique_pointer` comes to mind) in many places where previously, they were not allowed. For example, these types can now be used as STL container elements.

Now that we know about `std::move`, we are in a position to see why the implementation of the rvalue reference overload of the copy assignment operator that I showed earlier is still a bit problematic. Consider a simple assignment between variables, like this:

```
a = b;
```

What do you expect to happen here? You expect the object held by `a` to be replaced by a copy of `b`, and in the course of this replacement, you expect the object formerly held by `a` to be destructed. Now consider the line

```
a = std::move(b);
```

If move semantics are implemented as a simple swap, then the effect of this is that the objects held by `a` and `b` are being exchanged between `a` and `b`. Nothing is being destructed yet. The object formerly held by `a` will of course be destructed eventually, namely, when `b` goes out of scope. Unless, of course, `b` becomes the target of a move, in which case the object formerly held by `a` gets passed on again. Therefore, as far as the implementer of the copy assignment operator is concerned, it is not known when the object formerly held by `a` will be destructed.

So in a sense, we have drifted into the netherworld of non-deterministic destruction here: a variable has been assigned to, but the object formerly held by that variable is still out there somewhere. That's fine as long as the destruction of that object does not have any side effects that are visible to the outside world. But sometimes destructors do have such side effects. An example would be the release of a lock inside a destructor. Therefore, any part of an object's destruction that has side effects should be performed explicitly in the rvalue reference overload of the copy assignment operator:

```
X& X::operator=(X&& rhs)
{
```

```
  // Perform a cleanup that takes care of at least those parts
of the
  // destructor that have side effects. Be sure to leave the
object
  // in a destructible and assignable state.

  // Move semantics: exchange content between this and rhs

  return *this;
}
```

IS AN RVALUE REFERENCE AN RVALUE?

As before, let X be a class for which we have overloaded the copy constructor and copy assignment operator to implement move semantics. Now consider:

```
void foo(X&& x)
{
  X anotherX = x;
  // ...
}
```

The interesting question is: which overload of `X`'s copy constructor gets called in the body of `foo`? Here, `x` is a variable that is declared as an rvalue reference, that is, a reference which preferably and typically (although not necessarily!) refers to an rvalue. Therefore, it is quite plausible to expect that `x` itself should also bind like an rvalue, that is,

```
X(X&& rhs);
```

should be called. In other words, one might expect that anything that is declared as an rvalue reference is itself an rvalue. The designers of rvalue references have chosen a solution that is a bit more subtle than that:

> Things that are declared as rvalue reference can be lvalues or rvalues. The distinguishing criterion is: IF IT HAS A NAME, then it is an lvalue. Otherwise, it is an rvalue.

In the example above, the thing that is declared as an rvalue reference has a name, and therefore, it is an lvalue:

```
void foo(X&& x)
{
  X anotherX = x; // calls X(X const & rhs)
}
```

Here is an example of something that is declared as an rvalue reference and does not have a name, and is therefore an rvalue:

```
X&& goo();
X x = goo(); // calls X(X&& rhs) because the thing on
             // the right hand side has no name
```

And here's the rationale behind the design: Allowing move sematics to be applied tacitly to something that has a name, as in

```
  X anotherX = x;
  // x is still in scope!
```

would be dangerously confusing and error-prone because the thing from which we just moved, that is, the thing that we just pilfered, is still accessible on subsequent lines of code. But the whole point of move semantics was to apply it only where it "doesn't matter," in the sense that the thing from which we move dies and goes away right after the moving. Hence the rule, "If it has a name, then it's an lvalue."

So then what about the other part, "If it does not have a name, then it's an rvalue?" Looking at the `goo` example above, it is technically possible, though not very likely, that the expression`goo()` in the second line of the example refers to something that is still accessible after it has been moved from. But recall from the previous section: sometimes that's what we want! We want to be able to force move semantics on lvalues at our discretion, and it is precisely the rule, "If it does not have a name, then it's an rvalue" that allows us to achieve that in a controlled manner. That's how the function `std::move` works. Although it is still too early to show you the exact implementation, we just got a step closer to understanding `std::move`. It passes its argument right through by reference, doing nothing with it at all, and its result type is rvalue reference. So the expression

```
std::move(x)
```

is declared as an rvalue reference and does not have a name. Hence, it is an rvalue. Thus,`std::move` "turns its argument into an rvalue even if it isn't," and it achieves that by "hiding the name."

Here is an example that shows how important it is to be aware of the if-it-has-a-name rule. Suppose you have written a class `Base`, and you have implemented move semantics by overloading `Base`'s copy constructor and assignment operator:

```
Base(Base const & rhs); // non-move semantics
Base(Base&& rhs); // move semantics
```

Now you write a class `Derived` that is derived from `Base`. In order to assure that move semantics is applied to the `Base` part of your `Derived` objects, you must overload `Derived`'s copy constructor and assignment operator as well. Let's look at the copy constructor. The copy assignment operator is handled analogously. The version for lvalues is straightforward:

```
Derived(Derived const & rhs)
  : Base(rhs)
{
  // Derived-specific stuff
}
```

The version for rvalues has a big fat subtlety. Here's what someone who is not aware of the if-it-has-a-name rule might have done:

```
Derived(Derived&& rhs)
  : Base(rhs) // wrong: rhs is an lvalue
{
  // Derived-specific stuff
}
```

If we were to code it like that, the non-moving version of `Base`'s copy constructor would be called, because `rhs`, having a name, is an lvalue. What we want to be called is `Base`'s moving copy constructor, and the way to get that is to write

```
Derived(Derived&& rhs)
  : Base(std::move(rhs)) // good, calls Base(Base&& rhs)
{
  // Derived-specific stuff
}
```

MOVE SEMANTICS AND COMPILER OPTIMIZATIONS

Consider the following function definition:

```
X foo()
{
  X x;
  // perhaps do something to x
  return x;
}
```

Now suppose that as before, X is a class for which we have overloaded the copy constructor and copy assignment operator to implement move semantics. If you take the function definition above at face value, you may be tempted to say, wait a minute, there is a value copy happening here from x to the location of `foo`'s return value. Let me make sure we're using move semantics instead:

```
X foo()
{
  X x;
  // perhaps do something to x
  return std::move(x); // making it worse!
}
```

Unfortunately, that would make things worse rather than better. Any modern compiler will applyRETURN VALUE OPTIMIZATION to the original function definition. In other words, rather than constructing an `X` locally and then copying it out, the compiler would construct the `X` object directly at the location of `foo`'s return value. Rather obviously, that's even better than move semantics.

So as you can see, in order to really use rvalue references and move semantics in an optimal way, you need to fully understand and take into account today's compilers' "special effects" such as return value optimization and copy elision. Dave Abrahams has written an excellentseries of articles on this subject on his blog. The details get pretty subtle, but hey, we chose C++ as our language of choice for a reason, right? We made our beds, so now let's lie in them." http://thbecker.net/articles/rvalue_references/section_01.html

REFERENCE COLLAPSING RULES

- `A& &` becomes `A&`
- `A& &&` becomes `A&`
- `A&& &` becomes `A&`
- `A&& &&` becomes `A&&`

OPERATORS OVERLOADING AND STD::MOVE

“A little while ago while I was surfing the web I stumbled on a blog of a C++ programmer. Sadly, I was unable to retrace my steps, so I will have to reconstruct what I found there from memory. The programmer was faced with the task of multiplying two matrices with each other. The result is yet another matrix. Efficiency and ease of use were important criteria. Here is the relevant part of the code:

```
1 matrix & matrix::operator* (matrix const & other) const
2 {
3    // snip: ... assert matrix sizes are compatible ...
4    matrix * result = new matrix(rows(), other.columns());
5    // snip: ... compute and store matrix elements ...
6    return *result;
7 }
```

Let us put on our CSI C++ hat for a while.

WHY THE CODE IS LIKE IT IS

Apparently, the programmer was aware of a few things. He was aware that copying a matrix should be avoided, since they are quite often very large. Thus, the function cannot return a matrix object. He thought that returning a pointer is unacceptable, since the user would have to take care of dereferencing and stuff. A reference is as convenient to use as a copy, but does not copy any data, so this might explain the return type. Finally, the programmer knew that local variables are destroyed when they go out of scope. Variables created on the

heap, i.e., created with new, escape this fate. So above code is a convenient and efficient solution to the multiplication problem. And it leaks memory. Oops!

INSPECTING THE MEMORY LEAK

Above snippet leaks memory because for the new there is no matching delete. To be fair, there is a way to use the code correctly:

```
1 matrix A; // snip: filling A with valuesthe
2 matrix B; // snip: filling B with values
3
4 matrix & C = A * B;
5 delete (&C); // call delete with address of matrix C
```

For chained multiplications (A * (B * C)) memory will be lost. In case of exceptionsdelete might not get called at all. Users may—and will—forget to call delete. This is why the sample code is dangerous and must be replaced by something else.

MOVING THE RESULT

Let us modify the code example a little by introducing std::move:

```
1 #include <utility>
2
3 matrix matrix::operator* (matrix const & other) const
4 {
5    // snip: ... assert matrix sizes are compatible ...
6    matrix result(rows(), other.columns());
7    // snip: ... compute and store matrix elements ...
8    return std::move(result);
9 }
```

If one would return the result with return result;, the compiler would actually return a new copy of matrix. The copy is created by the copy constructor:

```
1 // declaration of copy constructor
2 matrix::matrix(matrix const & source);
```

The useful utility function std::move instead triggers that the returned matrix object is constructed with a different constructor, a so-called move constructor:

```
1 // declaration of move constructor
2 matrix::matrix(matrix && source);
```

ADDING A MOVE CONSTRUCTOR

This move constructor may modify source as it pleases, as long as source remains in a valid state (i.e., it can be destroyed without problems). If no special move constructor exists, return std::move(result); will call the copy constructor. Move constructors are already provided for the containers in C++11′s standard template library. Hence, the matrix class can be easily extended with a move constructor:

```
1  class matrix {
2  public:
3      matrix(matrix && source);
4      // snip: ... other member functions ...
5  private:
6      std::size_t columns_;
7      std::size_t rows_;
8      std::vector<double> elements_;
9  };
10
11 matrix::matrix(matrix && source) :
12     columns_(source.columns_),
13     rows_(source.rows_),
```

```
14    elements_(std::move(source.elements_))
15 {
16 }
```

CONCLUSION

All in all, the use of std::move in conjunction with a move constructor allowed us to write clean, efficient, and safe code. In case of exceptions, the destructor of matrix will automatically be called. No user will ever have to think about how to use our code correctly. All she has to keep in mind is that matrix multiplication is not commutative…

” http://clean-cpp.org/std_move-it/

ON THE SUPERFLUOUSNESS OF STD::MOVE

“During my presentation of "Universal References in C++11" at C++ and Beyond 2012, I gave the advice to apply `std::move` to rvalue reference parameters and `std::forward` to universal reference parameters. In this post, I'll follow the convention I introduced in that talk of using *RRef* for "rvalue reference" and *URef* for "universal reference."

Shortly after I gave the advice mentioned above, an attendee asked what would happen if `std::forward` were applied to an RRef instead of `std::move`. The question took me by surprise. I was so accustomed to the RRef-implies-`std::move` and URef-implies-`std::forward` convention, I had not thought through the implications of other possibilities. The answer I offered was that I wasn't sure what would happen, but I didn't really care, because even if using `std::forward` with an RRef would work, it would be unidiomatic and hence potentially confusing to readers of the code.

The question has since been repeated on stackoverflow, and I've also received it from attendees of other recent presentations I've given. It's apparently one of those obvious questions I simply hadn't considered. It's time I did.

`std::move` unconditionally casts its argument to an rvalue. Its implementation is far from transparent, but the pseudocode is simple:

?

```
// pseudocode for for std::move

template<typename T>

T&& std::move(T&& obj)

{

  return (T&&)obj;        // return obj as an rvalue
```

```
}
```

`std::forward` is different. It casts its argument, which is assumed to be a reference to a deduced type, to an rvalue only if the object to which the reference is bound is an rvalue. (Yes, that's a mouthful, but that's what `std::forward` does.) Whether the object to which the reference is bound is an rvalue is determined by the deduced type. If the deduced type is a reference, the referred-to object is an lvalue. If the deduced type is a non-reference, the referred-to object is an rvalue. (This explanation assumes a lot of background on how type deduction works for universal reference parameters, but that's covered in the talk as well as in its printed manifestations in *Overload* and at ISOcpp.org.)

As with `std::move`, `std::forward`'s implementation is rather opaque, but the pseudocode isn't too bad:

?

```
// pseudocode for for std::forward

template<typename T>

T&& std::forward(T&& obj)

{

  if (T is a reference)

    return (T&)obj;           // return obj as an lvalue

  else

    return (T&&)obj;         // return obj as an rvalue

}
```

Even the pseudocode makes sense only when you understand that (1) if `T` is a reference, it will be an lvalue reference and (2) thanks to reference collapsing, `std:forward`'s return type will turn into`T&` when `T` is an lvalue reference. Again, this is covered in the talk and elsewhere.

Now we can answer the question of what would happen if you used `std::forward` on an RRef. Consider a class `Widget` that offers a move constructor and that contains a `std::string` data member:

?

```
class Widget {

public:

  Widget(Widget&& rhs);       // move constructor



private:

  std::string s;
```

```
};
```

The way you're supposed to implement the move constructor is:

?

```
Widget::Widget(Widget&& rhs)
: s(std::move(rhs.s))
{}
```

Per convention, `std::move` is applied to the RRef `rhs` when initializing the `std:string`. If we used `std::forward`, the code would look like this:

?

```
Widget::Widget(Widget&& rhs)
: s(std::forward<std::string>(rhs.s)
{}
```

You can't see it in the pseudocode for `std::forward`, but even though it's a function template, the functions it generates don't do type deduction. Preventing such type deduction is one of the things that make `std::forward`'s implementation less than transparent. Because there is no type deduction with `std::forward`, the type argument `T` must be specified in the call. In contrast, `std::move`does do type deduction, and that's why in the `Widget` move constructor, we say "`std::move(rhs.s)`", but "`std::forward<std::string>(rhs.s)`".

In the call "`std::forward<std::string>(rhs.s)`", the type `std::string` is a non-reference. As a result, `std::forward` returns its argument as an rvalue, which is exactly what `std::move` does. That answers the original question. If you apply `std::forward` to an rvalue reference instead of `std::move`, you get the same result. `std::forward` on an rvalue reference does the same thing as `std::move`.

Now, to be fully accurate, this assumes that you follow the rules and pass to `std::forward` the type of the RRef without its reference-qualifiers. In the `Widget` constructor, for example, my analysis assumes that you pass `std::forward<std::string>(rhs.s)`. If you decide to be a rebel and write the call like this,

?

```
std::forward<std::string&>(rhs.s)
```

`rhs.s` would be returned as an lvalue, which is not what `std::move` does. It also means that the `std::string` data member in `Widget` would be copy-initializd instead of move-initialized, which would defeat the purpose of writing a move constructor.

If you decide to be a smart aleck and write this,

?

```
std::forward<std::string&&>(rhs.s)
```

the reference-collapsing rules will see that you get the same behavior as `std::move`, but with any luck, your team lead will shift you to development in straight C, where you'll have to content yourself with writing bizarre macros.

Oh, and if you make the mistake of writing the move constructor like this,

?

```
Widget::Widget(Widget&& rhs)
: s(std::forward<Widget>(rhs.s))
{}
```

which, because I'm so used to passing `std::forward` the type of the function parameter, is what I did when I initially wrote this article, you'll be casting one type (in this case, a `std::string`) to some other unrelated type (here, a `Widget`), and I can only hope the code won't compile. I find the idea so upsetting, I'm not even going to submit it to a compiler.

Summary time:

- If you use `std::forward` with an RRef instead of `std::move`, and if you pass the correct type to `std::forward`, the behavior will be the same as `std::move`. In this sense, `std::move` is superfluous.
- If you use `std::forward` instead of `std::move`, you have to pass a type, which opens the door to errors not possible with `std::move`.
- Using `std::forward` requires more typing than `std::move` and yields source code with more syntactic noise.
- Using `std::forward` on an RRef is contrary to established C++11 idiom and contrary to the design of move semantics and perfect forwarding. It can work, sure, but it's still an anathema.

" http://scottmeyers.blogspot.fi/2012/11/on-superfluousness-of-stdmove.html

DIFFERENCE BETWEEN STD::MOVE AND STD::FORWARD

"In really simple terms, `std::move` returns an argument as an rvalue reference while `std::forward` returns either an lvalue reference or an rvalue reference based on how the argument was passed in to the current function. While this is fairly obvious once you get the hang of it, it can be quite tricky to grasp earlier on. The easiest way to see how they work differently is to try some code such as the following:

```
void Show(int&&)
{
```

```
        cout << "int&& called" << endl;
}

void Show(int&)
{
        cout << "int& called" << endl;
}

template<typename T> void Foo(T&& x)
{
        cout << "straight: ";
        Show(x);

        cout << "move: ";
        Show(move(x));

        cout << "forward: ";
        Show(forward<T>(x));

        cout << endl;
}

int _tmain(void)
{
        Foo(10);

        int x=10;
        Foo(x);
        return 0;
}
```

The output of the above code is:

```
straight: int& called
move: int&& called
forward: int&& called

straight: int& called
move: int&& called
forward: int& called
```

When `Foo` is called with an rvalue, `T&&` is deduced as `int&&`. But the variable itself is an lvalue within that function. So the straight call calls the `Show(int&)` overload – no surprises there. Both the move-call and the forward-call calls `Show(int&&)` as expected. In the 2nd case, `Foo` is called with an lvalue, and again the straight call will go to `Show(int&)` while the move-call will go to `Show(int&&)`.

