





C++ Advice and Information

CSCI 104 Teaching Team

January 31, 2014

Abstract

This document is a collection of useful bits of information related to programming tasks you will encounter throughout the semester. You will be working in C++, one of the most powerful and challenging modern programming languages. It is our hope that the information here will make it easier to understand the mistakes you make and eventually to avoid making them altogether.

Throughout the guide, helpful information is marked by , style suggestions are marked by , C++11 specific information by , and other information by .

Contents

1	Useful Background Information	3
1.1	Incredibly Useful Computer Basics	3
1.1.1	Number Systems	3
1.1.2	Counting	4
1.2	Memory	4
1.2.1	Endianness	5
1.3	The Compiler	6
1.3.1	Step One: The Pre-Processor	7
1.3.2	Step Two: The Compiler	7
1.3.3	Step Three: The Assembler	8
1.3.4	Step Four: The Linker	8
2	C++ Coding Tips and Style	10
2.1	C++ Reference	10
2.2	The Pre-Processor	10
2.2.1	Macros	11
2.2.2	Include Guards	12
2.3	Namespaces	13
2.3.1	The using keyword	14
2.4	Classes and Structs	15
2.4.1	Access Control	15
2.4.2	Friendship	16
2.5	Const	16
2.5.1	Constant Variables	17
2.5.2	Constant Member Functions	17
2.6	Passing by Value vs. Reference	18
2.6.1	Passing by Value	18
2.6.2	Passing by Reference	19
2.7	Pre and Post-Increment and Decrement	20
2.8	Measuring Program Speed	21
2.8.1	Basic Method	22
2.8.2	Iterative Method	22
2.8.3	Using C++11	24

3	How to Fix Your Code	25
3.1	C++ Compiler Error	26
3.2	C++ Linker Error	27
3.3	Run-Time Error	31
3.3.1	Basic Debugging	32
3.3.2	Advanced Debugging	32
3.4	Logic Error	34
3.5	References	34
4	Collected Questions and Answers	35
4.1	Templates	35
4.1.1	Linker Error	35
4.2	Exceptions	36
4.2.1	Throwing a Runtime Exception	36
4.2.2	Exceptions in While Loops	38
4.3	References	39
4.3.1	Passing a Pointer by Reference into a Function	39
4.3.2	Member Variable is Private	40
4.4	Virtual Functions	41
4.4.1	Constructors and Virtual	41
4.5	Iterators	42
4.5.1	General Iterator Confusion	42
4.6	Inheritance	44
4.6.1	Initialization Lists	44
4.6.2	Is-a string	45
4.6.3	Protected Variables	46
4.7	Const	48
4.7.1	error: passing 'const Wall' as 'this' argument of 'WallPost Wall::get(int)' discards qualifiers	48

Chapter 1

Useful Background Information

Before you delve into advice on style or substance on C++, it is useful to build up a small amount of knowledge about how the computer and the compiler work. This information will help you understand 'the *why*' of various errors that you will encounter. Ultimately you understand how your computer actually works, at a very basic and high level. As you progress through your education these ideas will be clarified.

Note

Don't get too intimidated by the information in this section - you only need to be familiar with it at a very high level. You will eventually build a detailed knowledge of it as you progress through your education.

1.1 Incredibly Useful Computer Basics

This section is a high level overview of some concepts that will ultimately help you better understand and fix common mistakes novice C++ programmers make. You will still make mistakes, but at least you'll be able to understand what they mean.

1.1.1 Number Systems

Computers operate on a different numerical base than what you use in your everyday life. Since computers are ultimately implemented using transistors and electricity, they operate on a system that considers only two values - on (1)

and off (0). The number system you are used to thinking in has been base 10 (the base referring to how high you need to count to need a new digit).

Being familiar with these number systems will benefit you greatly when you have to reason about memory and perform debugging.

Binary

The numeral system that computers use to perform all computation is called binary. Binary is base 2, which means that it only has two digits: 1 and 0. You won't need to know too much about binary for this course, other than knowing that eventually everything you do on your computer makes its way down to binary. Your computer does all of its arithmetic, represents the characters you are reading, and even the colors of your screen in binary. Some example binary numbers: 0b1 is equivalent to 1 (base 10, assume if no prefix the number is base 10), 0b1001 is 9, and 0b10 is 2.

