

C# to C++ - A Somewhat Short Guide

Last updated: 2012-05-11

Table of Contents

Introduction	1
Simple test programs	1
Namespaces	2
Defining namespaces.....	2
Scope resolution operator ::	3
Global namespace access.....	4
Fundamental types.....	5
enums.....	5
Objects – class vs. struct vs. union.....	6
Multiple inheritance	7
Union	9
Functions.....	9
Member functions	9
Standalone functions	10
Declaration vs. definition	10
Inline functions.....	11
A brief word on the volatile keyword	11
C++ constructors.....	12
Default constructor.....	12
Parameterized constructor	12
Copy constructor.....	12
Move constructor.....	12
Code example.....	13
Storage duration.....	17
Automatic duration	17

Dynamic duration.....	18
Thread duration.....	18
Static duration	19
Initialization of thread and static duration objects	19
Unwrapped 'new' keywords are dangerous; shared_ptr, unique_ptr, and weak_ptr	19
RAII - Resource Acquisition Is Initialization	20
Const-correctness.....	21
Const pointer	21
Pointer to const.....	22
Const pointer to const	22
Constant values.....	22
Const member functions	22
Mutable data members.....	23
Summary and const_cast	23
Casting values.....	24
static_cast	24
dynamic_cast.....	25
reinterpret_cast.....	26
Strings.....	26
Prefix increment vs. postfix increment	27
Collection types	28
The List<T> equivalent is std::vector	28
The Dictionary<TKey,TValue> equivalent is std::unordered_map.....	29
The SortedDictionary<TKey,TValue> equivalent is std::map.....	31
Others	31
On lvalues and rvalues (and xvalues and prvalues).....	31
Pointers.....	32
Using pointers.....	33
nullptr	35
Pointers to class member functions and the 'this' pointer; WinRT event handlers.....	35
References	37

Lvalue references	37
Rvalue references.....	38
Templates	38
Range-based for loops.....	40
Lambda expressions.....	41
Setup code.....	41
No frills, no capture lambda.....	42
Parameter specification.....	42
Specifying the return type	42
Capturing outside variables.....	43
Overriding the default capture style.....	44
Sample function object	44
Nested Lambdas	45
Using lambdas in class member functions.....	45
MACROS.....	46
Other preprocessor features	46
C++/CX (aka C++ Component Extensions).....	47
Visual Studio and C++	49
Initial configuration.....	49
IntelliSense	50
Code snippets.....	50
Including libraries	50
Precompiled headers	51
Generating assembly code files	51
Terrifying build errors	52

Introduction

This is a somewhat short guide to the important things to know if you are a C# programmer and find yourself needing or wanting to work in C++, for example to create Metro style games for Windows 8 using C++ and DirectX. In fact, this guide is written with that goal in mind so it's not necessarily a universal guide for all platforms and purposes.

This guide is also going to be fairly utilitarian and pithy, with code standing in place of elaborate commentary. I'm expecting that you know how to program already and have a good understanding of C# (or of some sort of imperative, object-oriented language at any rate).

I'm also assuming you are fairly handy with navigating the MSDN library. Its Bing search box is really awesome; if you haven't used it before, do give it a try. I like how the search is tailored to not just MSDN but also other great programmer sites like the Stack Exchange sites, CodeProject, CodePlex, etc.

I'm sprinkling a fair bit of code throughout as I said above. This is both to show you a (pseudo) real example of something and also to help illustrate words with code so that each will hopefully help to clarify the other.

Simple test programs

I highly encourage you to create a simple scratch program that you can mess around with. I do this all the time with various languages and frameworks in Visual Studio. I normally append "Sandbox" to the name so that I know it's something I am just playing around in. I tend to comment code out when done with it rather than delete it since I may want to look at it again in the future (perhaps to see a specific syntax that I puzzled out but haven't used in a while or maybe for some technique I was trying that I now want to use in a real project). If the code in question might be a bit confusing later on, I try to add some comments that will help me understand it. It's helpful to use a descriptive naming scheme for variables, classes, and functions (though I admit that I'm not too good about this in my sandbox apps). If a project gets too full or busy then I might use regions to hide a section of code or I might create another project or even another solution that will serve as a clean slate.

While developing this guide I've mostly been working in a project I called CppSandbox (developed initially in Visual Studio 2010 Ultimate and later in Visual C++ 2010 Express). It's just a C++ Win32 Console Application (without ATL or MFC but with a precompiled header). This has let me test things like thread local storage duration to confirm my understanding of certain behaviors. C++/CX code can only be tested on

using the Visual Studio 11 preview on Windows Developer Preview (and, presumably, the next preview release of Windows 8, which is due towards the end of this month (Feb 2012)). So for that section, that is what I used. For everything else you can use either Visual Studio 2010 Professional, Premium, or Ultimate, or Visual C++ 2010 Express (available free here: <http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express>).

The major feature not present in VC++ 2010 Express is the ability to compile 64-bit applications. I'm not going to be touching on that here and I don't know whether that will be possible with the free versions of VS11 when they are released. I did do some tests with 64-bit compilation in VS 2010 Ultimate just so I could examine the assembly code it generates (out of curiosity).

Let's get down to business!

Namespaces

We'll discuss namespaces first since we'll be using them quite a bit throughout the document. I assume you are familiar with namespaces and why they are a good thing to use when programming. C++ allows you to use namespaces. Indeed, everything in the C++ Standard Library is defined inside of the `std` namespace. This avoids polluting the global namespace and puts everything in one convenient namespace that's easy to remember and short to type.

Defining namespaces

To place types within a namespace, you must declare them within the namespace. You can nest namespaces if you like. For example:

```
namespace SomeNamespace
{
    namespace Nested
    {
        class SomeClass
        {
        public:
            SomeClass(void);
            ~SomeClass(void);
        };
    }
}
```

To define a member function of a class that you declared within a namespace you can do one of two things. You can use the fully-qualified type declaration when defining the member function. For example:

```
SomeNamespace::Nested::SomeClass::SomeClass (void)
{
    // Constructor
}
```

Alternatively, you can include the appropriate using directive before defining the function, e.g.:

```
using namespace SomeNamespace::Nested;
```

Note that you must say "using namespace ..." (i.e. the word namespace must be there).

Using directives can be useful but they also bring the potential of conflicting types. The same issue appears in C#. For example there might be two types named, e.g., Point within different namespaces that you have using directives for. This results (in both C++ and C#) in a compiler error because of ambiguous types. As such you would need to refer to which type you wanted explicitly by its namespace in order to resolve the ambiguity.

It's very bad style to include a using directive in a C++ header file. The reason for this is that C++ will drag along that using directive into any file that #includes that header file. This can quickly create a nightmare of conflicting types. As such, for header files you should always specify types using their fully qualified name. It's fine to include using directives within CPP files since those are not the proper target of a #include preprocessor directive and thus do not create the same potential headaches.

Scope resolution operator ::

In C++, '::' is the scope resolution operator. It is used for separating namespaces from their nested namespaces, for separating types from their namespace, and for separating member functions from their type.

Note that it is only used in the last situation when defining a member function, when accessing a member of a base class within a member function definition, or when accessing a static member function. You don't use it to access instance member functions; for those you use either '.' or '->' for depending on whether you are working through a pointer ('->') or not ('.').

This can seem complicated since C# defines just the '.' operator which is used for all of the purposes that '::' is in C++ along with accessing instance member functions. (C# also has '->' which serves the same purpose as in C++, but you may not know this since pointer use in C# is so rare. We'll discuss pointers and the '.' and '->' operators later on.) For the most part you'll be fine though. The only place it really is likely to trip you up is if you try to access a base class member by using the '.' operator rather than the '::'

operator. If you ever compile and get a syntax error complaining about "missing ';' before '.'", it's a good bet that you used a '.' where you should've used a '::' instead.

Global namespace access

If you need to explicitly reference something within the global namespace, simply begin with the '::' operator. For example (assume this code is not within a namespace declaration):

```
int GiveMeTen(void)
{
    return 10;
}

namespace Something
{
    float GiveMeTen(float ignored)
    {
        return 10.0f;
    }

    float GiveMeTwenty(void)
    {
        float a = Something::GiveMeTen(1.0f);
        int b = ::GiveMeTen(); // If you left off the :: it
                               // would think it was the float
                               // version, not the int version.
                               // You'd then get a syntax error
                               // since you aren't passing
                               // a float, which is better than
                               // silently calling the wrong
                               // function, at least (which is
                               // what would happen if the input
                               // parameters to the two different
                               // functions matched).

        return a + b;
    }
}

float GiveMeTwenty(void)
{
    // You can start with :: followed by a namespace to refer to a
    // type using its fully qualified name. If you are writing code
    // within a namespace and you have using directives that each have
    // a type with the same name, this syntax is how you would specify
    // which one you wanted.
    return ::Something::GiveMeTwenty();
}
```

By having that '::' at the beginning we are simply saying "hey compiler, start at the global namespace and resolve what follows from there".

Fundamental types

The C++ standard only requires that the integral types and floating point types each be at least as large as their next smallest counterpart. The exact sizes are left up to individual implementations. (The integer type min and max values are specified in the C++ Standard Library header file `climits`).

So when is a long not a long? When one long is a C# long (8 bytes) and the other is a Visual C++ long (4 bytes – same as an int). A Visual C++ 'long long' is 8 bytes. Microsoft has a table of fundamental sizes here:

<http://msdn.microsoft.com/en-us/library/cc953fe1.aspx>

There are two possible workarounds for this size problem.

There are Microsoft-specific integral types that specify sizes exactly, such as `__int32` (a 32-bit integer, same as a C# int), `__int64` (a 64-bit integer, same as a C# long), etc.

Also, starting in VC++ 2010, you can include the `cstdint` header, which defines types in the `std` namespace in the form of `intN_t` and `uintN_t`, where *N* is a number. For example, `std::int32_t` would be a 32-bit integer. Implementations that provide integer types of 8, 16, 32, or 64 bits in size are required to define those types (n.b. this comes from the C standard library as defined in the C99 standard which is (mostly) incorporated into C++11). These are simply typedef types that correlate with the appropriate type on the system, not separate types themselves.