Now here's where `forward` comes in handy, it will go to `Show(int&)` because that's what `T` has been deduced to in this instance of the function call (perfect forwarding). If you are wondering why, the collapsing rule for `T&&` collapses it to `T&` during template argument type deduction.

" http://voidnish.wordpress.com/2012/05/22/difference-between-stdmove-and-stdforward/

A BRIEF INTRODUCTION TO RVALUE REFERENCES

"INTRODUCTION

This document gives a quick tour of the new C++ language feature RVALUE REFERENCE. It is a brief tutorial, rather than a complete reference. For details, see these references.

THE RVALUE REFERENCE

An RVALUE REFERENCE is a compound type very similar to C++'s traditional reference. To better distinguish these two types, we refer to a traditional C++ reference as an LVALUE REFERENCE. When the term reference is used, it refers to both kinds of reference: lvalue reference and rvalue reference.

An lvalue reference is formed by placing an & after some type.

```
A a;
A& a_ref1 = a;  // an lvalue reference
```

An rvalue reference is formed by placing an && after some type.

```
A a;
A&& a_ref2 = a;  // an rvalue reference
```

An rvalue reference behaves just like an lvalue reference except that it can bind to a temporary (an rvalue), whereas you can not bind a (non const) lvalue reference to an rvalue.

```
A&  a_ref3 = A();  // Error!
A&& a_ref4 = A();  // Ok
```

Question: Why on Earth would we want to do this?!

It turns out that the combination of rvalue references and lvalue references is just what is needed to easily code MOVE SEMANTICS. The rvalue reference

can also be used to achieve perfect forwarding, a heretofore unsolved problem in C++. From a casual programmer's perspective, what we get from rvalue references is more general and better performing libraries.

MOVE SEMANTICS

ELIMINATING SPURIOUS COPIES

Copying can be expensive. For example, for `std::vectors`, `v2=v1` typically involves a function call, a memory allocation, and a loop. This is of course acceptable where we actually need two copies of a `vector`, but in many cases, we don't: We often copy a `vector` from one place to another, just to proceed to overwrite the old copy. Consider:

```
template <class T> swap(T& a, T& b)
{
    T tmp(a);   // now we have two copies of a
    a = b;      // now we have two copies of b
    b = tmp;    // now we have two copies of tmp (aka a)
}
```

But, we didn't want to have ANY copies of a or b, we just wanted to swap them. Let's try again:

```
template <class T> swap(T& a, T& b)
{
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

This `move()` gives its target the value of its argument, but is not obliged to preserve the value of its source. So, for a `vector`, `move()` could reasonably be expected to leave its argument as a zero-capacity `vector` to avoid having to copy all the elements. In other words, move is a potentially destructive read.

In this particular case, we could have optimized `swap` by a specialization. However, we can't specialize every function that copies a large object just before it deletes or overwrites it. That would be unmanageable.

The first task of rvalue references is to allow us to implement `move()` without verbosity, or runtime overhead.

MOVE

The move function really does very little work. All `move` does is accept either an lvalue or rvalue argument, and return it as an rvalue WITHOUT triggering a copy construction:

```
template <class T>
typename remove_reference<T>::type&&
move(T&& a)
{
    return a;
}
```

It is now up to client code to overload key functions on whether their argument is an lvalue or rvalue (e.g. copy constructor and assignment operator). When the argument is an lvalue, the argument must be copied from. When it is an rvalue, it can safely be moved from.

OVERLOADING ON LVALUE / RVALUE

Consider a simple `handle` class that owns a resource and also provides copy semantics (copy constructor and assignment). For example a `clone_ptr` might own a pointer, and call `clone()` on it for copying purposes:

```
template <class T>
class clone_ptr
{
private:
    T* ptr;
public:
    // construction
    explicit clone_ptr(T* p = 0) : ptr(p) {}

    // destruction
    ~clone_ptr() {delete ptr;}

    // copy semantics
    clone_ptr(const clone_ptr& p)
        : ptr(p.ptr ? p.ptr->clone() : 0) {}

    clone_ptr& operator=(const clone_ptr& p)
    {
        if (this != &p)
        {
            delete ptr;
            ptr = p.ptr ? p.ptr->clone() : 0;
        }
        return *this;
    }

    // move semantics
    clone_ptr(clone_ptr&& p)
        : ptr(p.ptr) {p.ptr = 0;}
```

```
    clone_ptr& operator=(clone_ptr&& p)
    {
        std::swap(ptr, p.ptr);
        return *this;
    }

    // Other operations
    T& operator*() const {return *ptr;}
    // ...
};
```

Except for the highlighted move semantics section above, `clone_ptr` is code that you might find in today's books on C++. Clients of `clone_ptr` might use it like so:

```
clone_ptr p1(new derived);
// ...
clone_ptr p2 = p1;  // p2 and p1 each own their own pointer
```

Note that copy constructing or assigning a `clone_ptr` is a relatively expensive operation. However when the source of the copy is known to be an rvalue, one can avoid the potentially expensive`clone()` operation by pilfering the rvalue's pointer (no one will notice!). The MOVE CONSTRUCTOR above does exactly that, leaving the rvalue in a default constructed state. The MOVE ASSIGNMENT operator simply swaps state with the rvalue.

Now when code tries to copy an rvalue `clone_ptr`, or if that code explicitly gives permission to consider the source of the copy an rvalue (using `std::move`), the operation will execute much faster.

```
clone_ptr p1(new derived);
// ...
clone_ptr p2 = std::move(p1);  // p2 now owns the pointer instead of p1
```

For classes made up of other classes (via either containment or inheritance), the move constructor and move assignment can easily be coded using the `std::move` function:

```
class Derived
    : public Base
{
    std::vector<int> vec;
    std::string name;
    // ...
public:
    // ...
    // move semantics
    Derived(Derived&& x)              // rvalues bind here
        : Base(std::move(x)),
          vec(std::move(x.vec)),
          name(std::move(x.name)) { }
```

```
    Derived& operator=(Derived&& x)   // rvalues bind here
    {
        Base::operator=(std::move(x));
        vec  = std::move(x.vec);
        name = std::move(x.name);
        return *this;
    }
    // ...
};
```

Each subobject will now be treated as an rvalue when binding to the subobject's constructors and assignment operators. `std::vector` and `std::string` have move operations coded (just like our eariler `clone_ptr` example) which will completely avoid the tremendously more expensive copy operations.

Note above that the argument `x` is treated as an lvalue internal to the move functions, even though it is declared as an rvalue reference parameter. That's why it is necessary to say `move(x)` instead of just `x` when passing down to the base class. This is a key safety feature of move semantics designed to prevent accidently moving twice from some named variable. All moves occur only from rvalues, or with an explicit cast to rvalue such as using `std::move`. If you have a name for the variable, it is an lvalue.

Question: What about types that don't own resources? (E.g. `std::complex`?)

No work needs to be done in that case. The copy constructor is already optimal when copying from rvalues.

MOVABLE BUT NON-COPYABLE TYPES

Some types are not amenable to copy semantics but can still be made movable. For example:

- `fstream`
- `unique_ptr` (non-shared, non-copyable ownership)
- A type representing a thread of execution

By making such types movable (though still non-copyable) their utility is tremendously increased. Movable but non-copyable types can be returned by value from factory functions:

```
ifstream find_and_open_data_file(/* ... */);
...
ifstream data_file = find_and_open_data_file(/* ... */);  // No copies!
```

In the above example, the underlying file handle is passed from object to object, as long as the source `ifstream` is an rvalue. At all times, there is still only one underlying file handle, and only one ifstream owns it at a time.

Movable but non-copyable types can also safely be put into standard containers. If the container needs to "copy" an element internally (e.g. `vector` reallocation) it will move the element instead of copying it.

```
vector<unique_ptr<base>> v1, v2;
v1.push_back(unique_ptr(new derived()));  // ok, moving, not copying
...
v2 = v1;              // Compile time error.  This is not a copyable type.
v2 = move(v1);        // Move ok.  Ownership of pointers transferred to
v2.
```

Many standard algorithms benefit from moving elements of the sequence as opposed to copying them. This not only provides better performance (like the improved `std::swap` implementation described above), but also allows these algorithms to operate on movable but non-copyable types. For example the following code sorts a `vector<unique_ptr<T>>` based on comparing the pointed-to types:

```
struct indirect_less
{
    template <class T>
    bool operator()(const T& x, const T& y)
        {return *x < *y;}
};
...
std::vector<std::unique_ptr<A>> v;
...
std::sort(v.begin(), v.end(), indirect_less());
```

As sort moves the `unique_ptr`'s around, it will use `swap` (which no longer requires `Copyability`) or move construction / move assignment. Thus during the entire algorithm, the invariant that each item is owned and referenced by one and only one smart pointer is maintained. If the algorithm were to attempt a copy (say, by programming mistake) a compile time error would result.

PERFECT FORWARDING

Consider writing a generic factory function that returns a `std::shared_ptr` for a newly constructed generic type. Factory functions such as this are valuable for encapsulating and localizing the allocation of resources. Obviously, the factory function must accept exactly the same sets

of arguments as the constructors of the type of objects constructed. Today this might be coded as:

```
template <class T>
std::shared_ptr<T>
factory()   // no argument version
{
    return std::shared_ptr<T>(new T);
}

template <class T, class A1>
std::shared_ptr<T>
factory(const A1& a1)   // one argument version
{
    return std::shared_ptr<T>(new T(a1));
}

// all the other versions
```

In the interest of brevity, we will focus on just the one-parameter version. For example:

```
std::shared_ptr<A> p = factory<A>(5);
```

Question: What if T's constructor takes a parameter by non-const reference?

In that case, we get a compile-time error as the const-qualifed argument of the factory function will not bind to the non-const parameter of T's constructor.

To solve that problem, we could use non-const parameters in our factory functions:

```
template <class T, class A1>
std::shared_ptr<T>
factory(A1& a1)
{
    return std::shared_ptr<T>(new T(a1));
}
```

This is much better. If a const-qualified type is passed to the `factory`, the const will be deduced into the template parameter (`A1` for example) and then properly forwarded to T's constructor. Similarly, if a non-const argument is given to `factory`, it will be correctly forwarded to `T`'s constructor as a non-const. Indeed, this is precisely how forwarding applications are coded today (e.g. `std::bind`).

However, consider:

```
std::shared_ptr<A> p = factory<A>(5);       // error
A* q = new A(5);                        // ok
```

This example worked with our first version of `factory`, but now it's broken: The "5" causes the `factory` template argument to be deduced as `int&` and subsequently will not bind to the rvalue "5". Neither solution so far is right. Each breaks reasonable and common code.

Question: What about overloading on every combination of `A1&` and `const A1&`?

This would allow us to handle all examples, but at a cost of an exponential explosion: For our two-parameter case, this would require 4 overloads. For a three-parameter `factory` we would need 8 additional overloads. For a four-parameter `factory` we would need 16, and so on. This is not a scalable solution.

Rvalue references offer a simple, scalable solution to this problem:

```
template <class T, class A1>
std::shared_ptr<T>
factory(A1&& a1)
{
    return std::shared_ptr<T>(new T(std::forward<A1>(a1)));
}
```

Now rvalue arguments can bind to the `factory` parameters. If the argument is const, that fact gets deduced into the `factory` template parameter type.

Question: What is that `forward` function in our solution?

Like move, `forward` is a simple standard library function used to express our intent directly and explicitly, rather than through potentially cryptic uses of references. We want to forward the argument `a1`, so we simply say so.

Here, `forward` preserves the lvalue/rvalue-ness of the argument that was passed to `factory`. If an rvalue is passed to factory, then an rvalue will be passed to `T`'s constructor with the help of the forward function. Similarly, if an lvalue is passed to `factory`, it is forwarded to `T`'s constructor as an lvalue.

The definition of forward looks like this:

```
template <class T>
struct identity
{
    typedef T type;
};

template <class T>
T&& forward(typename identity<T>::type&& a)
```

```
{
    return a;
}
```

” http://www.artima.com/cppsource/rvalue.html

THE DRAWBACKS OF IMPLEMENTING MOVE ASSIGNMENT IN TERMS OF SWAP

“More and more, I bump into people who, by default, want to implement move assignment in terms of swap. This disturbs me, because (1) it's often a pessimization in a context where optimization is important, and (2) it has some unpleasant behavioral implications as regards resource management.

Let's consider a simplified case of a container that contains a pointer to its contents, which are stored on the heap. I'm using a raw pointer, because I don't want to abstract anything away through the use of smart pointers.

?

```
class Container {
public:
  Container& operator=(Container&& rhs);

private:
  int *pData;           // assume points to an array
};
```

Implementing the move assignment operator using std::swap, the code looks like this:

?

```
Container& Container::operator=(Container&& rhs)
{
  std::swap(pData, rhs.pData);
  return *this;
}
```

Swapping two pointers calls for three pointer assignments,

?

```
template<typename T>
```

```
void swap(T& lhs, T& rhs)

{

  auto temp(lhs);

  lhs = std::move(rhs);

  rhs = std::move(temp);

}
```

so the cost of implementing move assignment using swap is three pointer assignments.

However, if we assume that an empty container has a null pData pointer, move assignment can be implemented using only 2 pointer assignments:

?

```
Container& Container::operator=(Container&& rhs)

{

  delete [] pData;



  pData = rhs.pData;

  rhs.pData = nullptr;



  return *this;

}
```

I'm ignoring the delete for now, but we'll get back to it later. At this point, I want to observe that non-swap move assignment performs only 2/3rds the assignments of its swap-based cousin. That's important, because move operations should typically be as efficient as possible. Remember that they're optimizations of copy operations, and if you're not concerned about their efficiency, why not just omit them and let rvalues be copied? My feeling is that the fact that a class author went to the trouble of adding support for move operations is a sign that the author perceives the class to be one where speed is important. If that's the case, it seems unreasonable to pay a premium to put the object being moved from into the state of the target of the assignment when it's cheaper to put the object into a different, but equally valid (typically default-constructed), state. After all, the semantics of move assignment typically don't specify the state of the source object of the move after the assignment has been performed, so if callers can't rely on it having the state of the target object before the assignment, why pay for it?

But back to the delete. Regardless of which move assignment implementation is used, the delete will eventually be performed. If it doesn't take place in the move assignment operator for the object that's the target of the move assignment, it'll probably occur in the destructor of the object that's the source of the move assignment. The cost therefore doesn't vary between the implementations, but *when* you incur that cost does, and that can be important.

Suppose that Container objects typically use a lot of memory--enough that you have to worry about it. Now consider the following scenario:

?

```
{
  Container c1, c2;

  ...

  c1 = std::move(c2);

  ...