Hexadecimal

Also known as hex, this is a base 16 numeral system, where the digits A, B, C, D, E, and F represent the base 10 values 10, 11, 12, 13, 14, and 15. Hexadecimal is usually written with a 0x prefix to the numbers, e.g. 0x10 is the number 16 in base 10.

1.1.2 Counting

In C++ counting always begins at zero, not at one. So the first index of something like an array is at location zero, while the second item is at location one. The why for this has to do with memory addresses, which we will go over briefly in the next section.

1.2 Memory

Your computer stores information in a hierarchy of components, with the CPU being at the top, and disk being at the bottom. As you move up the hierarchy, the amount of space gets smaller but the speed gets much faster. For this class, all you really need to know is that somewhere along the line your computer stores information in memory.

You can think of memory as a big array (solid contiguous block) of information (1s and 0s). Since everything on the computer is ultimately represented in binary, it can all be stored in memory. Since counting in binary is tedious,

memory addresses are most often represented by a hexadecimal (base 16) number. One hexadecimal digit represents four binary digits, e.g. 0b1010 (10) is 0xA.

When talking about memory, we call one binary digit one **bit**. When we have enough bits, we call it a **byte**. The number of bits in a byte is dependent upon your computer architecture, but for this class you can safely assume that this number is 8 (eight bits in one byte). We can represent a byte using two hexadecimal numbers, e.g. 0x31 is 0b00110001. A byte is generally the smallest unit of information your computer can allocate.

We can now think of memory as a large array of bytes, where the first location (one byte or 8 bits) in memory is addressed by 0x0, and the second address by 0x1 (see section 1.1.2 on counting).

Useful Information

Information is stored in memory as a series of 1s and 0s known as **bits**. It generally takes 8 bits to represent 1 **byte**, the smallest unit of allocation on a computer. You should think of memory as an array of these bytes, with the first entry indexed by memory address 0x0.

1.2.1 Endianness

Note

Unless you are serializing data in a binary format (reading or writing from disk in binary, or over the network), you generally don't need to worry about endianness.

If we want to represent some number that is too big to fit into a byte (its binary representation would take more digits than a byte can hold), how would we do that? Let's consider the number 256, which is one more than the maximum number we can represent using 8 bits, or 1 byte. The hexadecimal for this number is 0x100. However, when we store things on the computer, we need to fit them into the smallest units of allocation, bytes. This number is three hexadecimal digits, so the smallest amount of space we can fit it in is two bytes, which would be four hexadecimal digits: 0x0100 (note the leading zero). We call the last byte we needed to add to represent the number the *most significant bit* (or byte, usually written as MSB).

When the computer puts a number like 0x0100 into memory, it has two choices for how it can order the bytes: big-endian, and little-endian:

Big Endian

In big-endian encoding, the MSB is ordered first in memory, so if we were to take the number 256 and represent it in this fashion, at memory address zero (0x0) we would have the MSB (0x01), and at memory address one (0x1) we would have the next most set of significant bits, which in this case is just the remainder of the number (0x00). So the memory would be laid out like: 01 00, with 01 being at the lowest memory address.

Little Endian

In little-endian encoding, the MSB is ordered highest in memory, meaning that the LSB (least significant bit) comes first. For our example of 256, this would put the 0x00 portion first, followed by the 0x01 portion. So the memory layout would look like: 00 01, with 00 being at the lowest memory address.

Confused? Check out the Wikipedia article¹ with pretty pictures!

1.3 The Compiler

The compiler is the magic box (it's actually a program) that turns the code you write into a program your computer can actually execute. Knowing a little bit about the compiler, namely the stages of compilation, will help you write good C++ code.

There are two types of files you can write for the compiler: header files and source files:

Header files (usually `.h`, `.hpp`, or `.H`) are files that are `#included` in other files. In general you should never include source files and only include header files. The file extension doesn't matter other than matching coding conventions - stick to `.h`, `.hpp`, or `.H` for your C++ header files. Header files are never directly compiled themselves, they are only compiled after being included into source files.

Source files (usually `.c`, `.cpp`, or `.C`) are files that are *compiled* by the compiler. What this means exactly will be detailed in the next few sections. Source files are almost never included in other source files.