In Windows, the float and double types are the same sizes as in C# (32-bit and 64-bit, respectively). Like with int and long, there's no guarantee of their size on any particular platform other than the previously mentioned next smallest counterpart rule. There's also a 'long double' type, but in Visual C++ it is the same as a double so I don't recommend using it to avoid confusion.

enums

Here's the basic way to define an enum:

```
enum DaysOfTheWeek
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
```

```
    Thursday,  
    Friday,  
    Saturday  
};
```

By default an enum starts at 0 and increments by 1. So in the example above, Sunday == 0, Monday == 1, Tuesday == 2, If you want to change this, you can assign values directly, like so:

```
enum Suits  
{  
    Hearts    = 1,  
    Diamonds,    // Will be equal to 2  
    Clubs     = 200,  
    Spades    = 40 // Legal but not a good idea  
};
```

If you want to assign a specific backing type (it must be an integer type (including char and bool)), you can do so like this:

```
enum DaysOfTheWeek : char  
{  
    Sunday,  
    Monday,  
    Tuesday,  
    Wednesday,  
    Thursday,  
    Friday,  
    Saturday  
};
```

The compiler will let you implicitly cast an enum to an int, but you must explicitly cast an int to an enum. We'll explore casting later on.

Objects – class vs. struct vs. union

The difference between a class and a struct is simply that a struct's members default to public whereas a class's members default to private. That's it. They are otherwise the same.

That said, typically you will see programmers use classes for elaborate types (combinations of data and functions) and structs for simple data-only types. Normally this is a stylistic choice that represents the non-object-oriented origins of struct in C and that makes it easy to quickly differentiate between a simple data container versus a full-blown object by looking to see if it's a struct or a class. I recommend following this style.

In WinRT programming, a struct that is public can only have data members (no properties or functions) and those data members can only be made up of fundamental

data types and other public structs (which, of course, have the same data-only, fundamental & public structs only restrictions).

As a terminology note, you'll commonly see structs that only have data members referred to as plain old data ("POD") structs.

You will sometimes see the friend keyword used within a class definition. It is followed by either a class name or a function declaration. What this does is give the class/function specified access to the non-public member data and functions of the class. It's probably not a great thing to use very often (if ever), but if you decide you need it for some reason and that refactoring your code would be impractical, it's there.

Multiple inheritance

C++ classes can inherit from multiple base classes. This is called multiple inheritance. I strongly recommend that you only use multiple inheritance as a workaround for the fact that C++ has no separate "interface" type. Design classes in the same way that you would in C# (i.e. with either no (explicit) base class or else with just one base class). When you want an interface, create an abstract class and inherit from that as well. To avoid trouble, have your interface classes only define pure virtual member functions. The syntax for such is to follow its parameter list with '= 0'. For example:

```
class IToString
{
public:
    // Require inheriting classes to define a ToString member function
    virtual std::wstring ToString() = 0;
};

class SomeBaseClass
    : virtual public IToString
{
public:
    SomeBaseClass()
        : m_count()
    {
    }

    ~SomeBaseClass() { }

    std::wstring ToString() { return std::wstring(L"SomeBaseClass"); }

    void AddToCount(int val) { m_count += val; };

    int GetCount() { return m_count; }
};
```

```

protected:
    int m_count;
};

class SomeClass
    : public SomeBaseClass
    , virtual public IToString
{
public:
    SomeClass() { }

    ~SomeClass() { }

    void SubtractFromCount(int val) { m_count -= val; }

    std::wstring BaseToString()
    {
        return SomeBaseClass::ToString();
    }

    std::wstring ToString() override
    {
        return std::wstring(L"SomeClass");
    }
};

```

You'll notice that we marked the inheritance from the "interface" class with the virtual keyword when inheriting from it. This prevents a bizarre, battle of the inheritances from playing out. It'll compile without it (maybe) and if so likely even work right without it. However if the class you inherit from twice had data members, your new class would wind up with two copies of those data members (thereby making the class take up more room in memory).

Also, without marking the inheritance virtual, you must implement ToString in SomeClass rather than just inheriting it from SomeBaseClass. Otherwise you would get a compile error complaining that SomeClass is an abstract class wherever you tried to instantiate it and errors about being unable to pick between SomeBaseClass::ToString and IToString::ToString whenever you tried to call SomeClass's ToString member function. So the compiler issues warnings to you if you don't have those virtual markers because it's not sure that you really wanted two implementations of IToString.

Note that if you left off the override keyword from the definition of ToString in SomeClass, you would get a compiler warning about how SomeBaseClass already provides you with an implementation of IToString::ToString. By telling it we want to override any other definitions we make it clear that we want to override it and didn't just accidentally add it.

SomeClass's BaseToString member function shows the syntax you use when you want to call a member function of a class you've inherited from. It's the same syntax you'd use if you were calling a static member function, but the compiler knows that it's not a static and makes sure to translate things correctly so that it'll call SomeBaseClass's ToString when you use the '::' operator with a base class name inside a derived class like that.

(n.b. There is a concept in various object-oriented languages known as a mixin. A mixin is essentially an interface where the methods are implemented rather than left as pure declarations. Some people find them useful. There may be situations in which they make sense (code that will always be the same and that, for some reason, cannot be stuck in a common base class). I'm leery of the idea and would rather stick with interfaces and templates, myself.)

Union

A union is a data type that can't inherit from other classes or be a base class for other classes to inherit from. (In the last sentence, class refers to 'class', 'struct', and 'union' which are considered to be mandatory class-keys that specify the particular behavior of a class type). A union is made up of data members and (optionally) functions. Only one of its non-static data members can be active. This lets it fill several different roles at different times. It can also open the door to potential micro-optimizations.

There's a lot more that could be said about unions, but I have no intention of explaining them here. Unions are complicated to explain and I don't personally find them helpful. I see them as being part archaic and part advanced. You can get by just fine without them and if you really need to know more you can read up about them elsewhere. If there's sufficient desire, I'll reconsider my decision and perhaps either add more about them here or else write about them elsewhere or at least link to someone who has.

Functions

C++ allows you to write two types of functions: member functions and standalone functions.

Member functions

Member functions are the equivalent of C# methods. It's really just a terminology difference. They can be virtual. They can be static. They can be const (we will discuss what this means when we talk about const-correctness later on).

Standalone functions

Standalone functions are kind of like static methods in a C# class, only without the class. You declare and define them in the same way as you would declare and define a class member function. For example, a declaration would look like this:

```
int Add(int, int);
```

and a corresponding definition would look like this:

```
int Add(int a, int b)
{
    return a + b;
}
```

That's it. If you've ever found yourself creating a static class with all sorts of unrelated methods in C#, you don't need to do that in C++. You can just use functions instead. Functions can be placed within namespaces.

Declaration vs. definition

Above we showed the declaration of a function and its subsequent definition. When the C++ compiler comes across a function call, it must already know the declaration for that function. Otherwise it won't be able to tell if you are using it properly (passing the correct number and types of parameters and capturing the result, if any, into an appropriate data type) or even if you really meant what you wrote or if it was just some typo. It doesn't need to know the definition, however (except if you want it to consider inlining the function; more on that below). This is true not just for functions but for types (class, struct, union) as well.

This is what makes it possible to just include header files; as long as the compiler knows the declaration of what it is working with, it can determine its size, its member functions, its parameters, its return type, and any other information it needs in order to properly generate code that references it.

Note that you do not need to have a separate declaration. If you are defining an inline function in a header file, for example, it wouldn't normally make sense to both declare it and define it. The definition of a function will serve as its declaration just so long as the compiler reaches the definition of that function before it gets to a point where it needs to know what that particular function is.

Inline functions

You can mark member and standalone functions with the inline keyword. Inline functions should be defined directly in a header file and should be on the smaller side (only a couple of lines of code).

When you apply the inline keyword, you are telling the compiler that you think it would be a good idea for it to take this particular function and put its code directly where you are calling it rather than setup a call. The following is an example of an inline function:

```
inline int Add(int a, int b)
{
    return a + b;
}
```

Inlining functions can be a performance optimization, provided that the functions are small. Do not expect the compiler to respect your request for inlining; it is designed to do a lot of optimization and will make a decision on inlining based on what it thinks will give the best results.

A brief word on the volatile keyword

There is a keyword in C++ (and C and C#) called 'volatile'. It likely does not mean what you think it means. For example, it is not good for multi-threaded programming (see, e.g. <http://software.intel.com/en-us/blogs/2007/11/30/volatile-almost-useless-for-multi-threaded-programming/>). The actual use case for volatile is extremely narrow. Chances are, if you put the volatile qualifier on a variable, you are doing something horribly wrong.

Indeed, Eric Lippert, a member of the C# language design team, said "[v]olatile fields are a sign that you are doing something downright *crazy*: you're attempting to read and write the same value on two different threads without putting a lock in place." (See: <http://blogs.msdn.com/b/ericlippert/archive/2011/06/16/atomicity-volatility-and-immutability-are-different-part-three.aspx>). He's right and his argument carries over perfectly well into C++.

The fact is that the use of 'volatile' should be greeted with the same amount of skepticism as the use of 'goto'. There are use cases for both, but chances are yours is not one of them.

As such, in this guide I'm going to pretend that the 'volatile' keyword does not exist. This is perfectly safe, since: a) it's a language feature that doesn't come into play unless you actually use it; and b) its use can safely be avoided by virtually everyone*.

(* If you are writing device drivers or code that will wind up on some sort of ROM chip then you *may* actually need volatile. But if you are doing that, then frankly you should be thoroughly familiar with the ISO/IEC C++ Standard itself along with the hardware specs for the device you are trying to interface with. You should also be familiar with assembly language for the target hardware so that you can look at code that is generated and make sure the compiler is actually generating correct code for your use of the volatile keyword. (See: <http://www.cs.utah.edu/~regehr/papers/emsoft08-preprint.pdf>).)

C++ constructors

C++ has four types of constructors: default, parameterized, copy, and move.

Default constructor

A default constructor is a constructor that can be called without an argument. Note that this includes a constructor that has parameters, provided that all the parameters have been assigned default values. There can only be one of these.

Parameterized constructor

A parameterized constructor is a constructor which has at least one parameter without a default value. You can create as many of these as you want just like in C#.

Copy constructor

A copy constructor is a special type of constructor which creates a copy of an existing class instance. There are two* potential types of these (one const, one non-const) but normally you only write a const version. The compiler will typically implicitly create one for you if you don't (there are specific rules governing this). I always find it better to be explicit rather than rely on rules I would have to look up and read carefully. (*There are potentially volatile versions as well but we are ignoring volatile.)

Move constructor

A move constructor is a special type of constructor which creates a new class instance that moves the data contents of another class into itself (thereby taking over the data). They have various uses, but an easy to understand example is where you have a `std::vector<SomeClass>` (the C# equivalent is `List<SomeClass>`). If you do an insert operation and there is no move constructor (think a `List<T>` where T is a value type) then every object that has to be moved must be copied. That churns a lot of memory and wastes a lot of time. If there is a move constructor then everything goes much faster

since the data can just be moved without copying. There are some instances where the compiler will implicitly create a move constructor for you. You should be explicit about this since you could easily wind up accidentally triggering a circumstance that causes one to suddenly be created/not created whereas previously it was not created/created (i.e. it flips behavior). You could in theory have both a const and a non-const move constructor, but a const one makes no sense at all.