}
```

In the non-swap implementation of move assignment, c1's memory is released at the point of the assignment, but in the swap version, that memory becomes associated with c2, and the memory may not be released until the end of the scope. That might well surprise the caller, who could hardly be blamed for thinking that when an assignment was made to c1, c1's old resources would be released. And it could certainly increase the maximum amount of memory used by the application at any given time.

In my experience, the impact of move-assignment-by-swap on the timing of resource release isn't as well known as it should be, even though the issue was well described many years ago (e.g., Thomas Becker here and David Abrahams here).

Now, bear in mind that I said at the outset that I was disturbed by people who, *by default*, want to implement move assignment in terms of swap. I have no issue with developers who, consciously aware of the performance and behavioral implications of move-assignment-via-swap, choose to use it anyway. For some types, it may be a perfectly valid implementation choice. My concern is that it's gaining a reputation as *the* way to implement move assignment, and I don't think that's good for C++.

Am I mis-analyzing the situation?" http://scottmeyers.blogspot.fi/2014/06/the-drawbacks-of-implementing-move.html

UNDERSTAND STD::MOVE AND STD::FORWARD.

"It's useful to approach std::move and std::forward in terms of what they *don't* do.

std::move doesn't move anything. std::forward doesn't forward anything. At runtime,

neither does anything at all. They generate no executable code. Not a single

byte.

std::move and std::forward are merely functions (actually function templates)

that perform casts. std::move unconditionally casts its argument to an rvalue, while

std::forward performs this cast only if a particular condition is fulfilled. That's it.

The explanation leads to a new set of questions, but, fundamentally, that's the complete story.

In the Annotation constructor's member initialization list, the result of std::move(text) is an rvalue of type const std::string. That rvalue can't be passed to std::string's move constructor, because the move constructor takes an rvalue reference to a *non-const* std::string. The rvalue can, however, be passed to the copy constructor, because an lvalue-reference-to-const is permitted to bind to a const rvalue. The member initialization therefore invokes the *copy* constructor in std::string, even though text has been cast to an rvalue! Such behavior is essential to maintaining const-correctness. Moving a value out of an object generally modifies the object, so the language should not permit const objects to be passed to functions (such as move constructors) that could modify them.

There are two lessons to be drawn from this example. First, don't declare objects const if you want to be able to move from them. Move requests on const objects are silently transformed into copy operations. Second, std::move not only doesn't actually move anything, it doesn't even guarantee that the object it's casting will be eligible to be moved. The only thing you know for sure about the result of applying std::move to an object is that it's an rvalue.

The story for std::forward is similar to that for std::move, but whereas std::move *unconditionally* casts its argument to an rvalue, std::forward does it only under certain conditions. std::forward is a *conditional* cast. To understand when it casts and when it doesn't, recall how std::forward is typically used. The most common scenario is a function template taking a universal reference parameter that is to be passed to another function:

```
void process(const Widget& lvalArg); // process lvalues
void process(Widget&& rvalArg); // process rvalues
template<typename T> // template that passes
```

```
void logAndProcess(T&& param) // param to process
{
auto now = // get current time
std::chrono::system_clock::now();
makeLogEntry("Calling 'process'", now);
process(std::forward<T>(param));
}
```

Consider two calls to logAndProcess, one with an lvalue, the other with an rvalue:

```
Widget w;
logAndProcess(w); // call with lvalue
logAndProcess(std::move(w)); // call with rvalue
```

Inside logAndProcess, the parameter param is passed to the function process. pro cess is overloaded for lvalues and rvalues. When we call logAndProcess with an lvalue, we naturally expect that lvalue to be forwarded to process as an lvalue, and when we call logAndProcess with an rvalue, we expect the rvalue overload of pro cess to be invoked.

But param, like all function parameters, is an lvalue. Every call to process inside logAndProcess will thus want to invoke the lvalue overload for process. To prevent this, we need a mechanism for param to be cast to an rvalue if and only if the argument with which param was initialized—the argument passed to logAndProcess—was an rvalue. This is precisely what std::forward does. That's why std::forward is a *conditional* cast: it casts to an rvalue only if its argument was initialized with an rvalue.

You may wonder how std::forward can know whether its argument was initialized with an rvalue. In the code above, for example, how can std::forward tell whether param was initialized with an lvalue or an rvalue? The brief answer is that that information is encoded in logAndProcess's template parameter T. That parameter is passed to std::forward, which recovers the encoded information.

std::move's attractions are convenience, reduced likelihood of error, and greater clarity.

More importantly, the use of std::move conveys an unconditional cast to an rvalue, while the use of std::forward indicates a cast to an rvalue only for references to which rvalues have been bound. Those are two very different actions. The first one typically sets up a move, while the second one just passes—*forwards*—an object to another function in a way that retains its original lvalueness or rvalueness. Because these actions are so different, it's good that we have two different functions (and function names) to distinguish them."

C++11: PERFECT FORWARDING - EXPLAINED

"Consider this function template `invoke` that invokes the function/functor/lambda expression passed as argument passing it the two extra arguments given:

```
1   #include <iostream>
2   #include <string>
3
4   using namespace std;
5
6   void sum(int a, int b)
7   {
8       cout << a + b << endl;
9   }
10
11  void concat(const string& a, const string& b)
12  {
13      cout << a + b << endl;
14  }
```

```
15
16  template <typename PROC, typename A, typename B>
17  void invoke(PROC p, const A& a, const B& b)
18  {
19      p(a, b);
20  }
21
22  int main()
23  {
24      invoke(sum, 10, 20);
25      invoke(concat, "Hello ", "world");
26      return 0;
27  }
```

Nice, it works as expected and the result is:

```
30
Hello world
```

The problem with my implementation is that it only works with arguments passed as constant references, so if I would like to invoke this function:

```
1  void successor(int a, int& b)
2  {
3      b = a + 1;
4  }
```

with this call:

```
1  int s = 0;
2  invoke(successor, 10, s);
3  cout << s << endl;
```

clang returns me this error in the `invoke` implementation:

```
Binding of reference to type 'int' to a value of type 'const int' drops
qualifiers
```

This error occurs because the second argument of my `successor` function is not a const-reference.

Before C++11 the only way to deal with this problem was to create a set of overloads containing all the possible combinations of const, non-const references in the methods, something like this:

```
1   #include <iostream>
2   #include <string>
3
4   using namespace std;
5
6   void sum(int a, int b)
7   {
8       cout << a + b << endl;
9   }
10
11  void concat(const string& a, const string& b)
12  {
13      cout << a + b << endl;
14  }
15
16  void successor(int a, int& b)
17  {
18      b = ++a;
19  }
20
```

```
21  template <typename PROC, typename A, typename B>
22  void invoke(PROC p, const A& a, const B& b)
23  {
24      p(a, b);
25  }
26
27  template <typename PROC, typename A, typename B>
28  void invoke(PROC p, const A& a, B& b)
29  {
30      p(a, b);
31  }
32
33  template <typename PROC, typename A, typename B>
34  void invoke(PROC p, A& a, const B& b)
35  {
36      p(a, b);
37  }
38
39  template <typename PROC, typename A, typename B>
40  void invoke(PROC p, A& a, B& b)
41  {
42      p(a, b);
43  }
44
45  int main()
46  {
47      invoke(sum, 10, 20);
        invoke(concat, "Hello", "world");
```

```
48        int s = 0;
49        invoke(successor, 10, s);
50        cout << s << endl;
51        return 0;
52    }
53
```

Notice I had to implement four overloads for my `invoke` function template because I need to forward two parameters to the function passed in P.

If I would have more parameters, this would be unmaintainable (for N arguments, I would need to have 2^N overloads).

C++11 lets us perform **perfect forwarding**, which means that we can forward the parameters passed to a function template to another function call inside it without losing their own qualifiers (const-ref, ref, value, rvalue, etc.).

In order to use this technique, my `invoke` function template must have arguments passed as rvalue references and I need to use the `std::forward` function template that is in charge of performing the type deduction and forward the arguments to the invoked function with its own reference, const-reference or rvalue-reference qualifiers. Look into my code modified using this technique:

```
1     #include <iostream>
2     #include <string>
3
4     using namespace std;
5
6     void sum(int a, int b)
7     {
8         cout << a + b << endl;
9     }
10
```

```
11  void concat(const string& a, const string& b)
12  {
13      cout << a + b << endl;
14  }
15
16  void successor(int a, int& b)
17  {
18      b = ++a;
19  }
20
21  template <typename PROC, typename A, typename B>
22  void invoke(PROC p, A&& a, B&& b)
23  {
24      p(std::forward<A>(a), std::forward<B>(b));
25  }
26
27  int main()
28  {
29      invoke(sum, 10, 20);
30      invoke(concat, "Hello", "world");
31      int s = 0;
32      invoke(successor, 10, s);
33      cout << s << endl;
34      return 0;
35  }
```

What I would want to use this technique for?

Consider having these classes:

```
struct A { };
struct B { int a; int b; };
struct C { int a; int b; string c; };
struct D
{
public:
    D(int a) {}
};
```

I want to have a `factory` function template that will let me run this code:

```
 int main()
 {
     auto a = factory<A>();
     auto b = factory<B>(10, 20);
     auto c = factory<C>(30, 40, "Hello");
     auto d = factory<D>(10);

     cout << c->c << endl;

 }
```

As you can see, the `factory` function template must be a variadic-template based function, so, a solution would be this one:

```
template <typename T, typename ... ARGS>
unique_ptr<T> factory(const ARGS&... args)
{
    return unique_ptr<T>(new T { args... });
```

```
5   }
```

Nice, and it works as expected. But the problem of losing qualifiers arises again. What if I have something like:

```
1   struct E
2   {
3       int& e;
4       E(int& e) : e(e) { e++; }
5   };
```

If you try to use this struct in this way:

```
1   int main()
2   {
3       int x = 2;
4       auto e = factory<E>(x);
5       cout << x << endl;
6   }
```

You will get an error because “x” is an “int&”, not a “const int&”.

Fortunately, **perfect forwarding** is the way to go:

```
1    #include <iostream>
2    #include <string>
3
4    using namespace std;
5
6    struct A { };
7    struct B { int a; int b; };
```

```
8   struct C { int a; int b; string c; };
9   struct D
10  {
11  public:
12      D(int a) {}
13  };
14
15  struct E
16  {
17      int& e;
18      E(int& e) : e(e) { e++; }
19  };
20
21  template <typename T, typename ... ARGS>
22  unique_ptr<T> factory(ARGS&&... args)
23  {
24      return unique_ptr<T>(new T { std::forward<ARGS>(args)... });
25  }
26
27  int main()
28  {
29      auto a = factory<A>();
30      auto b = factory<B>(10, 20);
31      auto c = factory<C>(30, 40, "Hello");
32      auto d = factory<D>(10);
33
34      int x = 2;
        auto e = factory<E>(x);
```

```
35          cout << x << endl;
36      }
37
```

Perfect forwarding also helps to avoid to write several overloads of functions to support move semantics and copy semantics.

For example, look at this code:

```
1   #include <iostream>
2
3   using namespace std;
4
5   struct X
6   {
7       X() { cout << "ctor" << endl; }
8       X(const X&) { cout << "copy ctor" << endl; }
9       X(X&&) { cout << "move ctor" << endl; }
10  };
11
12  struct Wrapper
13  {
14      X w;
15
16      Wrapper(const X& w) : w(w) { }
17  };
18
19  int main()
20  {
```

```
21        Wrapper w1(X { });
22        cout << "***" << endl;
23        X y;
24        Wrapper w2(y);
25    }
```

The output is:

```
ctor
copy ctor
***
ctor
copy ctor
```

The little problem here is that when constructing `w1`, the instance of `X` is a temporary instance that will be destroyed immediately after invoking the `Wrapper` constructor. For this case, invoking the move constructor should be better (performance-wise).

So, we can create a second constructor and modify our `Wrapper` class to be like this one:

```
1   struct Wrapper
2   {
3       X w;
4
5       Wrapper(const X& w) : w(w) { }
6       Wrapper(X&& w): w(std::move(w)) { } //Adding a overloaded constructor
7   };
```

Cool! It works as expected:

```
ctor
move ctor
***
ctor
copy ctor
```

When constructing `w1`, the `Wrapper` constructor that has a rvalue reference parameter is used. In my example with a constructor with one parameter, this work nice and is easy to maintain, but with constructors or functions that have a lot of parameters, the same maintainability issues shown above raise again. So, we can one more time rely on **perfect forwarding** to let the compiler decide what constructor to use and generate the required (and desired) code:

```
1   struct Wrapper
2   {
3       X w;
4
5       template <typename Q>
6       Wrapper(Q&& w) : w(std::forward<Q>(w)) { }
7   };
```

It's nice, isn't it?

" http://oopscenities.net/2014/02/01/c11-perfect-forwarding/

C++: SMART POINTERS – EXPLAINED

C++: SMART POINTERS, PART 1

"Memory management in C is too error prone because keeping track of each bunch of bytes allocated and deallocated can be really confusing and stressing.

Although C++ has the same manual memory management than C, it provides us some additional features that let us to do this management easier:

- When an object is instantiated in the stack (e.g. `Object o;`); the C++ runtime ensure the destructor of such object is invoked when the object goes out of scope (the end of the enclosing block is reached, a premature 'return' is found or an exception is thrown); thus, releasing all memory and resources allocated for such object.
- (Ab)using the feature of operator overloading, we can create classes that simulate the behavior of the pointers. Such classes are called: **Smart pointers**.

So, smart pointers are classes (typically implemented as class templates, to make them highly reusable) that wrap a pointer and simulate its same behavior, but implement also some policies to release the objects “automatically”. The “magic” of smart pointers lies in the fact that they are always defined as stack variables, thus, the invocation of their destructor is guaranteed.

Consider this piece of code:

```
1
2
3   #include <iostream>
4
5   using namespace std;
6
7   class A
8   {
9     private:
10      int val;
11    public:
12      A(int val) : val(val) { cout << "A ctor" << endl; }
13      ~A() { cout << "A dtor" << endl; }
14      int get_value() const { return val; }
15  };
16
17  int main()
18  {
19    cout << "Enter a number: " << endl;
20    int n;
21    cin >> n;
22
23    A* a = new A(n);
24    if (n == 2)
25      return 0;
26
27    cout << "Value entered: " << a->get_value() << endl;
28    delete a;
    return 0;
}
```

The code shown above asks the user to enter a number, the user enters it and if the number is 2, the program finishes, otherwise, the program shows the number entered before exiting. If you look into the code you will realize that if the value entered equals 2, the program finishes WITHOUT calling the destructor of A. Though my example is quite trivial, forgetting to call the destructors when returning in several points in our code is one of the most common sources of memory leaks.

Smart pointers are good candidates in this point; we can write a smart pointer class template that releases the object when the smart pointer gets out of scope. To simulate the behavior of plain-old-pointers, the `operator*` and`operator->` must be overloaded: Look into this piece of code implementing my smart pointer:

```
1
2     template <typename T>
3     class ptr
4     {
5       private:
6         T* pointee;
7
8       public:
9         explicit ptr(T* pointee) : pointee(pointee) { }
10        ~ptr() { delete pointee; }
11
12        T* operator->() { return pointee; }
13        const T* operator->() const { return pointee; }
14
15        T& operator*() { return *pointee; }
16        const T& operator*() const { return *pointee; }
17    };
18
```

And now look into a modified `main()` function that uses the smart pointer instead of the normal one:

```
1
2     int main()
3     {
4       cout << "Enter a number: " << endl;
5       int n;
6       cin >> n;
7
8       ptr<A> a(new A(n));
9       if (n == 2)
10        return 0;
11
12      cout << "Value entered: " << a->get_value() << endl;
13      return 0;
14    }
15
```

My variable `a` is defined as a smart pointer and will hold the pointer to a new instance of A() allocated in the heap. As you can see, since the `ptr<a> a` declaration is a stack declaration, the C++ runtime guarantees that when the `main()` method will be finished (through the premature return of through the final one), the `a` destructor will be invoked and it will call the destructor of the pointee object.

A very similar policy is also implemented in the deprecated `std::auto_ptr<T>` and in the C++11 `std::unique_ptr<T>` smart pointers.

Other approach of smart pointers is implementing a policy that will keep the track of how many smart pointers are pointing to the same object in the heap. Using reference counting, this kind of smart pointers call the destructor of the objects when the reference count has reached 0 (meaning that there are zero smart pointers pointing to such object in the heap).

A reference counter smart pointer can be implemented in a way similar to this one:

```
1  struct refcntptrdata
2  {
3    T* pointee;
4    int refcnt;
5  };
6
7  template <typename T>
8  class refcntptr
9  {
10   private:
11     refcntptrdata<T>* data;
12
13     void release()
14     {
15       data->refcnt--;
16       if (data->refcnt == 0)
17       {
18         delete data->pointee;
19         delete data;
20       }
21     }
22
23   public:
24     explicit refcntptr(T* pointee) : data(new refcntptrdata<T>())
25     {
26       data->pointee = pointee;
27       data->refcnt = 1;
28     }
29
30     refcntptr(const refcntptr<T>& source) : data(source.data)
31     {
32       data->refcnt++;
33     }
34
     refcntptr<T>& operator=(const refcntptr<T>& source)
     {
       release();
       data = source.data;
       data->refcnt++;
     }
```

```
    ~refcntptr()
    {
      release();
    }

    T* operator->() { return data->pointee; }
    const T* operator->() const { return data->pointee; }

    T& operator*() { return *(data->pointee); }
    const T& operator*() const { return *(data->pointee); }
};
```

As you can see in my implementation, the `refcntptr<T>` object just points to a `refcntptrdata<T>` object that is the one that keeps the pointer to the pointee and the reference counter.

The nice thing on this kind of pointers is that the copy constructor is very cheap because it just increments the reference counter.

C++11 has also a smart pointer implementing this policy: `std::shared_ptr<T>`

See how this smart pointer can be used in the example below:

```
int main()
{
  refcntptr<A> a(new A(2));
  refcntptr<A> b(new A(12));
  refcntptr<A> c = a;
  refcntptr<A> d = b;
  refcntptr<A> e(new A(5));
  e = b; //A=5 should be destroyed here because no one is using it anymore
  b = a;

  cout << "A: " << a->get_value() << endl;
  cout << "B: " << b->get_value() << endl;
  cout << "C: " << c->get_value() << endl;
  cout << "D: " << d->get_value() << endl;
  cout << "E: " << (*e).get_value() << endl;

}
```

16
17

” http://oopscenities.net/2011/10/14/c-smart-pointers/

C++11: SMART POINTERS, PART 2: UNIQUE_PTR

“C++11 ships with a set of out-of-the-box smart pointers that help us to manage the memory easily.

One of those smart pointers is the `unique_ptr`.

Consider this piece of code:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
```

```
#include <iostream>
#include <ctime>

using namespace std;

class A
{
  public:
    A() { cout << "ctor invoked" << endl; }
    virtual ~A() { cout << "dtor invoked" << endl; }
    void sayHi() const { cout << "HI" << endl; }
};

class B : public A { };

void test()
{
    if (clock() % 5 == 0)
        throw std::exception();
}

int main()
{
  A* a = new A();
  A* b = new B();

  clock_t clk = clock();
  if (clk % 2 == 0)
    return -1;

  test();

  a->sayHi();

  delete b;
  delete a;
}
```

34
35
36
37

If you execute this code several times, you will get three types of output:

This one:

```
ctor invoked
ctor invoked
HI
dtor invoked
dtor invoked
```

Or this, that occurs when the number returned by clock() is even:

```
ctor invoked
ctor invoked
```

Or this, that occurs when the exception in test() is thrown:

```
ctor invoked
ctor invoked
terminate called after throwing an instance of 'std::exception'
what(): std::exception
Abort trap: 6
```

As you see, the expected behavior just occurs in the first case, in the other ones (with a premature return or when an exception is thrown), we are not invoking delete, so we are leaving memory leaks.