What happens when you compile a program with something like `g++ -std=c++11 myfile.cpp main.cpp -o myprogram`? The compiler will hopefully end up cre-

¹<http://en.wikipedia.org/wiki/Endianness>

ating a program called `myprogram` if there are no errors, but how does it get to this step? There are four steps to compilation²:

1. The pre-processor
2. The compiler
3. The assembler
4. The linker

When you compile a program, it goes through stages 1 to 3 for **every source file** you tell it about. Stage 4 only happens once after all source files have already gone through stages 1 through 3. This means that for the previous example, both `myfile.cpp` and `main.cpp` will go through steps 1 through 3 independently, and then be combined together into your program in step 4. The individual files are referred to as *translation units* or *compilation units*.

Useful Information

Remember that each source file goes through steps 1 through 3 independently and that they are only combined through the linker in step 4 once all source files have successfully made it through step 3. We call files that have made it through step 3 *compiled*. Each file, during its compilation process, is referred to as a *translation unit* or *compilation unit*.

1.3.1 Step One: The Pre-Processor

The pre-processor is the first stage of compilation and operates on any line that begins with a pound/hash sign (`#`). Depending on the command following the `#`, the pre-processor will do some action. The important thing to remember is that at its core, *the pre-processor can only cut, copy, and paste text*.

Whenever you type `#include <something.h>`, the entire contents of `something.h` are copy and pasted to that location in your file. If you use macros such as `#define SQUARE(x) x*x`, these will be evaluated by copying and pasting arguments at the text level.

The pre-processor operates directly on the written source code and finishes copying and pasting before the compiler even runs. More detail on this can be found in the C++ pre-processor section (section 2.2).

1.3.2 Step Two: The Compiler

After the pre-processor has performed its textual substitutions, the compiler is run. The compiler is a program that will take your human readable source

²<http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html>

code and turn it into significantly less readable assembly code. Assembly code is the lowest level of instruction that is still human readable and maps directly to binary instructions the CPU can execute.

The compiler will perform a large amount of error checking to help ensure that the program you write is valid and will behave as you expect it to. Compilers can generate two kinds of diagnostic messages: warnings and errors.

Warnings are emitted for code that could potentially cause problems but is not guaranteed to result in an issue. It's a good idea to try and get your code to compile with no warnings, as these often eventually manifest into run time or logic errors. Try compiling with all warnings turned on (`-Wall`) as well as warnings as errors (`-Werror`) if you want to avoid letting warnings pass through.

The other diagnostic is a compiler error. This is usually some syntax error you have made that is not legal C++. The compiler will do its best to tell you what you did wrong, but learning to decipher compiler messages is something of an art you will have to slowly learn through experience. Take a look at section 3 for some help on debugging and fixing errors.

1.3.3 Step Three: The Assembler

The assembler is a step that will take the file generated by the compilation stage (usually a `.s` file), which is written in assembly code, and generate *object code*, which is in a binary format the CPU can directly understand.

Once a file has passed through this stage, it is considered to be fully *compiled*. All that remains is for it to be *linked* in the last stage. Remember that each source file proceeds up to this point independently.

1.3.4 Step Four: The Linker

Once every single source file you told the compiler about has been fully compiled, it is the linker's job to combine them all together and create the final executable program. The linker operates on all files simultaneously unlike the preceding three steps which each operated on a single file in isolation. The purpose of the linker is to match up function calls and variables that span multiple translation units (source files). The linker does this by using a system of *imports* and *exports* of function/variable names.

imports are any function calls or external variables you use that do not have local implementations (in the same source file/translation unit). These are put on a 'import list' that is a request of names that other translation units must supply in order for the linking to be successful.

exports are any globally visible (global scope) functions or variables. When a single source file is being compiled, any globally visible function or variable, after performing all **#include** and other pre-processor commands, will be given to the linker as a list of so-called exported names. These exported names can then be matched to import requests in other translation units.

If the linker cannot match every import to an available exported name, you will get a translation error. Translation errors are difficult to read and usually mention undefined references. These undefined references are the missing exported names.

Chapter 2

C++ Coding Tips and Style

This chapter covers some standards for styling your C++ code and some general tips for programming in C++. You should definitely read chapter 4 before diving in.