With both copy and move constructors, if you provide one you should also provide the equivalent assignment operator overload.

Code example

We'll examine a class with all types of constructors now.

The following is a class named `SomeClass`. It is split between two files: a header file named `SomeClass.h` and a code file named `SomeClass.cpp`. These are standard naming conventions.

The header file contains the class declaration along with the definition of any class member functions that are inlined (whether by being defined in the declaration or by having the 'inline' keyword applied to their definition).

The code file contains the definition of any class member functions. It can also contain other code but this would not be the typical case.

The class doesn't do anything of particular value but it does demonstrate a variety of things, such as all constructor types, a destructor, the use of checked iterators to safely copy data, and the use of some helpful C++ Standard Library functions such as `swap`, `fill_n`, and `copy`. It also demonstrates both a static member function and an instance member function (one that is marked `const` (see [Const-Correctness](#) below)).

First, the header file:

```
#pragma once

#include <string>
#include <iostream>
#include <memory>
#include <iterator>

class SomeClass
{
public:
    // Default constructor with default value parameter.
    SomeClass(const wchar_t* someStr = L"Hello");
```

```

// Copy constructor.
SomeClass(const SomeClass&);

// Copy assignment operator.
SomeClass& operator=(const SomeClass&);

// Move constructor.
SomeClass(SomeClass&&);

// Move assignment operator.
SomeClass& operator=(SomeClass&&);

// Constructor with parameter.
SomeClass(int, const wchar_t* someStr = L"Hello");

// Destructor.
~SomeClass(void);

// Declaration of a static member function.
static void PrintInt(int);

// Declaration of an instance member function with const.
void PrintSomeStr(void) const;

// Declaration of a public member variable. Not per se a
// good idea.
std::wstring                                m_someStr;

private:
    int                                        m_count;
    int                                        m_place;

// This is going to be a dynamically allocated array
// so we stick it inside a unique_ptr to ensure that
// the memory allocated for it is freed.
std::unique_ptr<long long[]>                m_data;

}; // Don't forget the ; at the end of a class declaration!

// Default constructor definition. Note that we do not
// restate the assignment of a default value to someStr here in the
// definition. It knows about it from the declaration above and will
// use it if no value is provided for someStr when calling this
// constructor.
inline SomeClass::SomeClass(const wchar_t* someStr)
    : m_someStr(someStr)
    , m_count(10)
    , m_place(0)
    , m_data(new long long[m_count])
{

```

```

std::wcout << L"Constructing..." << std::endl;
// std::fill_n takes an iterator, a number of items, and a value
// and assigns the value to the items
std::fill_n(
    stdext::checked_array_iterator<long long*>(this->m_data.get(),
        m_count), m_count, 0);
}

// Copy constructor definition.
inline SomeClass::SomeClass(const SomeClass& other)
    : m_someStr(other.m_someStr)
    , m_count(other.m_count)
    , m_place(other.m_place)
    , m_data(new long long[other.m_count])
{
    std::wcout << L"Copy Constructing..." << std::endl;

    std::copy(other.m_data.get(), other.m_data.get() + other.m_count,
        stdext::checked_array_iterator<long long*>(this->m_data.get(),
            this->m_count));
}

// Copy assignment operator definition.
inline SomeClass& SomeClass::operator=(const SomeClass& other)
{
    std::wcout << L"Copy assignment..." << std::endl;

    this->m_someStr = other.m_someStr;
    this->m_count = other.m_count;
    this->m_place = other.m_place;

    if (this->m_data != nullptr)
    {
        this->m_data = nullptr;
    }

    this->m_data =
        std::unique_ptr<long long[]>(new long long[other.m_count]);

    std::copy(other.m_data.get(), other.m_data.get() + other.m_count,
        stdext::checked_array_iterator<long long*>(this->m_data.get(),
            this->m_count));

    return *this;
}

// Move constructor definition.
inline SomeClass::SomeClass(SomeClass&& other)

```

```

        : m_someStr(other.m_someStr)
        , m_count(other.m_count)
        , m_place(other.m_place)
        , m_data(other.m_data.release())
    {
        std::wcout << L"Move Constructing..." << std::endl;
    }

// Move assignment operator definition.
inline SomeClass& SomeClass::operator=(SomeClass&& other)
{
    std::wcout << L"Move assignment..." << std::endl;

    std::swap(this->m_someStr, other.m_someStr);
    std::swap(this->m_count, other.m_count);
    std::swap(this->m_place, other.m_place);
    std::swap(this->m_data, other.m_data);

    return *this;
}

// Parameterized constructor definition. Note that we do not
// restate the assignment of a default value here in the definition.
// It knows about it from the declaration above and will use it if
// no value is provided for someStr when calling this constructor.
inline SomeClass::SomeClass(int count, const wchar_t* someStr)
    : m_someStr(someStr)
    , m_count(count)
    , m_place()
    , m_data(new long long[m_count])
{
    std::wcout << L"Constructing with parameter..." << std::endl;

    for (int i = 0; i < m_count; i++)
    {
        m_data[i] = (1 * i) + 5;
    }
}

inline SomeClass::~SomeClass(void)
{
    std::wcout << L"Destroying..." << std::endl;
    //// This isn't necessary since when the object is destroyed
    //// the unique_ptr will go out of scope and thus it will be
    //// destroyed too, thereby freeing any dynamic memory that was
    //// allocated to the array (if any).
    //if (this->m_data != nullptr)
    //{

```

```

        //      this->m_data = nullptr;
        //}
}

```

Next the `SomeClass.cpp` file:

```

////// If you are using a precompiled header, you'd include that first,
////// e.g.:
// #include "stdafx.h"
#include "SomeClass.h"

// Note that we don't have the static qualifier here.
void SomeClass::PrintInt(int x)
{
    std::wcout << L"Printing out the specified integer: " << x <<
        std::endl;
}

// But we do need to specify const again here.
void SomeClass::PrintSomeStr(void) const
{
    std::wcout << L"Printing out m_someStr: " << m_someStr <<
        std::endl;

    // If we tried to change any of the member data in this method
    // we would get a compile-time error (and an IntelliSense
    // warning) since this member function is marked const.
}

```

Hopefully the above was enlightening. Terms like move constructor and copy constructor are used a lot when discussing C++ so having an example to look at should prove helpful. My goal was not to produce a class that is useful, but one where you could see the declaration patterns of these constructor types and see some ways to implement them (along with the required assignment operator overloads) by using Standard Library functions.

Storage duration

There are four possible storage durations: static, thread, automatic, and dynamic.

Automatic duration

Within a block (one or more lines of code within curly braces), a variable declared

- a) either with no duration keyword or with the 'register' keyword; AND
- b) without using the 'new' operator to instantiate it

has automatic storage duration. This means that the variable is created at the point at which it is declared is destroyed when the program exits the block. Note that each time the declaration statement is executed, the variable will be initialized. In the following:

```
for (int i = 0; i < 10; ++i)
{
    SomeClass someClass;
    someClass.DoSomething(L"With Some String");
}
```

you'll run the `SomeClass` constructor and destructor ten times (in the order constructor - destructor - constructor - destructor - ... since the current `SomeClass` instance will go out of scope each time before the condition (`i < 10`) is evaluated).

(Note that the 'auto' keyword used to be a way of explicitly selecting automatic storage duration. It's been repurposed to function the same as the 'var' keyword in C# as of C++11 (this new meaning of auto is the default in VS2010 and later). If you try to compile something using the old meaning of auto you'll get a compiler error since auto as a type specifier must be the only type specifier. If you've got a lot of legacy code you can disable the new behavior (look it up on MSDN); otherwise stick with the new meaning of auto.)

Dynamic duration

Dynamic duration is what you get when you use either the `new` operator or the `new []` operator. While it is fine and even necessary to use dynamic duration objects, you should *never* allocate them outside of either a `shared_ptr` or a `unique_ptr` (depending on which suits your needs). By putting dynamic duration objects inside of one of these, you guarantee that when the `unique_ptr` or the last `shared_ptr` that contains the memory goes out of scope, the memory will be properly freed with the correct version of `delete` (`delete` or `delete []`) such that it won't leak. If you go around playing with naked dynamic duration, you're just asking for a memory leak. For more about this see the next topic.

Thread duration

It is also possible to declare certain types of variables as having thread duration. This is similar to static duration except that instead of lasting the life of the program (as we'll see shortly), these variables are local to each thread and the thread's copy exists for the duration of the thread. Note that the thread's copy is initialized when the thread is started and does not inherit its value from the thread that started it.

C++11 has added a new keyword to declare this ('thread_local') however this keyword is not yet recognized such that you need to use the Microsoft `__declspec(thread)` syntax to obtain this behavior. For more information, see: <http://msdn.microsoft.com/en-us/library/9w1sdazb.aspx> . See below for a general overview of initialization.

Since this is a bit weird, I created a small sample to make sure I knew what was going on. It's a Win32 Console App tested in VC++ 2010 Express.

Static duration

We finish up with static duration. Primarily because static duration is what you get when none of the other durations apply. You can ask for it explicitly with the static keyword.

Initialization of thread and static duration objects

The details of exactly what happens during initialization of static and thread duration objects are complicated. Everything will be initialized before you need it and will exist until the end of the program/thread. If you need to rely on something more complex than this, you're probably doing something wrong. At any rate, you'll need to sit down and read the C++ standard along with the compiler's documentation to figure out what's going on when exactly. Some of the initialization behavior is mandatory, but a lot of it is "implementation defined", meaning you need to read the compiler's documentation (i.e. the relevant MSDN pages).

Unwrapped 'new' keywords are dangerous; shared_ptr, unique_ptr, and weak_ptr

If you've worked in a .NET (or other garbage collected) language for a while, you're likely very used to using the 'new' keyword (or its equivalent in your language of choice). Well, in C++ the 'new' keyword is an easy way to create a memory leak. Thankfully, modern C++ makes it really easy to avoid this.

First, if you have a class with a default constructor, then when you declare it, it automatically constructs itself and when it goes out of scope it is automatically destroyed then and there. We discussed this earlier in automatic storage duration.

Next, the language provides two constructs that make it easy to allocate memory and ensure that it is properly freed: `shared_ptr` and `unique_ptr`. A `shared_ptr` is an

automatically reference counted container that holds a pointer type (including dynamic arrays such as "new float[50]"). One or more `shared_ptr`s can exist for the same underlying pointer, hence the name.

Another object is the `unique_ptr`. You should use this in place of raw pointers except when you need multiple pointers to the same dynamic data (in which case use `shared_ptr`). Using a `unique_ptr` ensures that the memory owned by it will be freed when the `unique_ptr` itself is destroyed (e.g. by going out of scope, via a destructor, or via stack unwinding during an exception).