What would be the perfect solution? Using stack variables, but that is not always possible: Maybe we invoke a method that returns a pointer to something, or returns a pointer of a base class to be used "polymorphically" or we need to rely on polymorphism (i.e. in virtual functions).

The ugly solution? Patching our code:

```
1   int main()
2   {
3     A* a = new A();
4     A* b = new B();
5
6     clock_t clk = clock();
7     if (clk % 2 == 0)
8     {
9       delete a;
```

```
      delete b;
```

```
10        return -1;
11      }
12
13      try
14      {
15        test();
16      }
17      catch (const std::exception& ex)
18      {
19        cerr << "An expected error occurred" << endl;
20        delete a;
21        delete b;
22        return -2;
23      }
24
25      a->sayHi();
26
27      delete b;
28      delete a;
29    }
30
```

Cons:

- Ugly
- A lot of error handling code
- Hard to maintain, a lot of duplicate code

The nice way? Using `unique_ptr`.

`unique_ptr` is a smart pointer that invokes automatically the destructor of the pointee object that is wrapping, when it reaches the end of its lifetime.

That mechanism is useful in these scenarios:

- You rely on objects allocated in the freestore, but you want to have them alive just in the method you are using them.
- You have variable members of your class declared as pointers to objects allocated in the freestore. Using `unique_ptr` instead of raw pointers, avoid you to invoke explicitly their destructors in the destructor of your class AND ensures your objects will be properly released if some exception occurs while constructing the objects of your class.

So, the code I implemented above would look like this using `unique_ptr`:

```
1
2
3
4   #include <ctime>
5   #include <iostream>
6   #include <memory>
7
8   using namespace std;
9
10  class A
11  {
12    public:
13      A() { cout << "ctor invoked" << endl; }
14      virtual ~A() { cout << "dtor invoked" << endl; }
15      void sayHi() const { cout << "HI" << endl; }
16  };
17
18  class B : public A { };
19
20  void test()
21  {
22      if (clock() % 5 == 0)
23          throw std::exception();
24  }
25
26  int main()
27  {
28    unique_ptr<A> a(new A { });
29    unique_ptr<A> b(new B { });
30
31    clock_t clk = clock();
32    if (clk % 2 == 0)
33    {
34      return -1;
35    }
36
37    try
38    {
39      test();
40    }
41    catch (const std::exception& ex)
42    {
43      cerr << "An expected error occurred" << endl;
44      return -2;
45    }

    a->sayHi();
  }
```

To get the raw pointer of the pointee object inside this smart pointer, the `unique_ptr` class template implements a method called `.get()`. This method should be only used in case you are using old code and you need to pass it raw pointers instead.

`unique_ptr` deletes the copy constructor and the assignment operator but provides the move constructor and the move assignment operator to transfer the pointee to other unique_ptr instance.

Look at this piece of code:

```
1
2
3   #include <iostream>
    #include <memory>
4
5   using namespace std;
6
7   class X
8   {
9       int x;
10    public:
11      X(int x) : x(x) { cout << "ctor invoked" << endl; }
12      ~X() { cout << "dtor invoked" << endl; }
13      void sayHi() const { cout << "HI " << x << endl; }
    };
14
15  int main()
16  {
17    unique_ptr<X> a(new X { 2 });
18    // unique_ptr<X> c = a; //does not compile! no copy constructor
19    unique_ptr<X> b = std::move(a); //valid: move constructor
20    cout << a.get() << endl;
21    b->sayHi();
    }
22
```

If you uncomment the second line of code in the `main` function, your program will not compile because the copy constructor has been explicitly removed. Instead of it, the move constructor is in place. This move constructor “steals” the pointer from `a` and sets it to `b`. As you see, when `a.get()` is executed, it returns a null pointer (because the pointer was moved to `b`).

Why is that behavior good? Because in that way the smart pointer ensure us that there is just one owner of the pointed object.

`unique_ptr` is the C++11 replacement to the old `auto_ptr` that has similar behavior, but that was deprecated because it does not support move semantics (and implements a "hack" in its copy constructor that moves the pointer to the target object (what is not legal for a copy constructor).

" http://oopscenities.net/2013/04/09/smart-pointers-part-2-unique_ptr-2/

C++: SMART POINTERS, PART 3: MORE ON UNIQUE_PTR

"Ok, here I am going to write about two other features that unique_ptr has that I did not mention in my last post.

`unique_ptr` default behavior consists on take ownership of a pointer created with `new` and that would normally be released with `delete`.

That is because the main `unique_ptr` declaration is:

```
1  template <typename T, typename Deleter = std::default_delete<T>>
2  class unique_ptr;
```

The template class `default_delete` implements a function object that performs a `delete` to the object pointed by the argument passed to the function object. Something like this:

```
1
2  template <typename T>
3  class default_delete
4  {
5  public:
6      void operator()(T* obj) const
7      {
8          delete obj;
9      }
   };
```

The idea of having the "deleter" as a template argument, lets us to use a `unique_ptr` to hold a pointer to a data structure created with, say, `malloc` or to free the memory using some custom deallocator.

Let's see both cases:

1. Using with malloc

Let's create a struct that holds two integers:

```
struct my_point
{
    int x;
    int y;
};
```

I want to create one instance of such struct in the heap using malloc, to have something like this:

```


 void show_point(const my_point& o)
 {
   cout << "(" << o.x << "; " << o.y << endl;
 }

 void function_that_possibly_throws_exception()
 {
    .....
    .....
 }

 int main()
 {
   my_point* p = static_cast<my_point*>(malloc(sizeof(my_point));
   p->x = 5;
   p->y = 8;
   show_point(*p);
   function_that_possibly_throws_exception();
   free(p);
 }


```

If I want to use `unique_ptr` to make sure that `free` is always invoked no matter if an exception occurs or not or if my function contains several return points; I need to implement a “custom deleter” to replace the`std::default_delete` deleter; something like this:

```
struct my_point_deleter
{
  void operator()(my_point* p) const
  {
    free(p);
  }
};
```

Take into account that my implementation is very similar to the `default_delete` implementation but uses `free` instead of `delete`.

To use it into my code, I need to modify my main function to be similar to this one:

```
int main()
{
  unique_ptr<my_point, my_point_deleter> p(static_cast<my_point*>(malloc(sizeof(my_poi
  p->x = 5;
  p->y = 8;
  show_point(*p);
  function_that_possibly_throws_exception();
  //free(p); --> no need for free, the unique_ptr will take care of this
}

```

2. Using with a custom deallocator

For example, I want to use unique_ptr to invoke `fclose()` everytime a FILE* gets out of scope.

Look at my implementation, as you see below, you can pass the custom deleter as a function pointer in the constructor; so I can provide that to my `unique_ptr` also as a lambda expression:

```

 int main()
 {
   unique_ptr<FILE, void (*)(FILE*)> f(fopen("file.cpp", "r"),
                                       [](FILE* f)
                                       {
                                         fclose(f);
                                       });

   char aux[100];
   while (!feof(f.get())) //f.get returns me the actual plain old pointer
   {
      fgets(aux, 100, f.get());
      cout << aux;
   }

   //No need to invoke fclose() because the unique_ptr will take care of it.
 }

```

C++14 will ship with a new variadic template function called `make_unique` that will make easier to create objects in the heap and wrap them into `unique_ptr` instances.

So, consider this instantiation using `new`:

```
unique_ptr<my_point> point(new my_point { 6, 5 });
```

can be rewritten to this one, using `make_unique`:

```
1    auto point = make_unique<my_point>(6, 5);
```

Pros? As Stephan T. Lavavej states, it saves to write "my_point" twice, it is consistent with `make_shared` (I will write about it in a later post), and it hides the usage of the operator `new` (that is good because we do not want to have our programs with `new but without delete` ;)).

" http://oopscenities.net/2013/08/06/c-smart-pointers-part-3-more-on-unique_ptr/

C++: SMART POINTERS, PART 4: SHARED_PTR

"As I mentioned in other posts, C++11 brings a new set of smart pointers into C++. The most useful smart pointer is `shared_ptr`: Its memory management policy consists in counting the number of `shared_ptr` instances that refer to the same object in the heap.

For example, if you have something like this:

```
shared_ptr<int> x(new int { 6 });
shared_ptr<int> y = x;
```

Your smart pointers `x` and `y` refer to the same integer created in the heap and both store the number of smart pointers that point to the same object (in our case, 2).

If we add something like:

```
x = nullptr;
```

The number of smart pointers referring to the same object is decremented to 1 because only `y` is referring it. When `y` gets out of scope, the `shared_ptr` destructor is invoked automatically (because of RAII) and the reference counter is decremented again (this time to 0, because no `shared_ptr` is pointing anymore to the object). Since the reference counter is 0, the object (in this case, our `int { 6 }`) destructor is invoked.

`shared_ptr` solves a lot of memory management problems and render the naked pointers (e.g. DataType*) unnecessary.

Consider this example in C++03:

```
1    #include <cstdio>
2    #include <cstring>
```

```
3
4   class Integer
5   {
6       int n;
7       public:
8         Integer(int n) : n(n) { }
9         ~Integer() { printf("Deleting %d\n", n); }
10        int get() const { return n; }
11  };
12
13  int main()
14  {
15      Integer* a = new Integer(10);
16      Integer* b = new Integer(20);
17      Integer* c = a;
18      Integer* d = new Integer(30);
19      Integer* e = b;
20      a = d;
21      b = new Integer(40);
22      Integer* f = c;
23      b = f;
24
25      printf("%d\n", a->get());
26      printf("%d\n", b->get());
27      printf("%d\n", c->get());
28      printf("%d\n", d->get());
29      printf("%d\n", e->get());
        printf("%d\n", f->get());
```

```
30
31        delete a;
32        delete b;
33        delete c;
34        delete e;
35        delete f;
36    }
37
```

When it runs, it returns something like:

```
30
10
10
30
20
10
Deleting 30
Deleting 10
Deleting 10
v(663) malloc: *** error for object 0x7fb4024000e0: pointer being freed was
not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

The program contains two severe problems: Memory leaks and crashes. If you look at the result, the `Integer(40)` instance has disappeared and its destructor has never been invoked, turning it into a memory leak. Also, we created 4 instances of the `Integer` class and we are invoking `delete` with 6 variables; what occurs right there is that one object is being deleted twice, producing a crash the second time.

`shared_ptr` comes to the rescue in C++11 and turns our code into something like this:

```
1     #include <cstdio>
2     #include <cstring>
3
4     #include <memory>
5
```

```
6   using namespace std;
7
8   class Integer
9   {
10      int n;
11      public:
12        Integer(int n) : n(n) { }
13        ~Integer() { printf("Deleting %d\n", n); }
14        int get() const { return n; }
15  };
16
17  int main()
18  {
19      shared_ptr<Integer> a(new Integer{ 10 });
20      shared_ptr<Integer> b(new Integer{ 20 });
21      shared_ptr<Integer> c = a;
22      shared_ptr<Integer> d(new Integer{ 30 });
23      shared_ptr<Integer> e = b;
24      a = d;
25      b = shared_ptr<Integer>(new Integer(40));
26      shared_ptr<Integer> f = c;
27      b = f;
28
29      printf("%d\n", a->get());
30      printf("%d\n", b->get());
31      printf("%d\n", c->get());
32      printf("%d\n", d->get());
        printf("%d\n", e->get());
```

```
33        printf("%d\n", f->get());
34    }
35
```

The output is similar to this one:

```
Deleting 40
30
10
10
30
20
10
Deleting 20
Deleting 10
Deleting 30
```

Result: No crashes and all the objects are released when they are not being used anymore.

In order to hide the `operator new` and to provide an optimization while allocating the object to be shared, the variadic template function **`make_shared`** was created. It is a template function that performs three tasks:

1. Allocates contiguous memory for the object and for the reference counter. This makes the creation and destruction of objects faster because only one allocation and deallocation will be needed when creating the object to be shared and its reference counter.
2. Invokes to the constructor of the class being instantiated forwarding the arguments used when this function was invoked.
3. Returns a shared_ptr to the newly created object.

`make_shared<T>` is a variadic template function that receives as arguments, the arguments that the constructor of class T needs.

So, our `main` function will look like this:

```
1     int main()
2     {
3         auto a = make_shared<Integer>(10);
```

```
4         auto b = make_shared<Integer>(20);
5         auto c = a;
6         auto d = make_shared<Integer>(30);
7         auto e = b;
8         a = d;
9         b = make_shared<Integer>(40);
10        auto f = c;
11        b = f;
12
13        printf("%d\n", a->get());
14        printf("%d\n", b->get());
15        printf("%d\n", c->get());
16        printf("%d\n", d->get());
17        printf("%d\n", e->get());
18        printf("%d\n", f->get());
19    }
```

What other uses can we think for `shared_ptr`?

Look at this C++03 code (I am reusing the `Integer` class):

```
1     #include <cstdio>
2     #include <cstring>
3
4     #include <vector>
5
6     using namespace std;
7
8     class Integer
```

```
9   {
10      int n;
11      public:
12        Integer(int n) : n(n) { }
13        ~Integer() { printf("Deleting %d\n", n); }
14        int get() const { return n; }
15  };
16
17  Integer* get_instance(int n)
18  {
19      return new Integer(n);
20  }
21
22  int main()
23  {
24      vector<Integer*> vec;
25
26      for (int i = 0; i < 100; i++)
27          vec.push_back(get_instance(i));
28
29      int sum = 0;
30      for (vector<Integer*>::const_iterator it = vec.begin(); it != vec.end(); ++it)
31          sum += (*it)->get();
32
33      //We do something with the elements of the vector
34      printf("Sum: %d\n", sum);
35
```

```
36      //We need to release them manually
37      for (vector<Integer*>::iterator it = vec.begin(); it != vec.end(); ++it)
38          delete *it;
39  }
```

And compare it against this C++11 code:

```
1   #include <cstdio>
2   #include <cstring>
3
4   #include <vector>
5   #include <memory>
6
7   using namespace std;
8
9   class Integer
10  {
11      int n;
12      public:
13        Integer(int n) : n(n) { }
14        ~Integer() { printf("Deleting %d\n", n); }
15        int get() const { return n; }
16  };
17
18  shared_ptr<Integer> get_instance(int n)
19  {
20      return make_shared<Integer>(n);
21  }
```

```
22
23  int main()
24  {
25      vector<shared_ptr<Integer>> vec;
26
27      for (int i = 0; i < 100; i++)
28          vec.push_back(get_instance(i));
29
30      //We do something with the elements of the vector
31      int sum = 0;
32      for (auto& i : vec)
33          sum += i->get();
34
35      printf("Sum: %d\n", sum);
36  }
```

What do you see? These advantages of the C++11 version:

- You can return `shared_ptr` instances from functions. This is far better than returning a naked pointer, because when returning a naked pointer, the programmer that invokes the function does not know a priori if he will need to use free, delete, a custom deallocator or nothing after using the object pointed by the returned pointer. When returning a `shared_ptr` instance, the programmer knows that the object will be released automatically when it will not be referred anymore.
- You can store `shared_ptr` instances inside a STL container. This makes the memory management for your application easier, because you do not need to deallocate the objects manually anymore.
- Because of the automatic memory deallocation, your program is exception safe. In the C++03 version, any problem occurring before the block in charge to release the pointers would leave a lot of objects leaking in the heap.
- Though the C++11 is still using pointers, because of `make_shared` usage, the `operator new` is not used anymore.

- Not related to `shared_ptr`; but `auto` and the range-based `for` loop make your program shorter and easier to write and understand :)

I based heavily my example in an example given by Herb Sutter in his talk: "(Not your father's) C++".

Shortcomings of `shared_ptr`? It does not work for circular references. If you want to implement something where circular references exist, you need to use `std::weak_ptr`. I will write about it in a new entry.

"http://oopscenities.net/2013/10/06/smart-pointers-part-4-shared_ptr/

C++: SMART POINTERS, PART 5: WEAK_PTR

"In modern C++ applications (C++11 and later), you can replace almost all your naked pointers to `shared_ptr` and `unique_ptr` in order to have automatic resource administration in a deterministic way so you will not need (almost, again) to release the memory manually.

The "almost" means that there is one scenario where the smart pointers, specifically, the `shared_ptr` instances, will not work: When you have circular references. In this scenario, since every shared_ptr is pointing to the other one, the memory will never be released.

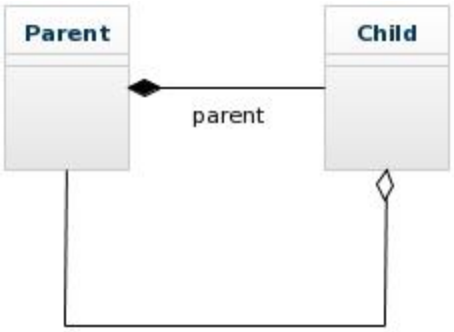
Let's look this scenario in code:

```
1   struct Child;
2
3   struct Parent
4   {
5       shared_ptr<Child> child;
6
7       ~Parent() { cout << "Bye Parent" << endl; }
8
9       void hi() const { cout << "Hello" << endl; }
10  };
11
12  struct Child
13  {
14      shared_ptr<Parent> parent;
15
16      ~Child() { cout << "Bye Child" << endl; }
17  };
18
19  int main()
20  {
21      auto parent = make_shared<Parent>();
```

```
19      auto child = make_shared<Child>();
20      parent->child = child;
21      child->parent = parent;
22      child->parent->hi();
23  }
24
25
26
```

In this program, the Parent and the Child have pointers to their respective child and parent (creating a circular reference between both); when you execute the program, you will notice the destructors are never invoked because I used shared_ptr instances.

If we model this problem, probably we can establish the Parent as “owner” of the Child object (it would mean that the child lifetime will be based on the parent lifetime) and the Child will just refer to its parent. If we diagram this using a class diagram, we can draw it like this:



If we use C++11 smart pointers to implement this kind of scenarios, we could use shared_ptr instances to implement the composition relationship and weak_ptr instances to implement the aggregation relationship:

```
1   struct Child;
2
3   struct Parent
4   {
5       shared_ptr<Child> child;
6
7       ~Parent() { cout << "Bye Parent" << endl; }
8
9       void hi() const { cout << "Hello" << endl; }
10  };
11
12  struct Child
```

```
13      weak_ptr<Parent> parent;
14
15      ~Child() { cout << "Bye Child" << endl; }
16  };
17
18  int main()
19  {
20      auto parent = make_shared<Parent>();
21      auto child = make_shared<Child>();
22      parent->child = child;
23      child->parent = parent;
24      child->parent.lock()->hi();
25  }
26
```

When you execute this program, you will notice the destructors are invoked correctly.

`weak_ptr` is a wrapper for a `shared_ptr` that does not represent ownership of the pointee object and, therefore, does not avoid the object to be released when the parent `shared_ptr` reference counter goes to 0.

In my code I am using a `weak_ptr` method called `lock`. `weak_ptr` does not implement the `operator->`, so, if you want to access the methods of the pointee object, you need to invoke the `lock(9` method. It creates a temporary`shared_ptr` instance that, since it increases the pointee reference counter, actually locks the object to be released while it is executing the invoked method.

Other nice feature of `weak_ptr` is that it can tell you if its pointee object has been already released. Look at this code:

```
1   struct A
2   {
3       int x;
4       A(int x)  : x(x) { cout << "HI" << endl; }
5       ~A() { cout << "Bye" << endl; }
6   };
7
8   weak_ptr<A> m()
9   {
10      auto a = make_shared<A>(12);
11      cout << a->x << endl;
12
13      return a;
14  }
15
16  int main()
17  {
18      auto a = m();
```

```
16
17          cout << "After m()" << endl;
18
19          if (a.expired())
20              cout << "Expired" << endl;
21          else
22              cout << a.lock()->x << endl;
23      }
24
25
26
```

The function `main()` receives an instance of a `weak_ptr`. If you look into the code, the function `m()` has a `shared_ptr` instance, but the object pointed by it lives only inside the function and it is released when returning the`weak_ptr`. Back to the `main()` function, before accessing the value of `x`, I am invoking the method `expired()` that returns true if the `weak_ptr` instance points to an already released method.

Thus, with `unique_ptr`, `shared_ptr`, and `weak_ptr`, all the scenarios where you need to use dynamically allocated memory are covered and you will not need to release your objects manually anymore.

”http://oopscenities.net/2014/08/03/c-smart-pointers-part-5-weak_ptr/

LAMBDA EXPRESSIONS

- **Avoid default capture modes**
  - Default by-reference capture can lead to dangling references
  - Default by-value capture is suspectible to dangling pointers(especially this), and it misleadingly suggests that lambdas are self-contained.
- **Use init capture to move objects into closures**
  - Use C++ 14’s init capture to move objects into closures
  - In C++11, emulate init capture via hand-written classes or std::bind
- **Use decltype on auto&& parameters to std::forward them**
- **Prefer lambdas to std:bind**
  - Lambdas are more readable, more expressive, and may be more efficient than using std::bind
  - In C++11 only, std::bind may be useful for implementing move capture or for binding objects with templatized function call operators

STD::BIND EXPLAINED (IF NEEDED)

“**Introduction**

A few days ago, I was messing with some old code of mine that uses the Boost random number library, looking at upgrading it to use the new random number features of C++11. Using the library requires the use of the **boost::bind** function, and this too can be replaced with the new C++11 **std::bind**. Unfortunately, I had not used either bind function in some time and had forgotten exactly how they work, and I had to write some simple code to remind myself. This tutorial, which explains what std::bind does, and why you may want to use, is an adaptation of the code I wrote. In order to follow it, you will need some (but not a great deal) knowledge of C++ and (if you want to compile the code) a C++ compiler that supports the C++11 <functional> facilities – I used GCC 4.6.1.

Source code for the examples is available at **https://bitbucket.org/neilb/bind-article**– use the "get source" menu if you don't have Mercurial installed.

Some Very Simple Stuff

We'll start by writing a very simple program that implements a function which adds two integers passed as parameters, and returns the result of the addition:

```
#include <iostream>

using namespace std;



int add( int a, int b ) {

    return a + b;

}



int main() {

    cout << add( 1, 2 ) << "\n";

}
```

If we compile this with GCC (note the use of the -std option to say we want the compiler to treat the code as C++11, or "c++0x" as the version of GCC I use insists on calling it):

```
g++ -std=c++0x prog1.cpp
```

and run the resulting executable, we get the exciting output:

```
3
```

Now, suppose (for some mad reason) we want to write a function that ALWAYS adds 1 and 2 together, and always does so using the **add** function. We could write something like this:

```
int add12() {
```

```
    return add( 1, 2 );

}
```

and call it:

```
cout << add12() << "\n";
```

This works, but means that if a bit later we find we want to add 3 and 4, we'd have to write another function, add34(). The number of functions we'd need to write could soon get out of hand. It would be nice if we could create the functions as and when we needed them without having to write separate function definitions for each one. In effect, this is what the **std::bind** function allows us to do.

Enter std::bind

Using the **std::bind** function, which is defined in the **<functional>** Standard Library header file, we can create a new function called **add12** without having to write the function body:

```
#include <iostream>

#include <functional>

using namespace std;


int add( int a, int b ) {

    return a + b;

}


int main() {

    auto add12 = bind( add, 1, 2 );

    cout << add12() << "\n";

}
```

This code, as in the previous examples, prints out the value "3". So how does it do it?

What the bind function does is to create a new object from its parameters. The first parameter is the name of the function we wish to use (in this case the **add** function), and the remaining parameters are the values that we wish to bind to the formal parameters of the named function. In this case we wish to bind the value **1** to the formal parameter **a**, and the value **2** to the formal parameter **b**. This means that when we call the function via the **add12** object, it is as if we had said **add(1, 2)**. Note that there is nothing magical about the name add12, we could equally well have written:

```
auto f = bind( add, 1, 2 );
```

```
cout << f() << "\n";
```

For the moment, don't worry about what kind of things **add12** and **f** in the above code actually are – just think of them as new functions that **bind** created for you.

Binding Variables And References

In the code we've written so far, we have just bound constant values to the function, but variables and references can also be bound. This code binds the two variables **x**and **y** to the function **add**, and outputs (yet again) the value "3":

```
int main() {

    int x = 1, y = 2;

    auto addxy = bind( add, x, y );

    cout << addxy() << "\n";

}
```

For variables, the value they represent is evaluated at the point binding takes place. If you change the variables after binding, the original values are still used, so this prints out "3" and "3", not "3" and "7";

```
int main() {

    int x = 1, y = 2;

    auto addxy = bind( add, x, y );

    cout << addxy() << "\n";

    x = 3;

    y = 4;

    cout << addxy() << "\n";

}
```

To get the binding to use the current values of x and y, we would need to bindREFERENCES to those variables. A special helper function is used to do this:

```
int main() {

    int x = 1, y = 2;

    auto addxy = bind( add, cref( x ), cref( y ) );

    cout << addxy() << "\n";

    x = 3;
```

```
    y = 4;

    cout << addxy() << "\n";

}
```

This now prints out "3" and "7" – the **std::cref** function performs binding by reference rather than by value, so that every time **addxy** is called, the current values of **x** and **y**are used..

Placeholders

Suppose we need a function **add1**, that works like this:

```
add1( 42 ); // returns 43

add1( x );  // returns x + 1
```

We would need to perform a binding which only binds _one_ of the add function's formal parameters, with the other one being supplied when we call the function via the object returned by **bind**. This is how to do it:

```
#include <iostream>

#include <functional>

using namespace std;

using namespace std::placeholders;


int add( int a, int b ) {

    return a + b;

}


int main() {

    auto add1 = bind( add, 1, _1 );

    cout << add1( 42 ) << "\n";

}
```

The special value **_1** is a PLACEHOLDER, defined in the **std::placeholders** namespace. Placeholders stand for the parameter values that will be used when the bound function is actually called; **_1** is the first parameter, **_2** the second and so on. The actual number of placeholders available is implementation defined.

So, in the above code the placeholder **_1** will be replaced by the value 42 (the first parameter of **add1**) when the function is actually called, and the result will be that "43" is printed.

What does std::bind really return?

Earlier on, I said not to worry about what kind of thing calling std::bind actually produces. The reason for this is fairly simple, the C++ Standard does not specify exactly what type of thing this should be! The Standard does lay down some requirements on what the type must be able to do, but does not say what it must be called, or really how it must work – these are details left to the implementation.

For example, for this code:

```
auto add12 = bind( add, 1, 2 );
```

the GCC compiler returns an object of the following type:

```
std::_Bind_helper<int (&)(int, int), int, int>::type
```

Phew! We would not want to have to enter that every time you used **std::bind** (and things get much worse when placeholders are used), which is why in all the code presented so far, we have used the new **auto** feature of C++11 which deduces the required type for us.

However, occasionally we will need to create an object of known type, and in those cases we can take refuge in the fact that whatever type **std::bind** actually returns, it must be storable in a **std::function** object. Thus we can write code like this:

```
int main() {

    std::function <int(void)> addup;

    int z;

    cin >> z;

    if ( z % 2 )

        addup = bind( add, 1, 2 );

    else

        addup = bind( add, 3, 4 );

    cout << addup() << "\n";

}
```

If you run the above code and enter an odd number, it displays "3", if you enter an even number "7". We cannot write directly equivalent the code using **auto** because of the scope issues.

But What's It All For?

You may at this point be wondering what the point of all this is. Well, in general, the purpose of **std::bind** is to adapt functions to situations where we want their functionality, but where they have the

wrong numbers, types or positions of parameters. Here's an example; suppose we have a function called apply, which works on std::strings:

```
template <typename FUNC>

string apply( const string & s, FUNC f ) {

    string result;

    for ( size_t i = 0; i < s.size(); i++ ) {

        result += f( s[i] );

    }

    return result;

}
```

This function takes a string and the name of the a function as parameters, and applies the function to each single character in the string, constructing a new string from the function's return values.

This will work well for functions that take a single char as their parameter. For example, this function returns the next character in the ASCII character set (assuming ASCII encoding) from the parameter value:

```
char nextchar( char c ) {

      return c + 1;

}
```

so if we write some code like this:

```
string x = "foobar";

cout << apply( x, nextchar ) << "\n";
```

the result is that;

```
gppcbs
```

is displayed.

However, suppose we have a function which takes more than one parameter, such as this one allows us to specify a list of characters which we wish to change, so that only characters on the list will shifted to the next character:

```
char nextif( char c, const string & chars ) {

    if ( chars.find( c ) != string::npos ) {

        return c + 1;
```

```
    }
    else {
        return c;
    }
}
```

In other words, calling:

```
nextf( 'f', "aeiou" );
```

would return the value **'f'** unchanged, as 'f' is not in the list of vowels.

How can we call this function with apply? We cannot say:

```
apply( x, nextif );
```

or:

```
apply( x, nextif, "aeiou" );
```

as both will result in type errors. This calls for **std::bind**! This does the job:

```
cout << apply( x, bind( nextif, _1, "aeiou"  ) ) << "\n";
```

Notice that we did not need to actually name the new function that **std::bind** creates, we can simply pass the new (nameless) function directly to **apply**.

Now, you may think "Well, I simply won't write functions like 'apply'!" – possibly not, but the **<algorithm>** section of the C++ Standard Library is full of things that look very like the **apply** function presented here, and if you want to use them, you will probably have to also use **std::bind**.

Summary

This has been quite a long article, and a lot has been covered (though by no means all – I haven't address binding member functions). The main thing to take away from this is that **std::bind** provides an easy means of adapting functions (for which you may well not have the source code) to situations where they would not otherwise be callable without writing (possibly many, possibly otherwise unnecessary) wrapper functions.

" https://latedev.wordpress.com/2012/08/06/using-stdbind-for-fun-and-profit/

C++11 TUTORIAL: LAMBDA EXPRESSIONS — THE NUTS AND BOLTS OF FUNCTIONAL PROGRAMMING

"ONE HIGHLIGHT OF C++11 IS LAMBDA EXPRESSIONS: FUNCTION-LIKE BLOCKS OF EXECUTABLE STATEMENTS THAT YOU CAN INSERT WHERE

NORMALLY A FUNCTION CALL WOULD APPEAR. LAMBDAS ARE MORE COMPACT, EFFICIENT, AND SECURE THAN FUNCTION OBJECTS. DANNY KALEV SHOWS YOU HOW TO READ LAMBDA EXPRESSIONS AND USE THEM IN C++11 APPLICATIONS.

Originally, functional programming in C consisted of defining a full-blown functions and calling them from other translation units. Pointers to functionsand file-scope (i.e., static) functions were additional means of diversifying function usage in C.

C++ improved matters by adding inline functions, member functions andfunction objects. However, these improvements, particularly function objects, proved to be labor intensive.

At last, the final evolutionary stage of functional programming has arrived in the form of lambda expressions.

LAMBDA BY EXAMPLE

Note: ACCORDING TO THE C++11 STANDARD, IMPLEMENTATIONS `#include <initializer_list>` IMPLICITLY WHEN NECESSARY. THEREFORE, YOU'RE NOT SUPPOSED TO `#include` THIS HEADER IN YOUR PROGRAMS. HOWEVER, CERTAIN COMPILERS (GCC 4.7 FOR EXAMPLE) AREN'T FULLY COMPLIANT WITH THE C++11 STANDARD YET. THEREFORE, IF THE NEW INITIALIZATION NOTATION CAUSES CRYPTIC COMPILATION ERRORS, ADD THE DIRECTIVE `#include <initializer_list>` TO YOUR CODE MANUALLY.

Before discussing the technicalities, let's look at a concrete example. The last line in the following code listing is a lambda expression that screens the elements of a vector according to a certain computational criterion:

```
//C++11
vector <accountant> emps {{"Josh", 2100.0}, {"Kate", 2900.0},
{"Rose",1700.0}};
const auto min_wage = 1600.0;
const auto upper_limit = 1.5*min_wage;
//report which accountant has a salary that is within a specific range
std::find_if(emps.begin(), emps.end(),
[=](const accountant& a) {return a.salary()>=min_wage && a.salary() <
upper_limit;});
```

As all lambda expressions, ours begins with the LAMBDA INTRODUCER `[]`. Even without knowing the syntactic rules, you can guess what this lambda expression does: The executable statements between the braces look like an ordinary function body. That's the essence of lambdas; they function (pun unintended) as locally-defined functions. In our example, the lambda expression reports whether the current accountant object in the vector `emps`gets a salary that is both lower than the upper limit and higher than the minimum wage.

Inline computations such as this are ideal candidates for lambda expressions because they consist of only one statement.

DISSECTING A LAMBDA EXPRESSION

Now let's dissect the syntax. A typical lambda expression looks like this:

```
[CAPTURE CLAUSE] (PARAMETERS) -> RETURN-TYPE {BODY}
```

As said earlier, all lambdas begin with a pair of balanced brackets. What's inside the brackets is the optional CAPTURE CLAUSE. I get to that shortly.

The lambda's parameters appear between the parentheses. Although you can omit the parentheses if the lambda takes no parameters, I recommend you leave them, for the sake of clarity.

Lambda expressions can have an explicit return type that's preceded by a `->` sign after the parameter list (find out more about this new functionality in Let Your Compiler Detect the Types of Your Objects Automatically). If the compiler can work out the lambda's return type (as was the case in the first example above), or if the lambda doesn't return anything, you can omit the return type.

Finally, the lambda's body appears inside a pair of braces. It contains zero or more statements, just like an ordinary function.

Back to the `find_if()` call. It includes a lambda expression that takes `const accountant&`. Where did this parameter come from? Recall that `find_if()` calls its predicate function with an argument of type `*InputIterator`. In our example, `*InputIterator` is `accountant`. Hence, the lambda's parameter is `const accountant&`. The `find_if()` algorithm invokes the lambda expression for every `accountant` in `emps`.

Since our lambda's body consists of the following Boolean expression:

```
{return a.salary()>= min_wage && a.salary() < upper_limit;}
```

The compiler figures out that the lambda's return type is `bool`. However, you may specify the return type explicitly, like this:

```
[=](const accountant& a)->bool
{return a.salary()>= min_wage && a.salary() < upper_limit;});
```

CAPTURE LISTS

Unlike an ordinary function, which can only access its parameters and local variables, a lambda expression can also access variables from the enclosing scope(s). Such a lambda is said to have EXTERNAL REFERENCES. In our example, the lambda

accesses, or CAPTURES, two variables from its enclosing scope: `min_wage` and `upper_limit`.

There are two ways to capture variables with external references:

- Capture by copy
- Capture by reference

The capture mechanism is important because it affects how the lambda expression manipulates variables with external references.

At this stage I'm compelled to divulge another behind-the-scenes secret. Conceptually, the compiler transforms every lambda expression you write into a function object, according to the following guidelines:

- The lambda's parameter list becomes the parameter list of the overloaded `operator()`.
- The lambda's body morphs into the body of the overloaded`operator()`.
- The captured variables become data members of the said function object.

The compiler-generated function object is called the CLOSURE OBJECT. The lambda's capture clause thus defines which data members the closure will have, and what their types will be. A variable captured by copy becomes a data member that is a copy of the corresponding variable from the enclosing scope. Similarly, a variable captured by reference becomes a reference variable that is bound to the corresponding variable from the enclosing scope.

A DEFAULT CAPTURE specifies the mechanism by which all of the variables from the enclosing scope are captured. A default capture by copy looks like this:

```
[=] //capture all of the variables from the enclosing scope by value
```

A default capture by reference looks like this:

```
[&]//capture all of the variables from the enclosing scope by reference
```

There is also a third capture form that I will not discuss here for the sake of brevity. It's used in lambdas defined inside a member function:

```
[this]//capture all of the data members of the enclosing class
```

You can also specify the capture mechanism for individual variables. In the following example `min_wage` is captured by copy and `upper_limit` by reference:

```
[min_wage, &upper_limit](const accountant& a)->bool
{return a.salary()>= min_wage && a.salary() < upper_limit;});
```

Finally, a lambda with an empty capture clause is one with no external references. It accesses only variables that are local to the lambda:

```
[] (int i, int j) {return i*j;}
```

Let's look at a few examples of lambdas with various capture clauses:

```
vector<int> v1={0,12,4}, v2={10,12,14,16}; //read about the new C++11
initialization notation
[&v1](int k) {vi.push_back(k); };  //capture v1 by reference
[&]  (int m) {v1.push_back(m); v2.push_back(m) };//capture vi and vi2 by ref
[v1]() //capture v1 by copy
{for_each(auto y=v1.begin(), y!=v1.end(), y++) {cout<<y<<" ";}};
```

NAME ME A CLOSURE

In most cases, lambda expressions are ad-hoc blocks of statements that execute only once. You can't call them later because they have no names. However, C++11 lets you store lambda expressions in named variables in the same manner you name ordinary variables and functions. Here's an example:

```
auto factorial = [](int i, int j) {return i * j;};
```

This `auto`-declaration defines a CLOSURE TYPE named `factorial` that you can call later instead of typing the entire lambda expression (a closure type is in fact a compiler-generated function class):

```
int arr{1,2,3,4,5,6,7,8,9,10,11,12};
long res = std::accumulate(arr, arr+12, 1, factorial);
cout<<"12!="<<res<<endl; // 479001600
```

On every iteration, the `factorial` closure multiplies the current element's value and the value that it has accumulated thus far. The result is the factorial of 12. Without a lambda, you'd need to define a separate function such like this one:

```
inline int factorial (int n, int m)
{
return n*m;
}
```

Now try to write a factorial function object and see how many more keystrokes that would require compared to the lambda expression!

> ***Tip**: When you define a named closure, the compiler generates a corresponding function class for it. Every time you call the lambda through its named variable, the compiler instantiates a closure object at the place of call. Therefore, named closures are useful for reusable functionality (factorial, absolute value, etc.), whereas unnamed lambdas are more suitable for inline ad-hoc computations.*

IN CONCLUSION

Unquestionably, the rising popularity of functional programming will make lambdas widely-used in new C++ projects. It's true that lambdas don't offer anything you haven't been able to do before with function objects. However, lambdas are more convenient than function objects because the tedium of writing boilerplate code for every function class (a constructor, data members and an overloaded `operator()` among the rest) is relegated to compiler. Additionally, lambdas tend to be more efficient because the compiler is able to optimize them more aggressively than it would a user-declared function or class. Finally, lambdas provide a higher level of security because they let you localize (or even hide) functionality from other clients and modules.

With respect to compiler support, Microsoft's Visual Studion 11 and GCC 4.5 support the most up-to-date specification of lambdas. EDG, Clang, and Intel's compilers also support slightly outdated versions of the lambda proposal.

" http://blog.smartbear.com/c-plus-plus/c11-tutorial-lambda-expressions-the-nuts-and-bolts-of-functional-programming/

LAMBDA FUNCTIONS IN C++11 - THE DEFINITIVE GUIDE

“One of the most exciting features of C++11 is ability to create lambda functions (sometimes referred to as closures). What does this mean? A lambda function is a function that you can write inline in your source code (usually to pass in to another function, similar to the idea of a functor or function pointer). With lambda, creating quick functions has become much easier, and this means that not only can you start using lambda when you'd previously have needed to write a separate named function, but you can start writing more code that relies on the ability to create quick-and-easy functions. In this article, I'll first explain why lambda is great--with some examples--and then I'll walk through all of the details of what you can do with lambda.

WHY LAMBDAS ROCK

Imagine that you had an address book class, and you want to be able to provide a search function. You might provide a simple search function, taking a string and returning all addresses that match the string. Sometimes that's what users of the class will want. But what if they want to search only in the domain name or, more likely, only in the username and ignore results in the domain name? Or maybe they want to search for all email addresses that also show up in another list. There are a lot of potentially interesting things to search for. Instead of building all of these options into the class, wouldn't it be nice to provide a generic "find" method that takes a procedure for deciding if an email address is interesting? Let's call the method findMatchingAddresses, and have it take a "function" or "function-like" object.

```
#include <string>
#include <vector>

class AddressBook
{
    public:
    // using a template allows us to ignore the differences between functors, function
pointers
    // and lambda
    template<typename Func>
    std::vector<std::string> findMatchingAddresses (Func func)
    {
        std::vector<std::string> results;
        for ( auto itr = _addresses.begin(), end = _addresses.end(); itr != end; ++itr
)
        {
            // call the function passed into findMatchingAddresses and see if it
matches
            if ( func( *itr ) )
            {
                results.push_back( *itr );
            }
        }
        return results;
    }

    private:
    std::vector<std::string> _addresses;
};
```

Anyone can pass a function into the findMatchingAddresses that contains logic for finding a particular function. If the function returns true, when given a particular address, the address will be returned. This kind of approach was OK in earlier version of C++, but it suffered from one fatal flaw: it wasn't

quite convenient enough to create functions. You had to go define it somewhere else, just to be able to pass it in for one simple use. That's where lambdas come in.

BASIC LAMBDA SYNTAX

Before we write some code to solve this problem, let's see the really basic syntax for lambda.

```
#include <iostream>

using namespace std;

int main()
{
    auto func = [] () { cout << "Hello world"; };
    func(); // now call the function
}
```

Okay, did you spot the lambda, starting with []? That identifier, called the capture specification, tells the compiler we're creating a lambda function. You'll see this (or a variant) at the start of every lambda function.

Next up, like any other function, we need an argument list: (). Where is the return value? Turns out that we don't need to give one. In C++11, if the compiler can deduce the return value of the lambda function, it will do it rather than force you to include it. In this case, the compiler knows the function returns nothing. Next we have the body, printing out "Hello World". This line of code doesn't actually cause anything to print out though--we're just creating the function here. It's almost like defining a normal function--it just happens to be inline with the rest of the code.

It's only on the next line that we call the lambda function: func() -- it looks just like calling any other function. By the way, notice how easy this is to do with auto! You don't need to sweat the ugly syntax of a function pointer.

APPLYING LAMBDA IN OUR EXAMPLE

Let's look at how we can apply this to our address book example, first creating a simple function that finds email addresses that contain ".org".

```
AddressBook global_address_book;

vector<string> findAddressesFromOrgs ()
{
    return global_address_book.findMatchingAddresses(
        // we're declaring a lambda here; the [] signals the start
        [] (const string& addr) { return addr.find( ".org" ) != string::npos; }
    );
}
```

Once again we start off with the capture specifier, [], but this time we have an argument--the address, and we check if it contains ".org". Once again, nothing inside the body of this lambda function is executed here yet; it's only inside findMatchingAddresses, when the variable func is used, that the code inside the lambda function executes.

In other words, each loop through findMatchingAddresses, it calls the lambda function and gives it the address as an argument, and the function checks if it contains ".org".

VARIABLE CAPTURE WITH LAMBDAS

Although these kinds of simple uses of lambd are nice, variable capture is the real secret sauce that makes a lambda function great. Let's imagine that you want to create a small function that finds addresses that contain a specific name. Wouldn't it be nice if you could write code something like this?

```
// read in the name from a user, which we want to search
string name;
cin>> name;
return global_address_book.findMatchingAddresses(
    // notice that the lambda function uses the the variable 'name'
    [&] (const string& addr) { return addr.find( name ) != string::npos; }
);
```

It turns out that this example is completely legal--and it shows the real value of lambda. We're able to take a variable declared outside of the lambda (name), and use it inside of the lambda. When findMatchingAddresses calls our lambda function, all the code inside of it executes--and when addr.find is called, it has access to the name that the user passed in. The only thing we needed to do to make it work is tell the compiler we wanted to have variable capture. I did that by putting [&] for the capture specification, rather than []. The empty [] tells the compiler not to capture any variables, whereas the [&] specification tells the compiler to perform variable capture.

Isn't that marvelous? We can create a simple function to pass into the find method, capturing the variable name, and write it all in only a few lines of code. To get a similar behavior without C++11, we'd either need to create an entire functor class or we'd need a specialized method on AddressBook. In C++11, we can have a single simple interface to AddressBook that can support any kind of filtering really easily.

Just for fun, let's say that we want to find only email addresses that are longer than a certain number of characters. Again, we can do this easily:

```
int min_len = 0;
cin >> min_len;
return global_address_book.find( [&] (const string& addr) { return addr.length() >=
min_len; } );
```

By the way, to steal a line from Herb Sutter, you should get used to seeing "});" This is the standard end-of-function-taking-lambda syntax, and the more you start seeing and using lambda in your own code, the more you'll see little piece of syntax.

LAMBDA AND THE STL

One of the biggest beneficiaries of lambda functions are, no doubt, power users of the standard template library algorithms package. Previously, using algorithms like for_each was an exercise in contortions. Now, though, you can use for_each and other STL algorithms almost as if you were writing a normal loop. Compare:

```
vector<int> v;
```

```
v.push_back( 1 );
v.push_back( 2 );
//...
for ( auto itr = v.begin(), end = v.end(); itr != end; itr++ )
{
    cout << *itr;
}
```

with

```
vector<int> v;
v.push_back( 1 );
v.push_back( 2 );
//...
for_each( v.begin(), v.end(), [] (int val)
{
    cout << val;
} );
```

That's pretty good looking code if you ask me--it reads, and is structured, like a normal loop, but you're suddenly able to take advantage of the goodness that for_each provides over a regular for loop--for example, guarantees that you have the right end condition. Now, you might wonder, won't this kill performance? Well, here's the kicker: it turns out that for_each has about the same performance, and is sometimes even **faster** than a regular for loop. (The reason: it can take advantage of loop unrolling.)

If you're interested in more on C++11 lambda and the benefits to the STL, you'll enjoy this video of Herb Sutter talking about C++11 lambdas.

I hope this STL example shows you that lambda functions are more than just a slightly more convenient way of creating functions--they allow you to create entirely new ways of writing programs, where you have code that takes other functions as data and allows you to abstract away the proccessing of a particular data structure. for_each works on a list, but wouldn't it be great to have similar functions for working with trees, where all you had to do was right some code that would process each node, and not have to worry about the traversal algorithm? This kind of decomposition where one function worries about the structure of data, while delegating the data processing to another function can be quite powerful. With lambda, C++11 enables this new kind of programming. Not that you couldn't have done it before--for_each isn't new--it's just that you wouldn't have wanted to do it before.

MORE ON THE NEW LAMBDA SYNTAX

By the way, the parameter list, like the return value is also optional if you want a function that takes zero arguments. Perhaps the shortest possible lambda expression is:

```
[] {}
```

Which is a function that takes no arguments and does nothing. An only slightly more compelling example:

```
using namespace std;
#include <iostream>
```

```
int main()
{
    [] { cout << "Hello, my Greek friends"; }();
}
```

Personally, I'm not yet sold on omitting the argument list; I think the [] () structure tends to help lambda functions stand out a little more in the code, but time will tell what standards people come up with.

RETURN VALUES

By default, if your lambda does not have a return statement, it defaults to void. If you have a simple return expression, the compiler will deduce the type of the return value:

```
[] () { return 1; } // compiler knows this returns an integer
```

If you write a more complicated lambda function, with more than one return value, you should specify the return type. (Some compilers, like GCC, will let you get away without doing this, even if you have more than one return statement, but the standard doesn't guarantee it.)

Lambdas take advantage of the optional new C++11 return value syntax of putting the return value after the function. In fact, you must do this if you want to specify the return type of a lambda. Here's a more explicit version of the really simple example from above:

```
[] () -> int { return 1; } // now we're telling the compiler what we want
```

THROW SPECIFICATIONS

Although the C++ standards committee decided to deprecate throw specifications (except for a few cases I'll cover in a later article), they didn't remove them from the language, and there are tools that do static code analysis to check exception specifications, such as PC Lint. If you are using one of these tools to do compile time exception checking, you really want to be able to say which exceptions your lambda function throws. The main reason I can see for doing this is when you're passing a lambda function into another function as an argument, and that function expects the lambda function to throw only a specific set of exceptions. By providing an exception spec for your lambda function, you could allow a tool like PC Lint to check that for you. If you want to do that, it turns out you can. Here's a lambda that specifies that it takes no arguments and does not throw an exception:

```
[] () throw () { /* code that you don't expect to throw an exception*/ }
```

HOW ARE LAMBDA CLOSURES IMPLEMENTED?

So how does the magic of variable capture really work? It turns out that the way lambdas are implemented is by creating a small class; this class overloads the operator(), so that it acts just like a function. A lambda function is an instance of this class; when the class is constructed, any variables in the surrounding enviroment are passed into the constructor of the lambda function class and saved as member variables. This is, in fact, quite a bit like the idea of a functor that is already possible. The benefit of C++11 is that doing this becomes almost trivially easy--so you can use it all the time, rather than only in very rare circumstances where writing a whole new class makes sense.

C++, being very performance sensitive, actually gives you a ton of flexibility about what variables are captured, and how--all controlled via the capture specification, []. You've already seen two cases--with nothing in it, no variables are captured, and with &, variables are captured by reference. If you make a lambda with an empty capture group, [], rather than creating the class, C++ will create a regular function. Here's the full list of options:

[] Capture nothing (or, a scorched earth strategy?)

[&] Capture any referenced variable by reference

[=] Capture any referenced variable by making a copy

[=, &foo] Capture any referenced variable by making a copy, but capture variable foo by reference

[bar] Capture bar by making a copy; don't copy anything else

[this] Capture the this pointer of the enclosing class

Notice the last capture option--you don't need to include it if you're already specifying a default capture (= or &), but the fact that you can capture the this pointer of a function is super-important because it means that you don't need to make a distinction between local variables and fields of a class when writing lambda functions. You can get access to both. The cool thing is that you don't need to explicitly use the this pointer; it's really like you are writing a function inline.

```
class Foo
{
public:
    Foo () : _x( 3 ) {}
    void func ()
    {
        // a very silly, but illustrative way of printing out the value of _x
        [this] () { cout << _x; } ();
    }

private:
        int _x;
};

int main()
{
    Foo f;
    f.func();
}
```

DANGERS AND BENEFITS OF CAPTURE BY REFERENCE

When you capture by reference, the lambda function is capable of modifying the local variable outside the lambda function--it is, after all, a reference. But this also means that if you return a lamba function from a function, you shouldn't use capture-by-reference because the reference will not be valid after the function returns.

WHAT TYPE IS A LAMBDA?

The main reason that you'd want to create a lambda function is that someone has created a function that expects to receive a lambda function. We've already seen that we can use templates to take a lambda function as an argument, and auto to hold onto a lambda function as a local variable. But how do you name a specific lambda? Because each lambda function is implemented by creating a separate class, as you saw earlier, even single lambda function is really a different type--even if the two functions have the same arguments and the same return value! But C++11 does include a convenient wrapper for storing any kind of function--lambda function, functor, or function pointer: std::function.

STD::FUNCTION

The new std::function is a great way of passing around lambda functions both as parameters and as return values. It allows you to specify the exact types for the argument list and the return value in the template. Here's out AddressBook example, this time using std::function instead of templates. Notice that we do need to use the 'functional' header file.