The last object to consider here is `weak_ptr`. The `weak_ptr` exists solely to solve the problem of circular references. If two objects hold `shared_ptr` references to each other (or if such a thing happens in the course of, say, a doubly-linked list) then `shared_ptr`'s internal reference count can never drop to zero and so the objects will never be destroyed. For such a situation, make one of the references a `weak_ptr` instead. `Weak_ptr` is essentially a `shared_ptr` that doesn't increase the reference count. If you need to use the `weak_ptr` to access the resource, call its `lock` function to get a `shared_ptr` of the resource and then use that. If the object was destroyed before you could get it with `lock`, you will get back an empty `shared_ptr`.

The above types are all in the C++ Standard Library's memory header file, which you include as so:

```
#include <memory>
```

Notice that there is no ".h" at the end there. That's the way all of the standard library's headers are. If you're curious as to why, see:

<http://stackoverflow.com/questions/441568/when-can-you-omit-the-file-extension-in-an-include-directive/441683#441683> .

RAII - Resource Acquisition Is Initialization

RAII is a design pattern that, when done properly, enables C++ code to successfully use exceptions without resource leaks. Since C++ doesn't have a GC the way C# does, you need to be careful to ensure that allocated resources are freed. You also need to be sure that critical sections (the equivalent of a lock statement in .NET) and other multi-threaded synchronization mechanisms are properly released after being acquired.

RAII works because of this: when an exception occurs, the stack is unwound and the destructors of any fully constructed objects on the stack are run. The key part is "fully constructed"; if you get an exception in the midst of a constructor (e.g. an allocation

failure or a bad cast) then since the object isn't fully constructed, its destructor will not run. This is why you always put dynamic allocations inside of `unique_ptr` or `shared_ptr`. Those each become fully constructed objects (assuming the allocation succeeds) such that even if the constructor for the object you are creating fails further in, those resources will still be freed by the `shared_ptr/unique_ptr` destructor. Indeed that's exactly where the name comes from. Resource acquisition (e.g. a successful allocation of a new array of integers) is initialization (the allocation happens within the confines of a `shared_ptr` or `unique_ptr` constructor and is the only thing that could fail such that the object will be initialized assuming the allocation succeeds (and if it doesn't then the memory was never acquired and thus cannot be leaked)).

RAII isn't only about `shared_ptr` and `unique_ptr`, of course. It also applies to classes that represent, e.g., file I/O where the acquisition is the opening of the file and the destructor ensures that the file is properly closed. Indeed this is a particularly good example since you only need to worry about getting that code right just the once (when you write the class) rather than again and again (which is what you would need to do if you couldn't use this and instead had to write the close logic every place that you needed to do file I/O).

So remember RAII and use it whenever dealing with a resource that, when acquired, must be freed. (A critical section is another good candidate; successfully getting the enter into the critical section is the acquisition and the destructor would then make sure to issue the leave).

Const-correctness

Const-correctness refers to using the `const` keyword to decorate both parameters and functions so that the presence or absence of the `const` keyword properly conveys any potential side effects. The `const` keyword has several uses in C++. For the first three uses, imagine we have the following variable:

```
int someInt = 0;
int someOtherInt = 0;
```

Const pointer

```
int* const someConstPointer = &someInt;
//someConstPointer = &someInt; // illegal
*someConstPointer = 1; // legal
```

A const pointer is a pointer that cannot be pointed at something else. You can change the value of the data at the location the const pointer points to. So above, attempting to change the target (even to the same target) is illegal and thus won't compile but

changing the value of `someInt` by dereferencing `someConstPointer` is perfectly legal and `someInt` will now have the value 1.

Pointer to const

```
const int* somePointerToConst = &someInt;  
somePointerToConst = &someOtherInt; // legal  
//*somePointerToConst = 1; // illegal
```

A pointer to const is a pointer to a value that you cannot change via the pointer. You can make the pointer point to something else, though. So above, you can change the target of `somePointerToConst`. But you cannot change the value of whatever it is pointing to. At least, not via the pointer; you can still set `someInt` and `someOtherInt` to have other values either directly or via a pointer that is not a pointer to const. In other words, the `const` keyword only affects the pointer, not the underlying data.

Const pointer to const

```
const int* const someConstPointerToConst = &someInt;  
//someConstPointerToConst = &someInt; // illegal  
//*someConstPointerToConst = 1; // illegal
```

A const pointer to const is, as you might guess, a pointer to a value that you cannot change via the pointer and that cannot be pointed at something else. It's an amalgamation of the previous two uses of `const`.

Constant values

You can also use `const` to specify that a value in general is constant. It need not be a pointer. For instance you could have

```
const int someConstInt = 10;
```

which would create an `int` that was constant (i.e. unchangeable).

If `someInt` up above was made a `const` then the declaration of `someConstPointer` would be illegal since you would be trying to create a pointer to an `int`, not a pointer to a `const int`. You would, in effect, be trying to create a pointer that could modify the value of a `const int`, which by definition has a constant, un-modifiable value.

Const member functions

Sometimes you will see a function that is declared and defined with the `const` keyword after the parentheses in which its parameters (if any) go. For example:

```
void PrintCount(void) const
{
    wcout << L"m_count = " << m_count << endl;
}
```

What this usage of `const` means is that the function itself will not modify any non-static data members of the class and that it will not call any member function of the class unless they are also marked `const`.

Mutable data members

In certain instances you may wish to be able to change a particular data member even within constant member functions. If you mark the data member with the mutable keyword, e.g.

```
mutable int m_place;
```

then that data member can be changed even within member functions marked as `const`.

Summary and `const_cast`

When you use `const` to appropriately decorate function parameters, mark class member functions `const` where appropriate, and mark member data that needs to be changed within member functions that are otherwise marked `const`, you make it easier to understand the side-effects of your code and make it easier for the compiler to tell you when you are doing something that you told yourself you would not do.

The point of `const`-correctness is to prevent bugs and to make it easier to diagnose a bug. If you have some instance data member that is getting a completely wrong value somehow, you can instantly eliminate any functions that are marked `const` from your search since they should never be changing the instance data (unless it's marked `mutable`, in which case you know to look at those `const` functions too).

Unfortunately there's this thing called `const_cast<T>` which can ruin the party. The `const_cast` operator can, in many circumstances, eliminate the "const"-ness of something. It also eliminates any `volatile` and `__unaligned` qualifiers. You should really, really try to avoid using `const_cast` if at all possible. But `const_cast` does have some legitimate uses (otherwise why include it). If you're interfacing with old code and/or a C language library that doesn't follow `const`-correctness and you **know** that the function you are calling does not modify a variable that it takes as a non-`const` parameter, then you can mark the parameter to your function that you want to pass as `const` and then use `const_cast` to strip the `const`-ness from it so you can pass it to that function.

Casting values

While C++ supports C-style casts, they are not recommended. This is a C-style cast.

```
int x = 20;
long long y = (long long)x;
```

We have discussed `const_cast` already. The other types are: `static_cast`, `reinterpret_cast`, and `dynamic_cast`. We'll take each in turn.

Note: We'll be using the classes from the "Multiple inheritance" section earlier for code examples.

`static_cast`

The `static_cast` operator is useful for casting:

- floating point types to integer types;
- integer types to floating point types;
- enum types to integer types;
- integer types to enum types;
- derived classes to base classes;
- from a type to a reference to a compatible type; and
- from a pointer to a derived class to a pointer to one of its base classes.

Note that floating point to integer is a truncating conversion, not a rounding conversion. So "10.6" `static_cast`d to an `int` will give you 10 not 11.

Here are some examples:

```
int a = 10;
float b = static_cast<float>(a);
int c = static_cast<int>(b);
```

```
// Define an enum to work with.
enum DaysOfTheWeek
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};
```

```
DaysOfTheWeek today = Tuesday;
```

```
today = static_cast<DaysOfTheWeek>(3); // Sets it to Wednesday.
int todayAsInt = today; // 'todayAsInt' is now 3. No cast needed.
```

```
SomeClass someCls;
SomeBaseClass someBase = static_cast<SomeBaseClass>(someCls);
SomeBaseClass& rSomeBase = static_cast<SomeBaseClass&>(someCls);
SomeClass* pSomeCls = &someCls;
SomeBaseClass* pSomeBase = static_cast<SomeBaseClass*>(pSomeCls);
```

The compiler does some basic checking when you use `static_cast`, but it will not pick up on every single bad cast you could make. There is no runtime checking of `static_cast`. If you succeed in doing something that makes no sense (e.g. casting a base class instance to a derived class when it really is an instance of that base class and not of the derived class), the behavior you get is undefined (e.g. it could sort of succeed only with any derived class data members being wrong values; it could set your hard drive on fire and laugh at you while your machine slowly burns; etc.).

dynamic_cast

When converting using pointers or references, you can use `dynamic_cast` to add runtime checking in. If a pointer cast fails, the result will be `nullptr`. If a reference cast fails, the program will throw a `std::bad_cast` exception.

```
SomeBaseClass someBase;
SomeClass someClass;

SomeBaseClass* pSomeBase = &someBase;
SomeClass* pSomeClass = &someClass;

pSomeBase = dynamic_cast<SomeBaseClass*>(pSomeClass);

// The following yields nullptr since the conversion fails.
pSomeClass = dynamic_cast<SomeClass*>(pSomeBase);

SomeBaseClass& rSomeBase = dynamic_cast<SomeBaseClass&>(someClass);

try
{
    // The following throws a std::bad_cast exception.
    SomeClass& rSomeClass = dynamic_cast<SomeClass&>(someBase);
}
catch(std::bad_cast e)
{
    wcout << e.what() << endl;
}
}
```

The predictable failure behavior of `dynamic_cast` makes it a good tool to use when tracking down casting issues. It is slower than `static_cast` because of the runtime checks, but it's probably better to track down bugs first with the guaranteed fail of `dynamic_cast` and then, if you need the performance, go back and convert to `static_cast` in areas of your code that see a lot of traffic. Note that `dynamic_cast` is limited to pointers and references only so you can't use it for all of the same casts that `static_cast` will do.

`reinterpret_cast`

This is a dangerous operator. It's dangerous in that it allows you to do a lot of really stupid things. But it's occasionally necessary in that it allows you to do certain important conversions. So it's in the language. But you really should not use it. The only thing it is guaranteed to do is properly convert a type back into itself if you `reinterpret_cast` away from it and then back to it. Everything else is up to the compiler vendor.

If you want to know more about all of the casts, I recommend reading the top answer to this post: <http://stackoverflow.com/questions/332030/when-should-static-cast-dynamic-cast-and-reinterpret-cast-be-used> .

Strings

If you did any C or C++ programming at some point in the past you might remember these `char*` things that were used for strings. **DO NOT USE THEM.** ASCII strings and characters (`char*` and `char`) have no place in a modern program. The 80s are over, and it's a new, Unicode world.

(Note: The `char` type is still frequently used to hold a byte of raw data so you may see it used in that context (both individually and as an array). It's fine when it's raw data. Just don't use it for an actual text string. Better for a byte would probably be `std::int8_t` as defined in the `cstdint` header since that makes your intent more clear.)

Generally you'll work with `std::wstring` and/or `wchar_t*` strings. These are the two types that Windows uses for Unicode strings. If you're getting data from the internet, you may well be getting it as UTF-8. This is not the same as `wchar_t*` or `std::wstring`. If you need to deal with UTF-8 data for some reason, look around on the web for suggestions.

The `std::wstring` type is defined in the `strings` header file.

Sometimes in Windows you'll see the `_TCHAR` macro used. If you're ever writing code that needs to run on Win 9x, use the `_TCHAR` system. Since I'm presuming that you're learning C++ to use current (D3D11) DirectX technologies, most (if not all) of which

don't even exist pre-Vista, I prefer to work directly with `wchar_t` and `std::wstring` since that way macros don't obscure function parameters in IntelliSense.

C++ recognizes several prefixes for string literals. For wide character strings (the two types above), the prefix `L` is used. So

```
const wchar_t* g_helloString = L"Hello World!";
```

creates a new wide character string. The `wchar_t` type is used for Unicode strings (specifically UTF-16).

The standard library's `std::wstring` is a container for a `wchar_t*` string. It provides functions for mutating the string and still lets you access it as a `wchar_t*` when needed. To create one you would do something like this:

```
std::wstring someHelloStr(L"Hello World!");
```

If you are using `std::wstring` (which you should for any mutable strings you create) when you need to get the pointer from it to pass to a function that requires a `const wchar_t*` then use `std::wstring`'s `data` function. (If it needs a non-const `wchar_t*` then ask yourself whether or not you can accomplish what it is proposing with `wstring`'s functionality instead. If not then you need to create a copy of the data using something like the `wscpy_s` function (in `wchar.h`). Beware of memory leaks.)

Prefix increment vs. postfix increment

If you're coming from C#, you're likely used to seeing the postfix increment operator (`i++`) most everywhere.

In C++ you'll normally see the prefix increment operator (`++i`) everywhere.

In both languages the two operators mean the same thing. Postfix means "increment the variable and return the variable's original value (i.e. its value prior to incrementing it)". Prefix means "increment the variable and return the variable's resulting value (i.e. its value after incrementing it)".

To do a postfix increment, the compiler needs to allocate a local variable, store the original value in it, perform the increment, and then return the local variable. (Sometimes it can optimize the local variable away, but not always.)

To do a prefix increment, the compiler can simply increment the variable and return the result without creating an additional local variable.

In truth, the compiler is likely going to be able to optimize away your `i++` extra temp allocation if you choose to use the postfix increment (VS 2010 SP1, for example, produces identical assembly code for both (no temp) with a typical `for (int i = 0; i < 10; i++) { ... }` loop even in Debug configuration where it isn't going hyperactive with optimizations). Cases where it likely can't are primarily with custom types that actually implement the increment and decrement operators (see <http://msdn.microsoft.com/en-us/library/f6s9k9ta.aspx>) and in cases outside of a 'for' loop where you are actually using the value returned by the increment (or decrement) calculation (in which case it definitely can't since you clearly want either the one value or the other).

In general, using the prefix increment mostly seems to serve as notice that you have spent at least some time learning something about C++ programming.

Collection types

When coming from C# and .NET, you're undoubtedly familiar with the generic containers in C#. C++ also has these sorts of things but the names and methods may not be familiar (they also work a bit differently). Here are some of the more common mappings for collection types typically used in C# game development.

The `List<T>` equivalent is `std::vector`

The header file include is `#include <vector>`

A typical way to create one is like this:

```
std::vector<SomeType> vec;
```

To add an item to the end of the vector, use the `push_back` member function. To delete the element at the end use `pop_back`. For example:

```
vec.push_back(someItem); // someItem added to the end.  
vec.pop_back(); // someItem has now been removed from the end.
```

To insert an item somewhere other than at the back, use the `insert` function, e.g.:

```
vec.insert(begin(vec) + 1, someItem); // someItem added to index 1  
// To add to the beginning, you'd just use begin(vec) without the + 1
```

To remove an item use the `erase` function, e.g.:

```
vec.erase(begin(vec) + 2); // The item at index 2 will be removed.
```

Note that unless you are storing a `shared_ptr`, any external references to the object will become bad since the destructor will run upon erasing it.

To iterate a vector's items, use code that looks something like this:

```
for (auto item = begin(vec); item != end(vec); ++item)
{
    // Do stuff with item.
}
```

If the class you are storing in a vector supports move semantics (i.e. move constructor and move assignment operator) then vector will take advantage of that if you ever need to do an insert or an erase. This can provide a vast speed increase over the copy semantics it would otherwise need to use.

The Dictionary<TKey,TValue> equivalent is `std::unordered_map`

The header file include is `#include <unordered_map>`

A typical way to create one is like this:

```
std::unordered_map<int, std::wstring> someMap;
```

The syntax for adding an item is a little funny (which is why I used real types above rather than made up ones; the key doesn't need to be an int and the value doesn't need to be a wstring). Here's an example of adding an item:

```
someMap.insert(std::unordered_map<int, std::wstring>::value_type(1,
std::wstring(L"Hello")));
```

Yeah, you need to recapitulate `std::unordered_map<TKey,TValue>` in order to access its static member `value_type` (which is an alias for the appropriate constructor of the `std::pair` type for your map), which you use to insert the key and value. You may want to typedef it so you can shorten it, e.g.

```
typedef std::unordered_map<int, std::wstring> IntWstrMap;
...
someMap.insert(IntWstrMap::value_type(1, std::wstring(L"Hello"));
```

If you try to use insert with a key that already exists, the insert fails. The insert function returns a `std::pair` with the iterator as the first item and a `bool` indicating success/failure as the second, so you can do something like:

```
if (!someMap.insert(
    std::unordered_map<int, std::wstring>::value_type(
    1, std::wstring(L"Hey"))).second)
{
    wcout << L"Insert failed!" << endl;
}
```

You can also insert items by array notation syntax, e.g.:

```
someMap[0] = std::wstring(L" World");
```

If an item exists at that key it will be replaced (unlike with insert). If not, the key will be added and the item inserted there. Note that this is different than in .NET where you would get an exception if you tried to use a key that didn't exist.

To determine if a key exists, use something like this:

```
if (someMap.find(2) == end(someMap))
{
    wcout << L"Key 2 not found. Expected." << endl;
}
```

where '2' is the key you are looking for. You could also store the result of the find, check it against end(someMap), and if it's not that, then you know that you have the item.

Indeed, if you want to retrieve an item only if it exists, this is the correct way to do it:

```
auto itemPair = someMap.find(1);
if (itemPair != end(someMap))
{
    wcout << itemPair->second << endl;
}
```

If you tried using array notation, e.g.

```
auto item1Wstr = someMap[1];
```

you would wind up inserting an item at key '1' if no such item existed, with item1Wstr being the empty wstring that was inserted when you tried to get a key that didn't exist (but does now).

You can also use the at function to get an element at a specific key. If that key doesn't exist, it will throw an exception of type std::out_of_range. For example:

```
std::wstring result;
try
{
    result = someMap.at(1);
}
catch (std::out_of_range e)
{
    // Do some error handling
    wcout << e.what() << endl;
}
```

To remove an item, the easiest way is to just call the erase member function with the key you wish to remove.

```
someMap.erase(1);
```

You can also use iterators to remove specific items or even a range of items, but in practice these may not be worth the bother. The same caution about references going bad applies here as well as it did in vector. Calling erase will trigger the destructor of any object, whether key or value.

The SortedDictionary<TKey,TValue> equivalent is std::map

This is a binary tree rather than a hash map, (which is what unordered_map is). It's used in pretty much the exact same way as unordered_map so read above for usage info.

Others

There are too many collection types to go through them all in such detail. Here are several others you may be interested in:

- std::list – LinkedList<T>
- std::stack – Stack<T>
- std::queue – Queue<T>

On lvalues and rvalues (and xvalues and prvalues)

You may here mention of lvalues and rvalues from time to time. C++ has divided up rvalues into two subtypes: xvalues and prvalues. Which also generated something called glvalues. But we'll get confused if we go any further without clarifying this lvalue and rvalue business. The L and R stand for left and right. An lvalue was a value that could be on the left side of an assignment operator (in other words to the left of an = sign) while an rvalue was a value that could appear to the right of an assignment operator. So given

```
int x = 5 + 4;
```

the x would be an lvalue, while 5, 4, and (5 + 4) would all be rvalues.

C++11 has added the concept of rvalue references (which we will discuss shortly). This has created the concept of an expiring value (an xvalue), and in turn the concept of pure rvalues (prvalues).

Prvalues are things like literals as well as the result of a function, provided that that result is not a reference.

An xvalue is an object that's nearing the end of its life; for the most part this means the result of a function that returns an rvalue reference.

An rvalue is either an xvalue or a prvalue.

An lvalue is an object or a function. This also includes the result of a function where the result is an lvalue reference.

A glvalue (a generalized lvalue) is either an lvalue or an xvalue.

If you want to know more (or if the above doesn't make any sense), see:

<http://msdn.microsoft.com/en-us/library/f90831hc.aspx> and

[http://en.wikipedia.org/wiki/Value_\(computer_science\)](http://en.wikipedia.org/wiki/Value_(computer_science))

Pointers

A pointer stores a memory address. You can have a pointer to a function, a pointer to a class instance, a pointer to a struct, to an int, a float, a double, ... you get the idea. You can declare a pointer in either of the two following ways:

```
int* pSomePtr;  
int *pSomeOtherPtr;
```

Which you use is purely a style thing; the compiler doesn't care. I use the `int*` syntax, personally.

Note the naming convention of beginning the name of a pointer with a lowercase `p`. This helps you instantly recognize that the variable you are working with is (or should be) a pointer. Using this naming style helps prevent bugs and helps make bugs easier to spot when they do crop up.

DO NOT DO THIS:

```
int* pSomePtr, pNotAPointer;
```

The `pNotAPointer` variable is not a pointer to an integer. It is just an integer. The same as if you had said:

```
int pNotAPointer;
```

The `*` must be applied to each variable in a comma separated declaration, which is what makes that declaration evil for pointers. When you use a declaration like the above, it's

hard for you (and others who look at your code) to know if you meant for `pNotAPointer` to be a pointer or just an integer (this is one place where the `p` naming convention helps). It's also easy to repeatedly overlook the missing `*` and waste a lot of time trying to find the source of the bug.