```
#include <functional>
#include <vector>

class AddressBook
{
    public:
    std::vector<string> findMatchingAddresses (std::function<bool (const string&)>
func)
    {
        std::vector<string> results;
        for ( auto itr = _addresses.begin(), end = _addresses.end(); itr != end; ++itr
)
        {
            // call the function passed into findMatchingAddresses and see if it
matches
            if ( func( *itr ) )
            {
                results.push_back( *itr );
            }
        }
        return results;
    }

    private:
    std::vector<string> _addresses;
};
```

One big advantage of std::function over templates is that if you write a template, you need to put the whole function in the header file, whereas std::function does not. This can really help if you're working on code that will change a lot and is included by many source files.

If you want to check if a variable of type std::function is currently holding a valid function, you can always treat it like a boolean:

```
std::function<int ()> func;
// check if we have a function (we don't since we didn't provide one)
if ( func )
```

```
{
    // if we did have a function, call it
    func();
}
```

A NOTE ABOUT FUNCTION POINTERS

Under the final C++11 spec, if you have a lambda with an empty capture specification, then it can be treated like a regular function and assigned to a function pointer. Here's an example of using a function pointer with a capture-less lambda:

```
typedef int (*func)();
func f = [] () -> int { return 2; };
f();
```

This works because a lambda that doesn't have a capture group doesn't need its own class--it can be compiled to a regular old function, allowing it to be passed around just like a normal function. Unfortunately, support for this feature is not included in MSVC 10, as it was added to the standard too late.

MAKING DELEGATES WITH LAMBDAS

Let's look at one more example of a lambda function--this time to create a delegate. What's a delgate, you ask? When you call a normal function, all you need is the function itself. When you call a method on an object, you need two things: the function and the object itself. It's the difference between func() and obj.method(). To call a method, you need both. Just passing in the address of the method into a function isn't enough; you need to have an object to call the method on.

Let's look at an example, starting with some code that again expects a function as an argument, into which we'll pass a delegate.

```
#include <functional>
#include <string>

class EmailProcessor
{
public:
    void receiveMessage (const std::string& message)
    {
        if ( _handler_func )
        {
            _handler_func( message );
        }
        // other processing
    }
    void setHandlerFunc (std::function<void (const std::string&)> handler_func)
    {
        _handler_func = handler_func;
    }

private:
        std::function<void (const std::string&)> _handler_func;
};
```

This is a pretty standard pattern of allowing a callback function to be registered with a class when something interesting happens.

But now let's say we want another class that is responsible for keeping track of the longest message received so far (why do you want to do this? Maybe you are a bored sysadmin). Anyway, we might create a little class for this:

```
#include <string>

class MessageSizeStore
{
    public:
    MessageSizeStore () : _max_size( 0 ) {}
    void checkMessage (const std::string& message )
    {
        const int size = message.length();
        if ( size > _max_size )
        {
            _max_size = size;
        }
    }
    int getSize ()
    {
        return _max_size;
    }

private:
    int _max_size;
};
```

What if we want to have the method checkMessage called whenever a message arrives? We can't just pass in checkMessage itself--it's a method, so it needs an object.

```
EmailProcessor processor;
MessageSizeStore size_store;
processor.setHandlerFunc( checkMessage ); // this won't work
```

We need some way of binding the variable size_store into the function passed to setHandlerFunc. Hmm, sounds like a job for lambda!

```
EmailProcessor processor;
MessageSizeStore size_store;
processor.setHandlerFunc(
        [&] (const std::string& message) { size_store.checkMessage( message ); }
);
```

Isn't that cool? We are using the lambda function here as glue code, allowing us to pass a regular function into setHandlerFunc, while still making a call onto a method--creating a simple delegate in C++.

IN SUMMARY

So are lambda functions really going to start showing up all over the place in C++ code when the language has survived for decades without them? I think so--I've started using lambda functions in production code, and they are starting to show up all over the place--in some cases shortening code, in some cases improving unit tests, and in some cases replacing what could previously have only been accomplished with macros. So yeah, I think lambdas rock way more than any other Greek letter.

Lambda functions are available in GCC 4.5 and later, as well as MSVC 10 and version 11 of the Intel compiler.

" http://www.cprogramming.com/c++11/c++11-lambda-closures.html

C++11 LAMBDA TO FUNCTION POINTER OR STD::FUNCTION

"I recently needed to store callbacks in a map, so I needed a proper way to get a common castable object from a lambda, so either a function pointer or an std::function.

Now, lambdas in C++ are instances of an anonymous class specifically created for each lambda object, this means that two lambdas with the same code are actually different objects.

This also means that you can't pass lambdas around without using a template because there's no lambda type.

Non capturing lambdas (the ones with nothing inside the []) can be converted to their function pointer by casting them with the corresponding signature, while capturing ones can be wrapped in std::function.

This means you can do:

```
auto lambda = [](int a, float b) {
  std::cout << "a: " << a << std::endl;
  std::cout << "b: " << b << std::endl;
};

auto function  = static_cast<void (*)(int, float)>(lambda);
function(23, 5.4);

auto function2 = static_cast<std::function<void(int, float)>>(lambda);
function2(23, 5.4);
```

And it will work properly, function being a raw pointer to the function andfunction2 being an std::function object.

This also means that we can cast function to a more general pointer like (void (*)()) and function2 to void* and store it in a struct and it will be subsequentially castable to a working function, knowing the types to call it with.

The issue arises when you don't know the signature of the lambda, you can only cast knowing the signature, and I didn't want to have to write the parameters twice, so here comes the solution to get the proper signature out of a lambda:

```
template <typename Function>

struct function_traits

  : public function_traits<decltype(&Function::operator())>

{};


template <typename ClassType, typename ReturnType, typename... Args>

struct function_traits<ReturnType(ClassType::*)(Args...) const>

{

  typedef ReturnType (*pointer)(Args...);

  typedef std::function<ReturnType(Args...)> function;

};


template <typename Function>

typename function_traits<Function>::pointer

to_function_pointer (Function& lambda)

{

  return static_cast<typename function_traits<Function>::pointer>(lambda);

}


template <typename Function>

typename function_traits<Function>::function

to_function (Function& lambda)

{

  return static_cast<typename function_traits<Function>::function>(lambda);

}
```

With this, we can now get the function pointer out of the lambda or wrap it in an std::function, and store it.

To call that function we can then do something like this:

```
template <typename... Args>
void
call (void (*function)(), Args... args)
{
  static_cast<void (*)(Args...)>(function)(args...);
}

template <typename... Args>
void
call (void* function, Args... args)
{
  (*static_cast<std::function<void(Args...)>*>(function))(args...);
}
```

Bam, we have callbacks that are called and used normally, instead of reverting to cstdarg and va_list.

Type safe callbacks (full example, supports capturing lambdas)

So far calls are not type safe, as in they can be casted to handle whatever arguments even if the lambda didn't have those argument types or argument number, this can lead to very weird bugs and this isn't a very nice solution, this is C++ not C.

We can use typeid to ensure the arguments are of the appropriate type and number.

```
#include <iostream>
#include <functional>
#include <stdexcept>
#include <typeinfo>
#include <string>
#include <map>

template <typename Function>
struct function_traits
```

```
  : public function_traits<decltype(&Function::operator())>
{};

template <typename ClassType, typename ReturnType, typename... Args>
struct function_traits<ReturnType(ClassType::*)(Args...) const>
{
  typedef ReturnType (*pointer)(Args...);
  typedef std::function<ReturnType(Args...)> function;
};

template <typename Function>
typename function_traits<Function>::pointer
to_function_pointer (Function& lambda)
{
  return static_cast<typename function_traits<Function>::pointer>(lambda);
}

template <typename Function>
typename function_traits<Function>::function
to_function (Function& lambda)
{
  return static_cast<typename function_traits<Function>::function>(lambda);
}

class callbacks final
{
  struct callback final
  {
```

```
  void*                function;
  const std::type_info* signature;
};

public:
  callbacks (void)
  {
  }

  ~callbacks (void)
  {
    for (auto entry : _callbacks) {
      delete static_cast<std::function<void()>*>(entry.second.function);
    }
  }

  template <typename Function>
  void
  add (std::string name, Function lambda)
  {
    if (_callbacks.find(name) != _callbacks.end()) {
      throw std::invalid_argument("the callback already exists");
    }

    auto function = new decltype(to_function(lambda))(to_function(lambda));

    _callbacks[name].function  = static_cast<void*>(function);
    _callbacks[name].signature = &typeid(function);
```

```
  }

  void
  remove (std::string name)
  {
    if (_callbacks.find(name) == _callbacks.end()) {
      return;
    }

    delete static_cast<std::function<void()>*>(_callbacks[name].function);
  }

  template <typename ...Args>
  void
  call (std::string name, Args... args)
  {
    auto callback = _callbacks.at(name);
    auto function = static_cast<std::function<void(Args...)>*>(
      callback.function);

    if (typeid(function) != *(callback.signature)) {
      throw std::bad_typeid();
    }

    (*function)(args...);
  }

private:
```

```
    std::map<std::string, callback> _callbacks;
};

int
main (int argc, char* argv[])
{
  callbacks cb;

  cb.add("lol", [](int a, float b) {
    std::cout << "a: " << a << std::endl;
    std::cout << "b: " << b << std::endl;
  });

  cb.call("lol", 23, 5.4f);
  cb.call("lol", 23, 43);

  return 0;
}
```

Explanation of the magic

The first function_traits inherits from the second, it just simplifies the definition of the function type.

Every lambda objects has an operator() method which is used to call the lambda, this means that its definition contains both the return type and the parameters types.

The decltype of a method, is composed of the class, the return type and the list of arguments types.

The decltype is used to define a typedef for a function pointer with the proper return type and arguments types inferred from the decltype of the method.

to_function_pointer and to_function do nothing else but use the typedef defined in the function_traits template to then cast the lambda to its function pointer.

Bam, magic.

”http://meh.schizofreni.co/programming/magic/2013/01/23/function-pointer-from-lambda.html

CONCURRENCY API

NEW SINCE END OF 2014

SEVERAL C++ SINGLETON IMPLEMENTATIONS

“

This article offers some insight into **singleton design-pattern**.
The **singleton pattern** is a design pattern used to implement the mathematical concept of a singleton, by restricting the instantiation of a class to one object. The**GoF book** describes the singleton as: “Ensure a class only has one instance, and provide a global point of access to it.”
The Singleton design pattern is not as simple as it appears at a first look and this is proven by the abundance of Singleton discussions and implementations. That’s way I’m trying to figure a few implementations, some base on C++ 11 features (smart pointers and locking primitives as mutexs). I am starting from, maybe, the most basic singleton implementation trying to figure different weaknesses and tried to add gradually better implementations.
The basic idea of a singleton class implies using a static private instance, a private constructor and an interface method that returns the static instance.

Version 1
Maybe, the most common and simpler approach looks like this:

```
1  class simpleSingleton
2  {
3      simpleSingleton();
4
5
6      static simpleSingleton* _pInstance;
7  public:
8      ~simpleSingleton() {
9      }
10
11     static simpleSingleton* getInstance() {
12         if(!_pInstance) {
13             _pInstance = new simpleSingleton();
14         }
```

```
15
16          return _pInstance;
17      }
18
19      void demo() {    std::cout << "simple singleton # next - your code ..." << std::endl;      }
20  };
21
22  simpleSingleton* simpleSingleton::_pInstance = nullptr;
```

Unfortunately this approach has many issues. Even if the default constructor is private, because the copy constructor and the assignment operator are not defined as private the compiler generates them and the next calls are valid:

```
1  // Version 1
2  simpleSingleton * p = simpleSingleton::getInstance();        // cache instance pointer
3  p->demo();
4
5  // Version 2
6  simpleSingleton::getInstance()->demo();
7
8  simpleSingleton ob2(*p);          // copy constructor
9  ob2.demo();
10
11 simpleSingleton ob3 = ob2;        // copy constructor
12 ob2.demo();
```

So we have to define the copy constructor and the assignment operator having **private** visibility.

Version 2 – Scott Meyers version
Scott Meyers in his Effective C++ book adds a slightly improved version and in the **getInstance()** method returns a reference instead of a pointer. So the pointer final deleting problem disappears.
One advantage of this solution is that the function-static object is initialized when the control flow is first passing its definition.

```
1  class otherSingleton
2  {
3          static otherSingleton * pInstance;
4
5          otherSingleton ();
6
7          otherSingleton(const otherSingleton& rs) {
8              pInstance = rs.pInstance;
9          }
10
11         otherSingleton& operator = (const otherSingleton& rs) {
12       if (this != &rs) {
13                       pInstance = rs.pInstance;
14       }
15
16       return *this;
17         }
18
19     ~otherSingleton ();
20
21 public:
22
23         static otherSingleton& getInstance()
24         {
25                 static otherSingleton theInstance;
26                 pInstance = &theInstance;
27
28                  return *pInstance;
29
```

```
30              }
31
32              void demo() {       std::cout << "other singleton # next - your code ..." << std::endl;       }
33 };
34
```

```
    otherSingleton * otherSingleton::pInstance = nullptr;
```

The destructor is private in order to prevent clients that hold a pointer to the Singleton object from deleting it accidentally. So, this time a copy object creation is not allowed:

```
otherSingleton ob = *p;
ob.demo();
```

```
error C2248: otherSingleton::otherSingleton ' : cannot access private member
declared in class 'otherSingleton'
error C2248: 'otherSingleton::~otherSingleton' : cannot access private member
declared in class 'otherSingleton'
```

but we can still use:

```
1 // Version 1
2 otherSingleton *p = & otherSingleton::getInstance();          // cache instance pointer
3 p->demo();
4 // Version 2
5 otherSingleton::getInstance().demo();
```

This singleton implementation is not **thread-safe**.

Multi-threaded environment
Both implementations are fine in a single-threaded application but in the multi-threaded world things are not as simple as they look. Raymond Chen explains **here** why C++ statics are not thread safe by default and this behavior is required by the C++ 99 standard.
The shared global resource and normally it is open for race conditions and threading issues. So, the singleton object is not immune to this issue.
Let's imagine the next situation in a multithreaded application:

```
1 static simpleSingleton* getInstance()
2 {
3     if(!pInstance)                    // 1
```

```
4     {
5         pInstance = new simpleSingleton();      // 2
6     }
7
8     return pInstance;                           // 3
9 }
```

At the very first access a thread call **getInstance()** and **pInstance** is null. The thread reaches the second line (2) and is ready to invoke the new operator. It might just happen that the OS scheduler unwittingly interrupts the first thread at this point and passes control to the other thread.
That thread follows the same steps: calls the new operator, assigns **pInstance** in place, and gets away with it.
After that the first thread resumes, it continues the execution at line 2, so it reassigns pInstance and gets away with it, too.
So now WE HAVE TWO SINGLETON OBJECTS INSTEAD OF ONE, and one of them will leak for sure. Each thread holds a distinct instance.

An improvement to this situation might be a thread locking mechanism and we have it in the new C++ standard C++ 11. So we don't need using POSIX or OS threading stuff and now
locking **getInstance()** from Meyers's implementation looks like:

```
1 static otherSingleton& getInstance()
2 {
3     std::lock_guard<std::mutex> lock(_mutex);
4
5     static otherSingleton theInstance;
6     pInstance = &theInstance;
7
8         return *pInstance;
9 }
```

The constructor of class **std::lock_guard** (C++11) locks the mutex, and its destructor unlocks the mutex. While **_mutex** is locked, other threads that try to lock the same mutex are blocked.
But in this implementation we're paying for synchronization overhead for each **getInstance()** call and this is not what we need. Each access of the singleton requires the acquisition of a lock, but in reality we need a lock only when initializing **pInstance**. If **pInstance** is called **n** times during the course of a program run, we need the lock only for the first time.
Writing a C++ singleton 100% thread safe implementation it's not as simple as it appears as long as for many years C++ had no threading standard support. In order to implement a thread safe singleton we have to apply the **double-checked locking** (DCLP) pattern.

The pattern consists in checking before entering in the synchronized code, and then check the condition again.
So the first singleton implementation would be rewritten using a temporary object:

```
1  static simpleSingleton* getInstance()
2  {
3          if (!pInstance)
4          {
5                  std::lock_guard<std::mutex> lock(_mutex);
6
7                  if (!pInstance)
8                  {
9                     simpleSingleton * temp = new simpleSingleton;
10                    pInstance = temp;
11                 }
12         }
13
14         return pInstance;
15 }
```

This pattern involves testing **pInstance** for nullness before trying to acquire a lock and only if the test succeeds the lock is acquired and after that the test is performed again. The second test is needed for avoiding race conditions in case other thread happens to initialize **pInstance** between the time **pInstance** was tested and the time the lock was acquired.
Theoretically this pattern is correct, but in practice is not always true, especially in multiprocessor environments.
Due to this rearranging of writes, the memory as seen by one processor at a time might look as if the operations are not performed in the correct order by another processor. In our case the assignment to pInstance performed by a processor might occur before the Singleton object has been fully initialized.
After the first call of **getInstance()** the implementation with pointers (non-smart) needs pointer to that instance in order to avoid memory leaks.

Version 3 – Singleton with smart pointers
Until C++ 11, the C++ standard didn't have a threading model and developers needed to use external threading APIs (POSIX or OS dependent primitives). But finally C++ 11 standard has threading support.
Unfortunately, the first C++ new standard implementation in Visual C++ 2010 is incomplete and threading support is available only starting with beta version of VS 2011 or the VS 2012 release oreview version.