Because bugs like that are hard to track down, you should never declare multiple pointers on the same line. Put them on separate lines like this:

```
int* pSomePtr;          // Definitely a pointer.
int* pSomeOtherPtr;    // Definitely a pointer.
int notAPointer;       // Definitely NOT a pointer.
```

The compiler doesn't care and you will avoid a bad style.

Using pointers

I think the best way to describe using pointers is with documented code. So here's some documented code. Note that some of this code is bad style. I point this out in the comments.

```
int x;                // Creates an integer named x. Its value is
                    // undefined (i.e. gibberish).

x = 0;                // x is now equal to zero rather than gibberish.

int* pX;              // Creates a pointer to an integer. Its value
                    // is undefined (i.e. gibberish). If you tried to
                    // dereference it you would hopefully crash your
                    // program.

pX = &x;              // pX now points to x. The & here means return
                    // the memory address of x. So pX now holds the
                    // memory address of x. This & is called the
                    // address-of operator. There's also an & that
                    // means an lvalue reference and one that means
                    // a bitwise AND. You'll learn which is which.

*pX = 1;              // Sets x to one. The * here "dereferences" the
                    // pointer (i.e. lets you operate on the value it
                    // points to rather than on the pointer itself).
                    // This * is called the indirection operator.

pX = 1;               // Bad - this makes the pointer point to memory
                    // address 0x01, which (if you are lucky) will
                    // cause your program to crash. If not you'll be
                    // randomly changing data the next time you use
                    // the pointer properly. Worse would be pX++
```

```

        // since that is more likely to give you a real
        // memory address. More on that below.

SomeClass sc;           // Create a SomeClass instance using
                        // its default constructor.

SomeClass* pSc;        // Create a pointer to a SomeClass
                        // instance.

pSc = &sc;             // Make pSc point to sc.

pSc->PrintSomeStr();   // This and the next statement are
                        // identical. The -> operator means
                        // dereference the pointer then get
                        // the member named PrintSomeStr. This
                        // syntax also works for data members.

(*pSc).PrintSomeStr(); // Here we are first dereferencing the
                        // pointer explicitly (*pSc) and then
                        // we're using the . operator to get the
                        // PrintSomeStr member. -> is just clearer
                        // looking.

pX = &x;              // Set pX pointing to x again after the mishap above.

(*pX)++;             // This increments the value of x by one. It is
                        // bad style because of what happens if you forget
                        // the parentheses.

*pX++;              // This DOES NOT increment the value of x by one.
                        // Instead this increments pX by one. It is the
                        // exact same result as if you had just written
                        // pX++; without having the * in front of it. In
                        // other words this changes the memory address
                        // pointed to by the pointer, not the value that
                        // is stored at the memory address. The * is ignored.

*pX = *pX + 1;      // This produces the exact same assembly code as
                        // (*pX)++; and doesn't have the same bug risk of
                        // accidentally forgetting the parentheses. So
                        // don't get cute with ++ and --. Just use this
                        // instead.

++*pX;              // This is another alternative that will produce the
                        // the same assembly code as (*pX)++ and *pX = *pX + 1
                        // This works because the prefix increment is not
                        // touching the pX directly but has the * in between
                        // such that it associates with the expression *pX.

int ax[4];          // Declares an array of four integers. The
                        // values are all undefined. But this is

```

```

        // not a dynamic array so you don't
        // need to wrap it in a unique_ptr or
        // anything.

int* pAx = &ax[0]; // Creates a pointer to the first
                  // element of ax.

*pAx = 0;         // Sets the value of the first element of
                  // ax to 0.

pAx++;           // Sets the pointer to point to the second
                  // element of ax. Arrays are linear in memory
                  // and the ++ increments the pointer by the
                  // size of one element (in this case the size
                  // of one integer).

*pAx = 20;       // Sets the value of the second element of
                  // ax to 20. It effects the same result as
                  // writing ax[1] = 20; would.

```

The above code should give you everything you need to know about how to work with pointers. Whether it's a pointer to `int` or a pointer to `SomeClass` doesn't matter. When working with an array, incrementing a pointer to the array's first element with `++` can be a lightning fast way to initialize the array. But if you mess up and run past the end of the array then you'll be corrupting memory (and hopefully will crash your program).

nullptr

When you need to specify that a value is null in C++, use the `'nullptr'` keyword. This is new in C++11 but exists in VC++ 2010.

If you look through old code, you will likely see things used like a `NULL` macro or even just the number `0`. These are holdovers from the olden days. You should never use them. Ever. There's nothing faster or better about them. The compiler won't generate better code with them. They are just old, crummy syntax from the days before there was an official keyword for the concept of nullity. Now that there is one, they are just bad relics that should be ignored.

Pointers to class member functions and the 'this' pointer; WinRT event handlers

One common pattern you'll see in WinRT for delegates and event handlers is this:

```

_someToken = SomeEvent += ref new SomeEventHandler(
    this,

```

```
        &SomeClass::MemberFunctionHandlerForSomeEvent  
    );
```

In C++, class member functions only exist once in memory no matter how many instances of the class you have. This makes sense since the function itself doesn't change for each instance; only the data that it operates on changes. So all the function needs to know is which data to operate on. As we'll see shortly, the C++ compiler takes care of that for us.

(The same is true in C# actually. Or usually true, anyway. The lone difference is that in C# if you've never run or referenced a method then the method probably won't be in memory since it likely hasn't been JITted yet. Once the program needs the method to be in memory it will be JITted and then it will exist once in memory, just like in C++.)

What the event handler constructor code is doing is constructing a delegate that says "hey you event, when you fire I want you to call me, this instance (ergo you pass the "this" pointer to specify that it should use this instance's data) with the class member function that is located at this memory address (which is what using the address-of operator with a class member function does gives you). It's the combination of instance data and the member function's address in memory that lets the event call the right member function with the right data whenever the event fires. (The bit about the token is just a peculiarity with how you unsubscribe from an event in WinRT when using C++.)

Note that we use the scope-resolution operator, "::", when taking the address of the member function. We use this (rather than . or ->) since what we are after is the address in memory of the member function itself, which as we already said is common to all instances of the class. It will make more sense if we briefly examine what happens when a C++ class member function is compiled.

Behind the scenes, when the C++ compiler compiles a member function it automatically adds a "this" pointer as a parameter to all of the instance member functions of our classes. The "this" pointer is a pointer to a particular instance of the class that the member function is a member of. When you call an instance member function the compiler takes that call and uses that instance's location in memory as the value of the "this" parameter that it added. The member function is wired up to use that "this" pointer to access and operate on the correct instance data.

Knowing all of this should help you understand what the purpose of that syntax for the event handler is. When the event is triggered, in order for the event to call the member function we specified it needs the address of the class instance to pass to the member function as the compiler-added "this" parameter. The only way it can know that is if we tell it what the address is. We do that by passing in "this" as a parameter to the delegate

constructor. If you wanted some other class instance, then instead of passing in "this" you would pass in &theOtherInstance as the first parameter (using the address-of operator to get its memory address).

References

References are an attempt to prevent you from shooting your foot off with pointers without removing the ability to pass parameters by reference. They largely succeed but have some inherent limits and are different than .NET references (which really act more like pointers in some ways).

Lvalue references

Lvalue references are a way to create a reference to a value. They are useful for passing a parameter by reference rather than copying by value. Like with pointers, some examples will help explain them.

```
int x = 0;    // Create an int named x and set it equal to zero.

int& rX = x; // Creates an integer reference named rX and sets
            // it to refer to x. From now on rX acts like x.

x++;        // x is now equal to one.
rX++;       // x is now equal to two. Note how we didn't need
            // to dereference anything. Once rX is assigned a
            // value it becomes that value for all intents and
            // purposes. You cannot make rX refer to anything
            // else.

int &rY; // This is illegal. A reference must be assigned a value
        // when it is created. The only exception is in a
        // function definition since the value is assigned when
        // the function is called or in a class data member
        // definition (though I don't see much utility in having
        // a data member that is a reference, personally).

// For the following function, by using a reference, we won't
// invoke the copy constructor when calling this function
// since rSc is a reference to an instance of SomeClass,
// not a separate instance of SomeClass. And we'll only be
// passing the size of a reference (4 bytes in a 32-bit
// program) versus the size of the object (44 bytes using the
// definition of SomeClass from the C++ constructors section).
// So references are your friend. In this case, since we won't
// be changing anything in the SomeClass instance, we mark the
// rSc parameter const.
void DoSomething(const SomeClass& rSc)
{
```

```
rSc.PrintSomeStr(); // Notice that we use the . operator and
                    // not the -> operator.
}
```

Passing parameters by reference (as in the DoSomething function above) prevents memory churning, prevents a constructor from running, and gives us the same behavior as we would expect if we passed a class as a parameter in .NET (i.e. a reference to the same object, not a separate copy of the object). (n.b. If you do not pass in an object of the type specified, you can actually get memory churn and a constructor running if the type of what you passed in is convertible to the type specified.)

If you want to pass a copy of the object to a function for some reason then you can. Just leave off the reference marker on the parameter declaration and it will make a copy of the object and pass that copy in when you call the function. If you want to pass a copy to a function that takes a parameter by reference then you need to construct a copy first and then pass the copy in as the parameter when you call the function (since the function definition will only take a reference such that it won't trigger the copy constructor on its own).

Rvalue references

We've already seen rvalue references in the C++ constructors. The move constructor and move assignment operator both dealt in them. They use a syntax that looks like this:

```
SomeClass&& rValRef;
```

They are only particularly useful for move semantics and as we've already covered that in the constructors section, there's not much more to say about them here. They are new in C++11 but VC++ 2010 supports them.

Templates

Templates are sort of like .NET generics except that they aren't. They work to accomplish the same goal (generic programming) but do so in a different way. If you are curious to learn more about how .NET generics work, I recommend reading this interview with Anders Hejlsberg: <http://www.artima.com/intv/generics.html> . I'm going to focus strictly on C++ templates.

Templates are an integral part of the C++ Standard Library. Indeed, many parts of the Standard Library derive from or are otherwise based on an earlier project called the Standard Template Library and it's common to see the Standard Library referred to as the STL. (See, e.g.: <http://stackoverflow.com/tags/stl/info> and <http://stackoverflow.com/tags/stdlib/info>).

C++ templates allow you to write classes and stand-alone functions which take in type parameters and perform operations on them. Here is an example:

```
#include <iostream>
#include <ostream>

template< class T >
void PrintTemplate(T& a)
{
    std::wcout << a << endl;
}
```

What C++ does with this is interesting. The compiler will generate a version of `PrintTemplate` for each type that you call it with and will then use that version for all invocations of `PrintTemplate` with that data type. So, for example, if you wrote:

```
int x = 40;
PrintTemplate<int>(x);
```

the compiler would create a special `int` version of `PrintTemplate`, verify that this version of `PrintTemplate` can in fact work with an `int` (in this case making sure that `int` has a `<<` operator defined for it), and if so create everything. Since everything is generated at compile-time it is very fast at execution time. A downside is that you can get some bizarre error stuff in the output window if you tried to pass in a type that doesn't have a `<<` operator defined (e.g. our `SomeClass` type). And the build will fail, of course.

Indeed, overloaded operators tend to play a big part in template programming. If you take in two types and try to add them, you need to make sure that there's a `+` operator defined that adds those two types, otherwise it'll be carnage in your output window.

As far as the syntax goes, you just prefix the function or class with `template< ... >` and you are set to go. You can pass as many or as few types as you want. The "class" keyword in there includes classes, structs, and built-in types like `int` (it likely includes unions too, though I have not tested that). The letter `T` is just a style convention, the same as in .NET; you can use anything as an identifier.

You can also use concrete types if you like, but then you need to pass a constant value in as the type parameter when invoking the template function/class.

When defining a template, separate multiple types with a comma.

It's very easy to mess up template syntax and figuring out what you did wrong is a process of looking at the error message, looking up the compiler error number on MSDN, and trying to fix it based on what the error means. If you get stuck, try reading through the MSDN reference documentation on templates:

<http://msdn.microsoft.com/en-us/library/y097fkab.aspx> .

Range-based for loops

A range-based for loop is the C++ equivalent of a C# foreach loop. You can use this syntax to loop through a `std::string`, a `std::wstring`, an array, any of the Standard Library collection types, or anything that can be used with `std::begin` and `std::end`. (Range-based for loops are supported in Visual C++ as of the VS11 Beta release, so they won't work with VC++ 2010 or the earlier VS11 Developer Preview).

For example, consider the following C# code:

```
var someList = new List<int>();

someList.Add(10);
someList.Add(20);
someList.Add(30);

foreach (var item in someList)
{
    Console.WriteLine(item);
}
```

In C++ you can write that like this:

```
std::vector<int> someVec;

someVec.push_back(10);
someVec.push_back(20);
someVec.push_back(30);

for (int item : someVec)
{
    std::wcout << item << std::endl;
}
```

C++ will even let you do something that C# won't: modify the values. For example:

```
std::vector<int> someVec;

someVec.push_back(10);
someVec.push_back(20);
someVec.push_back(30);

// Note how we make the item an lvalue reference with the &
for (int& item : someVec)
{
```

```
    item = item * 10;
}

for (int item : someVec)
{
    std::wcout << item << std::endl;
}
```

That will print out:

```
100
200
300
```

not

```
10
20
30
```

since the first range-based for loop used 'int& item' as its declaration. Note that while you can change the collection members directly using an lvalue reference declaration, just as in C#, you can't add or remove things from the collection itself in a range-based for loop since that would invalidate the iterator.

Lambda expressions

A lambda expression creates something called a function object. The function object can be invoked in the same way as an ordinary function.

There are several ways to declare lambda expressions and the easiest way to show them is probably with a code sample. Each section continues the code from the previous section and will reuse variables (and changed values) that result from previous sections.

Setup code

```
// Here we are just defining several variables we will use in the
// examples below.
int a = 10;
int b = 20;
int c = 30;
int d = 0;
```

No frills, no capture lambda

```
// This is the simplest form of lambda. The square brackets - [] -
// are the syntax that signals that this is a lambda function.
// There are various things you can put inside the []. If you leave
// it empty, as we do here, it means you are not capturing any of
// the variables outside of the lambda.
//
// The parentheses at the very end invoke the anonymous function. If
// they were not there, you'd get a function object that returns an
// int as the result instead of getting the int itself.
d = [] ()
{
    return 40; // Returns 40.
}();
```

Parameter specification

```
// If you want to specify any parameters, they go inside the ().
d = [] (int x)
{
    return x + 10;
}(20); // Returns 30 since we are passing 20 as x's value.
```

Specifying the return type

```
// If you want or need to explicitly specify the return type, you
// need to use trailing return type syntax.
d = [] () -> int // The -> after the () signals that what follows is
                // the return type. In this case, int.
{
    return 10; // Returns 10.
}();
```

```
// When your lambda has multiple return statements, you must provide
// a return type. If you do not, it will be treated as if the return
// type was void and you will get a compiler error.
d = [] (bool x)
{
    if (x)
    {
        return 20; // Error C3499: a lambda that has been specified to
                  // have a void return type cannot return a value
    }
    return 10; // Error C3499: a lambda that has been specified...
}(true);
```

```
// The above lambda is fixed by adding the trailing return type.
d = [] (bool x) -> int
{
    if (x)
    {
        return 20; // No error.
    }
    return 10; // No error.
}(true); // Returns 20 since x is true.
```

Capturing outside variables

```
// If you leave the [] empty you cannot use any of the variables
// outside of the lambda. The following will not compile.
d = [] () -> int
{
    return a + 10; // Error C3493: 'a' cannot be implicitly captured
                  // because no default capture mode has been
                  // specified.
}();
```

```
// The & inside the [] means that we want to, by default, capture
// any outside variables we use ('a' in this case) by reference.
// In essence, this just means that any changes we make to these
// variables will appear in the outside variable as well. See
// the References topic later on for more about this.
d = [&] () -> int
{
    a = 20; // Since this is by reference, the outside 'a' is also
           // now equal to 20.
    return a + 10; // Returns 30 (20 + 10).
}();
```

```
// The = inside the [] means that we are, by default, capturing any
// outside variables we use ('a') by copy.
d = [=] () -> int
{
    return a + 70; // Returns 90 (20 + 70).
}();
```

```
// If you want to be able to modify variables that are captured by
// copy, you need to use the 'mutable' keyword. The changes will not
// be reflected on the outside variable. It's just a syntax thing
// that helps prevent you from mistakenly thinking you are modifying
```

```
// an outside variable's value when you are just working on a copy.
d = [=] () mutable -> int
{
    b = 80; // Internal 'b' is now 80. External 'b' is still 20.
    return b + c; // Returns 110 (80 + 30).
}();
```

Overriding the default capture style

```
// You aren't stuck with all by reference or all by copy. You can
// specify a default and then override it for certain variables.
// Here we are specifying that by copy is the default. We then are
// saying that we want 'b' by reference.
d = [=, &b] () -> int
{
    b = 80; // The default is capture by copy but we said to
           // capture 'b' by reference so this is fine without
           // a 'mutable' statement and both inside and outside
           // 'b' are now equal to 80.

    return b + c; // Returns 110 (80 + 30). 'c' was captured by
                 // copy since that was the default capture mode.
}();
```

```
// When the default is by reference, you override it like this:
d = [&, b] () -> int // Notice that we do not prefix 'b' with an '='.
{
    // b = 60; - This would be an error since b is captured by copy.
    // If you wanted to be able to change 'b' you would need to mark
    // the lambda as 'mutable'.

    return b - c; // Returns 50. (80 - 30)
}();
```

Sample function object

```
// Here we are creating a lambda expression called f.
auto f = [] (int x, int y) -> int
{
    return x + y;
};

// And now we will invoke f.
d = f(30, 70); // 'd' is now 100.
```

Nested Lambdas

```
// You can also nest lambdas. The following is a correct example
// for VC++11. Notice how we are explicitly capturing the variables
// we will need within the nested structure.
a = 10;
b = 20;
c = 30;
d = [a,b,c]() -> int
{
    return a + [b,c]() -> int
    {
        return b + []() -> int
        {
            return c;
        }();
    }();
}(); // 'd' is now 60 (10 + (20 + (30))).

// Because of a bug in VC++ 2010, the above does not work there.
// Instead you must do something like this.
a = 10;
b = 20;
c = 30;
d = [&]() -> int
{
    int& b1 = b; // Sadly VC++ 2010 doesn't allow you to pass
                // references along inside the [] to nested lambdas.
                // If you don't reference the parameters like this,
                // you'll get an IntelliSense error but it will
                // compile. Other compilers might blowup if you did
                // that, though. This workaround of creating
                // references should work in all compilers.

    int& c1 = c;
    return a + [&]() -> int
    {
        int& c2 = c1;
        return b1 + [&]() -> int
        {
            return c2;
        }();
    }();
}(); // 'd' is now 60 (10 + (20 + (30))).
```

Using lambdas in class member functions

```
// If you create a lambda within a class member function, you can
// access the 'this' pointer for the class.
class AA
```

```

{
public:
    AA()
    : m_someInt(10)
    { }

    ~AA() { }

    int AddToSomeInt(int x)
    {
        [&] (int a)
        {
            this->m_someInt += a;
        } (x);

        return m_someInt;
    }

private:
    int m_someInt;
};

AA someAa;

d = someAa.AddToSomeInt(20); // 'd' is equal to 30.

```

MACROS

Don't create macros. You'll shoot your eye out. See, e.g.: <http://msdn.microsoft.com/en-us/library/dy3d35h8.aspx> (halfway through the Remarks section). If you're thinking about something that you think might make a good macro, use an inline function instead.

(Mind you, don't intentionally avoid macros that exist within headers that come with the Windows SDK or other SDKs and toolkits you decide to use. Just be aware of the fact that there can be unintentional side effects with macros like in the MSDN example above so avoid code with the potentiality for side effects when using a macro (. Keep it simple.)

Other preprocessor features

Do, however, use other preprocessor features. For example, you should always put

```
#pragma once
```

at the top of all header files you write to make sure they are only processed once no matter how many files `#include` them. That notation is not ISO/IEC compliant, but it is supported in most, if not all, C++ compilers, including Microsoft's.

Copyright © 2012 Michael B. McLaughlin

An ISO-compliant alternative syntax is this:

```
#ifndef __SOMECLASS_H_
#define __SOMECLASS_H_

// Your header file code here.

#endif
```

It's up to you which to use; I use the #pragma syntax since I don't need to worry about name collision that way. If you use the ISO-compliant syntax and somehow ended up with two files with the same name and forgot, you'll need to patch up the resulting error from the one file not being included due to the symbol already being defined by the previously included other file. One way to avoid that problem would be to bake in any namespaces or directory paths into the symbol names, e.g.

`__SOMENAMESPACE_SOMECLASS_H_`. You also need to make sure all the header file code is inside the region between the #define ... and the #endif if you use ISO-compliant syntax.

C++/CX (aka C++ Component Extensions)

Go watch Herb Sutter's excellent //build/ conference presentation, "Using the Windows Runtime from C++": <http://channel9.msdn.com/Events/BUILD/BUILD2011/TOOL-532T>. It's a little over an hour long and will give you good insight not just into C++/CX but also into WinRT and Metro style app development in general.