```
class smartSingleton
{
 private:
        static std::mutex                       _mutex;


        smartSingleton();
        smartSingleton(const smartSingleton& rs);
        smartSingleton& operator = (const smartSingleton& rs);



 public:
         ~smartSingleton();


        static std::shared_ptr<smartSingleton>& getInstance()
        {
                static std::shared_ptr<smartSingleton> instance = nullptr;


                if (!instance)
                {
                        std::lock_guard<std::mutex> lock(_mutex);


                        if (!instance)
            {
                                instance.reset(new smartSingleton());
                        }
                }


                return instance;
```

```
30              }
31
32              void demo() {       std::cout << "smart pointers # next - your code ..." << std::endl;       }
    };
```

As we know, in C++ by default the class members are **private**. So, our default constructor is **private** too. I added here in order to avoid misunderstanding and explicitly adding to**public** / **protected**.
Finally, feel free to use your special instance (singleton):

```
1  // Version 1
2  std::shared_ptr< smartSingleton > p = smartSingleton::getInstance(); // cache instance pointer
3  p->demo();
4
5  // Version 2
6  std::weak_ptr< smartSingleton > pw = smartSingleton::getInstance();// cache instance pointer
7  pw.lock()->demo();
8
9  // Version 3
10 smartSingleton::getInstance()->demo();
```

And no memory leaks emotion… 
Multiple threads can simultaneously read and write different **std::shared_ptr** objects, even when the objects are copies that share ownership.
But even this implementation using double checking pattern is still not thread safe.

Version 4 – Thread safe singleton C++ 11
To have a thread safe implementation we need to make sure that the class single instance is locked and created only once in a multi-threaded environment.
Fortunately, C++ 11 comes in our help with two new entities: **std::call_once** and **std::once_flag**.
Using them with a standard compiler we have the guaranty that our singleton is thread safely and no memory leak.
Invocations of **std::call_once** on the same **std::once_flag** object are serialized.
INSTANCES OF ***std::once_flag*** ARE USED WITH ***std::call_once*** TO ENSURE THAT A PARTICULAR FUNCTION IS CALLED EXACTLY ONCE, EVEN IF MULTIPLE THREADS INVOKE THE CALL CONCURRENTLY.
INSTANCES OF STD::ONCE_FLAG ARE NEITHER COPYCONSTRUCTIBLE, COPYASSIGNABLE, MOVECONSTRUCTIBLE NOR MOVEASSIGNABLE.

Here it is my proposal for a singleton thread safe implementation in C++ 11:

```
1  class safeSingleton
2  {
3          static std::shared_ptr< safeSingleton > instance_;
4          static std::once_flag                                   only_one;
5
6          safeSingleton(int id) {
7                  std::cout << "safeSingleton::Singleton()" << id << std::endl;
8          }
9
10         safeSingleton(const safeSingleton& rs) {
11                 instance_  = rs.instance_;
12         }
13
14         safeSingleton& operator = (const safeSingleton& rs)
15         {
16                 if (this != &rs) {
17                         instance_  = rs.instance_;
18                 }
19
20                 return *this;
21         }
22
23 public:
24         ~safeSingleton() {
25                 std::cout <<  "Singleton::~Singleton" << std::endl;
26         }
27
28         static safeSingleton & getInstance( int id )
29
```

```
30          {
31                  std::call_once( safeSingleton::only_one,
32                                  [] (int idx)
33                                  {
34                                    safeSingleton::instance_.reset( new safeSingleton(idx) );
35
36                                    std::cout << "safeSingleton::create_singleton_() | thread id " + idx << std::endl;
37                                  }
38                                  , id );
39
40                  return *safeSingleton::instance_;
41          }
42
43          void demo(int id) {         std::cout << "demo stuff from thread id " << id << std::endl;          }
44  };
45
46  std::once_flag                              safeSingleton::only_one;
    std::shared_ptr< safeSingleton >    safeSingleton::instance_ = nullptr;
```

The parameter to **getInstance()** was added for demo reasons only and should be passed to a new proper constructor. As you can see, I am using a **lambda** instead normal method.
This is how I tested my **safeSingleton** and **smartSingleton** classes.

```
1  std::vector< std::thread > v;
2  int num = 20;
3
4  for( int n = 0; n < num; ++n ) {
5     v.push_back( std::thread( []( int id )
6                               {
7                                       safeSingleton::getInstance( id ).demo( id );
```

```
8                                   }
9                                                       , n ) );
10  }
11
12 std::for_each( v.begin(), v.end(), std::mem_fn( &std::thread::join ) );
13
14 // Version 1
15 std::shared_ptr<smartSingleton> p = smartSingleton::getInstance(1);  // cache instance pointer
16 p->demo("demo 1");
17
18 // Version 2
19 std::weak_ptr<smartSingleton> pw = smartSingleton::getInstance(2); // cache instance pointer
20 pw.lock()->demo(2);
21
22 // Version 3
23 smartSingleton::getInstance(3)->demo(3);
```

So I create 20 threads and I launch them in parallel (**std::thread::join**) and each thread accesses **getInstance()** (with a demo **id** parameter). Only one of the threads that is trying to create the instance succeeds.

Other comments
I tested this implementation on a machine with Intel i5 processor (4 cores). If you see some concurrent issues in this implementation please fell free to share here. I am open to other good implementations, too.
An alternative to this approach is creating the singleton instance of a class in the main thread and pass it to the objects which require it. In case we have many singleton objects this approach is not so nice because the objects discrepancies can be bundled into a single 'Context' object which is then passed around where necessary.

Update: According to Boris's **observation** I removed **std::mutex** instance from **safeSingleton** class. This is not necessary anymore because **std::call_once** is enough to have thread safe behavior for this class.

Update2: According to Ervin and Remus's observation, in order to make things clear I simplified the implementation version 3 and this is not using **std::weak_ptr** anymore.

" http://silviuardelean.ro/2012/06/05/few-singleton-approaches/

A GLIMPSE INTO C++14: COMBINE FLEXIBILITY AND PERFORMANCE WITH DYNAMIC ARRAYS AND RUNTIME-SIZED ARRAYS

“C99 INTRODUCED THE NOTION OF VARIABLE LENGTH ARRAYS: STACK ALLOCATED BUILT-IN ARRAYS WHOSE SIZE IS DETERMINED AT RUNTIME. C++ LACKS A SIMILAR FEATURE, TO THE DISCONTENT OF MANY A PROGRAMMER. HOWEVER, TWO RECENT PROPOSALS FOR ADDING DYNAMIC ARRAYS AND RUNTIME-SIZED ARRAYS TO C++14 ARE CLOSING THE GAP AT LAST. LEARN HOW TO USE THESE NEW FEATURES TO IMITATE C99’S VARIABLE LENGTH ARRAYS IN C++.

Just as you’ve grown used to C++11, a new standard dubbed C+14 is seen on the horizon. Fret not, though, for the new standard doesn’t bring any revolutionary features such as rvalue references or multithreading. C++14 mainly consists of technical fixes and tweaks to the current C++11 standard.

However, C++14 also includes some new features, including a class template that simulates *dynamic arrays* and a core language feature known as *runtime-sized arrays with automatic storage duration*. Both of these provide mechanisms for creating array-like objects and arrays whose sizes are determined at runtime, very much like C99’s variable length arrays(VLAs). In this article I explain these new C++14 features and how they differ from traditional Standard Library containers and built-in static arrays, respectively.

Before looking at these exciting new features let’s see why traditional C++ containers don’t fit the bill.

IN QUEST OF DYNAMIC ARRAYS

Seemingly, you can stick to `std::vector` as an alternative to VLAs. In fact, at one time the C++ standards committee considered replacing built-in arrays with vectors all across the board!

Then again, vectors are not arrays. They allocate their objects on the free-store exclusively (whereas arrays’ storage is not restricted to a specific storage class). Second, `std::vector` defines the member function`resize()` that can change a vector’s

size. `resize()` violates a crucial arrayinvariant, namely its immutable size. For these reasons, you can't use`std::vector` as a VLA substitute.

At this stage, you might consider using `std::array`. `std::array` satisfies all of the requirements of a container. As with built-in arrays, it may store its data on the stack or on the free-store. Let's look at an example of`std::array` usage:

```
#include <array>
#include <iostream>
int main()
{
 //allocate on the stack
std::array<int, 10> vals{};
for (int count=0; count<vals.size(); ++count)
  vals[count]=count+1;
std::cout<<"the first value is: "
  <<vals.front()<<std::endl;
std::cout<<"the last value is: "<<vals[(vals.size()-1)]
  <<std::endl;
std::cout<<"the total number of elements is: " <<vals.size()<<std::endl;
int* alias=vals.data();
std::cout<<"the third element is: "<<alias[2]<<std::endl;
}
```

There is, however, one problem here. std::array supports only static initialization. You can't specify its size at runtime:

```
void func(int sz)
{
 std::array <int,sz> arr;//sz isn't a constant expression
}
```

`std::array` may be a reasonable substitute for built-in arrays whose size is known at compile-time. However, if a dynamic array is what you're after, you need to look elsewhere.

The standard `std::valarray` container won't do either. It supports dynamic initialization. However, it also defines `resize()`.

Let's face it: The C++11 Standard Library doesn't have a proper substitute for C99's VLAs. This is why committee members came up with two new proposals. The first proposal introduces a new container class called std::dynarray. The second introduces runtime-sized arrays with automatic storage duration. Both proposals were approved in April 2013 at the Bristol meeting. Let's look at `std::dynarray` first.

TO HAVE AND TO HOLD

A container-based approach offers two advantages:

- Seamless interfacing with standard iterators and algorithms
- Code safety, particularly with respect to range-checking and copying

The newly-added `std::dynarray` container provides the familiar interface of a Standard Library container, albeit with two restrictions:

- Once the container is initialized, its size cannot change throughout the container's lifetime.
- The container object shall not be restricted to the free-store. It may as well reside on the stack – at the discretion of the implementation.

The `std::dynarray` constructor takes a parameter indicating the number of elements in the container. There is no default constructor since there is no accepted default size for a `dynarray` object. (Note that you cannot instantiate an empty `dynarray` and populate it later. However, zero-sized`dynarray` objects are permitted.)

Additionally, `std::dynarray` supports copy construction. This implies that elements of a `dynarray` object must also be copy-constructible. The proposal doesn't mention move semantics, which is no surprise. Moving from a `std::dynarray` might violate its size invariant.

`std::dynarray` provides random access iterators and reverse iterators. However, it doesn't support construction from an `initializer_list` nor does it provide a constructor from `first` and `last` forward iterators. While the last two restrictions might seem limiting, the rationale behind them is as follows: `initializer_list` necessarily has a static number of elements. In such cases, a built-in array or a `std::array` are probably more appropriate design choices anyway. With respect to construction from `first` and `last`iterators, the technical consideration is that `std::distance(first, last)`is a constant time operation only for random access iterators. Users can calculate this value manually and pass the result to the `std::array`constructor.

The proposal introduces a new standard header file called `<dynarray>` that defines the class template `std::dynarray`. An instance of `dynarray<T>`stores elements of type `T` contiguously. Thus, if `d` is a `dynarray<T>` then it obeys the identity `&a[n] == &a[0] + n` for all `0 <= n < N`. Note that there is no mechanism for detecting whether a std::dynarray object is allocated on the free-store or on the stack.

PUTTING STD::DYNARRAY TO WORK

The following function creates a loop that traverses a std::dynarray object and prints its elements. Next, the function copy-constructs a newstd::dynarray, sorts it, accumulates its elements' values and finally, accesses the std::dynarray elements using data(), at() and the [] operator:

```
#include <dynarray> //C++14
#include <iostream>
#include <algorithm>
void show(const std::dynarray <int>& vals )
{
  std::dynarray<int>::const_iterator cit=vals.begin();
  for (; cit != vals.end(); cit++)
    std::cout << " " << *cit;
  std::cout << std::endl;
    //runtime initialization of a new dynarray
  std::dynarray <int> target (vals);
```

```
    //algorithms, iterators, data(), & rev iterators
  std::sort(target.begin(), target.end());
  for (int n=0, *p=target.data(); n<target.size(); n++ )
    std::cout << " " << p[n]<<std::endl;
  int res=
     std::accumulate(target.rbegin(),target.rend(),0);

  for (int j=0; j<target.size(); j++) {
    //subscript access and at()
   target.at(j)=j*2;
   std::cout<<" "<<target[j] <<std::endl;
  }
  std::cout<<"total: "<<res<<std::endl;
}
```

Notice how similar the interfaces of std::dynarray and std::vector are – so much so that you may replace every occurrence of std::dynarray in the listing above with std::vector without breaking the code or changing its semantics. The only crucial difference between std::dynarray and std::vectorin this context is that once you instantiate a std::dynarray object, you cannot change its size.

RUNTIME-SIZED ARRAYS

Runtime-sized arrays offer the same syntax and performance of C99's VLAs. The std::dynarray facility, on the other hand, offers the robustness of a first class Standard Library container. Additional differences between these two features include:

- Runtime-sized arrays do not allow zero-sized arrays; std::dynarraydoes.
- One can use initializer lists with runtime-sized arrays. These aren't currently supported with std::dynarray.
- There are subtle differences with respect to the exceptions that each feature might throw. I won't get into the details here but remember always to check the exception specification of each core feature and library class you use.

The following code uses a runtime-sized array. Any C++14 compliant compiler should accept it:

```
//C++14 only
#include <algorithm>
void f(std::size_t n)
{
  int arr[n]; //runtime-sized array
  for (std::size_t i=0; i< n; ++i)
    arr[i] = i*2;
  std::sort(arr, arr+n);
  for (std::size_t i=0; i< n; ++i)
    std::cout<<" "<<arr[i]<<std::endl;
}
```

Bear in mind that runtime-sized arrays aren't precisely the same as C99's VLAs. The C++14 feature is more restrained, which is just as well. Specifically, the following properties are excluded:

- Runtime-sized multidimensional arrays
- Modifications to the function declarator syntax
- sizeof(a) being a runtime-evaluated expression returning the size of a
- typedef int a[n]; evaluating n and passing it through the typedef

Examples:

```
void f(std::size_t n)
{
  int a[n]; //C++14
  unsigned int x=sizeof(a);              //ill-formed
  const std::type_info& ti = typeid(a); //ill-formed
  typedef int t[n];                      //ill-formed
}
```

IN CONCLUSION

Runtime-sized arrays with automatic storage are to std::dynarray as ordinary, statically-sized arrays are to std::array. If you want a low-level, efficient core language arrays whose size is determined at runtime, use them. If however you need a full-blown container that simulates runtime-sized arrays, use std::dynarray instead.

With respect to compiler support of runtime-sized arrays, gcc, Intel C++, and Clang already implement this feature. At present I am not aware of any implementation that supports std::dynarray. However, after its recent approval, vendors should support it soon.

”http://blog.smartbear.com/development/a-glimpse-into-c14/

C++ MEMORY MANAGEMENT AND VECTORS

“

1. When your application crashes or gets aborted for whatever reason, the OS reclaims the memory. If not you are using a truly rare OS and have discovered a bug.
2. You need to distinguish between the memory used by the vector itself and the memory of its contained objects. The vector can be created on the heap or on the stack as you noted, the memory it allocates for its contained elements is always on the heap (unless you provide your own allocator which does something else). The memory allocated by the vector is managed by the implementation of vector, and if the vector is destructed (either because it goes out of scope for a vector on the stack or because you delete a vector on the heap) its destructor makes sure that all memory is freed.

” http://stackoverflow.com/questions/965401/c-memory-management-and-vectors

REFERENCES

Bendersky, E. (2014). *Understanding lvalues and rvalues in C and C++*. Retrieved from Eli Bendersky's website: http://eli.thegreenplace.net/2011/12/15/understanding-lvalues-and-rvalues-in-c-and-c

LearnCPP. (2014). *Learn CPP*. Retrieved from Learn CPP: http://www.learncpp.com/

Meyers, S. (2005). *Effective C++ Third Edition 55 Specific Ways to Improve Your Programs and Designs.* Addison Wesley Professional.

Meyers, S. (2014). *Effective Modern C++*. o'Reilly.

Microsoft. (2014). *Lvalues and Rvalues*. Retrieved from Lvalues and Rvalues: http://msdn.microsoft.com/en-us/library/f90831hc.aspx