C++ Component Extensions are a set of language extensions that make it possible to interface with the Windows Runtime and to write components that can be used from languages like C#, VB, and even JavaScript.

The two most common things you will see are the hat symbol '^' which is basically a WinRT pointer and the ref keyword (used in defining a WinRT class and instantiating a new instance of them). WinRT classes are automatically reference counted so you do not need to worry about putting them inside a unique_ptr. Instead you instantiate them like this:

```
auto someRTClass = ref new SomeRTClass();
```

The use of 'ref new' is necessary to create a WinRT class instance.

You only need to write a WinRT class if you are writing a WinRT component that's meant to be used as a library in some other application. You don't need to do this for

classes that are directly in your game/application (though you can if you want; there's some overhead due to the automatic reference counting but it shouldn't be all that bad).

The following is an example of a WinRT class:

```
#pragma once

#include <unordered_map>
#include <collection.h>

using namespace Windows::Foundation;
using namespace Windows::Foundation::Collections;

namespace SomeComponent
{
    public ref class SomeRTClass sealed
    {
    public:
        SomeRTClass()
            : someStr_(L"")
            , someInts_(ref new Platform::Vector<int>())
        {
            // Do nothing
        }
        ~SomeRTClass() { }

        property Platform::String^ SomeStr
        {
            Platform::String^ get() { return someStr_; }
            void set(Platform::String^ value) { someStr_ = value; }
        }

        int GetKeyedNamesCount(void) { return m_keyedNames.size(); }
        Platform::String^ GetKeyedName(int key)
        {
            try
            {
                return ref new
Platform::String(m_keyedNames.at(key).data());
            }
            catch(...)
            {
                throw ref new Platform::FailureException();
            }
        }

        void SetKeyedName(int key, Platform::String^ value)
        {
            m_keyedNames[key] = std::wstring(value->Data());
        }
    }
}
```

```

    property IVector<int>^                SomeInts
    {
        IVector<int>^ get() { return someInts_; }
    }

private:
    Platform::String^                    someStr_;
    Platform::Vector<int>^                someInts_;
    std::unordered_map<int, std::wstring> m_keyedNames;
};
}

```

Anything in your component's public interface needs to deal in WinRT types (including fundamental types such as int; see: [http://msdn.microsoft.com/en-us/library/windows/apps/br212455\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/windows/apps/br212455(v=vs.110).aspx)).

Public collection types must be WinRT interfaces to collection types (i.e. IVector<T> instead of Vector<T>).

Private code can use non-WinRT types such as std::unordered_map.

Also, the sealed keyword is only necessary to use the WinRT component in JavaScript.

If you ever did any managed C++ coding, you will likely notice that C++/CX syntax is pretty much the exact same syntax as managed C++. Microsoft elected to reuse the syntax since it had already been approved as a language extension by the ECMA standards organization. But C++/CX is entirely native; .NET is not involved and there is no GC running in the background (WinRT classes are automatically reference counted, remember).

For more on C++/CX and WinRT (including all of the rules governing WinRT components), I recommend checking out <http://dev.windows.com/> (especially the Visual C++ reference for Windows Runtime page - <http://msdn.microsoft.com/en-us/library/windows/apps/br229567.aspx>).

Visual Studio and C++

Initial configuration

If you're using VC++ 2010 Express for the first time, there are a few settings changes I recommend making (especially if you are coming from C#).

First, in the "Tools" menu under "Settings" switch to "Expert Settings". I recommend this for all Express SKUs of Visual Studio.

Next, in "Tools"->"Options..." under "Environment"->"Keyboard" switch the drop down menu from "(Default)" to "Visual C# 2005". This will prevent you from going crazy when F6 doesn't compile. If you are more comfortable with a different key mapping, use that one instead. The point is to switch to keys that you are familiar with.

While in "Options..." change any other settings you like. One thing I like to do is under "Text Editor"->"C/C++", in "General" I like to turn line numbers on and in "Tabs" I've taken to setting "Insert spaces" rather than "Keep tabs".

IntelliSense

If you're using almost any of the Visual Studio keyboard mappings, typing Ctrl+J will bring up IntelliSense. In VS11 IntelliSense should appear automatically in C++. In VS 2010 and earlier, you need to manually invoke it.

Code snippets

I've never made too much use of code snippets, though I find myself using them more and more since I learned that the magic secret to accepting the parameters and starting to code is just to hit Enter.

Code snippets are a new feature for C++ in VS11; they don't exist in earlier versions. If you've never used them (in any language), in a C# project start typing 'for' to begin a for loop but once IntelliSense has chosen the for snippet, press the Tab key twice and watch as a for loop appears complete with automatic fields that you can edit (use the Tab key to switch between fields). When you're done editing the fields, press Enter and the cursor will be transported within the loop body with the field edits you made (if any) accepted and appearing now as normal text.

Including libraries

In C++, it's usually not enough to just include a header file. Normally you need to tell the linker to link against a library that implements the code that is declared in the header file. To do this, you need to edit the project's properties, accessible as "ProjectName Properties..." in the "Project" menu. In the properties, under "Configuration Properties" -> "Linker" -> "Input", one of the fields is "Additional Dependencies". This is a semi-colon separated list of the .LIB files you need to link against. It should end with

```
%(AdditionalDependencies)
```

so that any additional libraries that are linked via MSBuild are properly added. For a typical DirectX 11 Metro style game you might see the following:

Copyright © 2012 Michael B. McLaughlin

Page 50 of 53

```
d2d1.lib; d3d11.lib; dxgi.lib; ole32.lib; windowscodecs.lib;  
dwrite.lib; xaudio2.lib; xinput.lib; mfcore.lib; mfplat.lib;  
mfreadwrite.lib; mfuuid.lib; %(AdditionalDependencies)
```

Precompiled headers

A precompiled header (PCH) is a special type of header file. Like a normal header file, you can stick both include statements and code definitions in it. What it does differently is that it helps to speed up compile times. The PCH will be compiled the first time you build your program. From then on, as long as you don't make any changes to the PCH or to anything that is `#included` in the PCH, the compiler can reuse its pre-compiled version of the PCH. So don't stick anything in it that is likely to change a lot. But do add things that are unlikely to change. This way your compile times will speed up since a lot of code (e.g. Standard Library headers) will not need to be recompiled every build.

If you use a PCH, you need to `#include` it at as the first include statement at the top of every CPP file (but not at the top of header files). If you forget to include it or put some other include statement above it then the compiler will generate an error. This is just a result from the way the compiler needs to see PCHs in order to make it work.

Generating assembly code files

If you want to view (a very close approximation of) the assembly code that your code is compiled down into, in your project's properties, under "Configuration Properties" -> "C/C++" -> "Output Files" set the "Assembler Output" option to something other than "No Listing".

I'd recommend either "Assembly-Only Listing (/FA)" or "Assembly With Source Code (/FAs)". I normally use the former; it sprinkles enough line number comments that I can cross-reference to see what code I'm dealing with. The latter can be helpful if you want one place to see it all rather than flipping back and forth between whatever you've opened the .ASM file in (I use Notepad++) and Visual Studio.

Note that the assembly that is generated uses MASM macros (you can look them up on MSDN). If you don't know what a particular assembly instruction means (e.g. LEA), you can search the internet for it or try downloading the appropriate programming manual from Intel's site (assuming x86/Itanium) or AMD's site (assuming x64) or ARM Holding's site (assuming ARM). If you've never learned any assembly, I definitely recommend it (try just creating a simple Windows Console app). The course I enjoyed most out of all the Comp Sci classes I took in my undergrad minor in CS was where I learned MIPS asm.

Terrifying build errors

Chances are if you come across a build error that looks completely horrible, it's from the linker. You'll see messages like this, for instance:

```
Error 2      error LNK2019: unresolved external symbol "public: __thiscall
SomeClass::SomeClass(wchar_t const *)" (??oSomeClass@@QAE@PB_W@Z)
referenced in function "void __cdecl DoSomething(void)" (?DoSomething@@YAXXZ)
D:\VS2010Proj\CppSandbox\CppSandbox\CppSandbox.obj CppSandbox
```

All that's saying is that it cannot find some function you said it should be able to find. In this case, I added the 'inline' keyword to a constructor function definition that was in the CPP file without remembering to relocate that definition to the header file. Any inline functions need to be in the header so that the linker won't hate you.

All those ?? and @@ and weird letters are just the way that C++ mangles names when it has compiled code into object files. Name mangling is internally consistent for the compiler in question but the ISO/IEC standard doesn't mandate any particular schema for name mangling such that different compilers can (and often will) mangle things differently.

Anyway, normally if you see some sort of horrifying build error message, chances are good that it's from the linker and that it's an unresolved symbol error. If so, if it's saying it can't find something that you wrote (in the above case my `SomeClass::SomeClass(wchar_t const *)` constructor function (I always write 'const type' not 'type const' so even that bit is reconstructed)) then check to make sure that your declaration (in the header file) matches the definition (usually in the code file but maybe you put it in the header or maybe you forgot to write it or maybe you declared it inline but still have it in the code file). If it's someone else's function (or other symbol), then chances are that you didn't tell the linker about the .lib file that contains it.

In .NET you just add a reference to an assembly and you get both the declaration bits and the actual definition stuff all in one. In C++, the declaration is the header file while the definition stuff (excluding inline stuff, which needs to be in the header file too) is in a separate library. See above about including libraries. Search the MSDN library for the symbol that it's telling you it is missing and see if you can find the name of the library file you need to add.

C++ build errors can look pretty scary. Especially when you get a build error involving a template... those can make you want to quit. But don't. Never let the horrible error messages win.

First figure out if it's coming from the compiler (it'll have a C#### error number format) or the linker (LNK#### error number format).

The compiler usually means some sort of syntax error. Check to see things like whether you forgot the `#pragma` once at the top of your header file. Another problem could be where you are using something from the standard library (e.g. `'endl'`) but forgot to have either a `#using namespace std`; or else to prefix it with `std::` (i.e. `'std::endl'`). You can do either (or both) but must do at least one. And some things might be in a different namespace (in VS 2010, some functionality is in the `stdext` namespace, for example). The same goes for any namespaces you might be using in your own code.

If you aren't having any luck on your own, try going on MSDN and typing in the first part of the error message. Chances are good that you'll get some helpful links to discussions on the MSDN forums, on StackOverflow, perhaps an MSDN article or an MSDN blog post, ... maybe even just the error code's page itself will have the hint you need. If all else fails, post a question on a forums site (MSDN, the appropriate StackExchange site, the App Hub).

A linker error is typically an unresolved symbol, which usually means you either have a mismatch in declaration and definition, have an inline outside of its header, or else don't have the right library added to the project's extra dependencies in the project's linker options. If it's something else, try the strategies from the previous paragraph; they apply just as well to linker errors as to compiler errors.