2.1 C++ Reference

One of the best resources for information on C++ is [cppreference.com](http://en.cppreference.com).¹ You can find language information² as well as standard library information³. A good secondary website is [cplusplus.com](http://www.cplusplus.com)⁴

Remember that while you are allowed to seek information on the internet and discuss problems with classmates, the solutions you ultimately present must be entirely your own. Do not take code you find online. If you do read code online, sleep on it before implementing it yourself, and do it from scratch. It is very easy for us to detect code that looks out of place or is taken from the internet or other students.

2.2 The Pre-Processor

The pre-processor is part of the C++ compiler that runs before any code is compiled. Pre-processor directives are prefixed with the hash symbol (#) and include things such as macros, includes, and compiler specific directives.

¹<http://en.cppreference.com>

²<http://en.cppreference.com/w/cpp/language>

³<http://en.cppreference.com/w/cpp>

⁴<http://www.cplusplus.com/>

2.2.1 Macros

Macros are ways of performing textual substitution inline in your code. These are much more commonly used in C than in C++ where templates and inline functions perform many of the same duties. Because these perform pure textual substitution, they can lead to logic errors that can be hard to spot:

Figure 2.1: Macro Pitfall

```
#define SQUARED(x) x*x

// in C++03 and earlier:
inline int squared(int x) { return x*x; }

// in C++11 and later,
// mark as a constant expression
inline constexpr int squared11(int x) { return x*x; }

int main()
{
    int y = 2;

    // macros perform raw textual substitution
    int y_squared = SQUARED(y++);
    // what is the value of y now?
    // what is the value of y_squared?

    int z = 2;

    // inline functions will often be optimized away
    int z_squared = squared(z++);

    // constexpr can happen at compile time
    int z_squared2 = squared11(z++);

    return 0;
}
```

Style Guide

Avoid the use of pre-processor macros unless absolutely necessary. Use C++ idioms such as inline functions, templates, and constexpr (C++11) instead.

★ C++11 Tip

In C++11, there is a new keyword `constexpr`^a that denotes functions that are capable of running at compile time, which you should use whenever possible.

^a<http://en.cppreference.com/w/cpp/language/constexpr>

2.2.2 Include Guards

C++ lacks a module system and all files included using `#include` are effectively pasted where the `#include` directive is, meaning that multiple `#includes` for the same file will cause it to be included several times, leading to naming conflicts.

The way to circumvent this is by using include guards, which is a name for pre-processor directives designed to detect a file has already been included and prevent it from being processed more than once in the same compilation unit.

These include guards need to be placed in every file that is intended to be `#included` from another one (such as headers). There are two ways of doing this:

1. Using `#ifndef`, `#define`, and `#endif`:

```
// use some unique name to identify the file
// usually the path from your project
// base directory and the file name
#ifndef PATH_TO_MYFILE_MYFILE_HPP_
#define PATH_TO_MYFILE_MYFILE_HPP_

// actual code goes here

#endif // PATH_TO_MYFILE_MYFILE_HPP_
```

The pre-processor will only allow the code to be compiled if the token is not defined. Including the file will cause the token to be defined, preventing a subsequent inclusion from causing the code to be duplicated.

2. Using `#pragma once`:

```
#pragma once

// actual code goes here
```

This is a much more succinct but is not as portable as the previous method; check with your compiler to see if this works.

Style Guide

Prefer the use of the first solution (`#ifndef`, `#define`, and `#endif`) since it is more portable.

2.3 Namespaces

The first place that you will likely see namespaces is the C++ standard template library (STL), which uses the namespace `std`. Namespaces are a way of organizing similar code into named hierarchies.

It is a good idea to keep all of the implementation code you write in its own namespace and use nested namespaces to organize things as necessary. Namespaces help avoid naming collisions and give an identity to the code they contain.

Namespaces only need to be explicitly stated when accessing something while outside of the scope of that namespace. If you are working inside of your own namespace, you can omit it while referencing other entities. The compiler will search all namespaces, starting with the most nested one:

Figure 2.2: Namespace examples

```
namespace // anonymous namespace
{ int x; }

namespace mycode
{ int x; }

int main()
{
    int y = x; // refers to anonymous x
    int z = mycode::x; // access mycode x
    return 0;
}
```

You can also declare anonymous namespaces by opening a namespace with no name. These can only be accessed from the same scope in which they were declared with no access specifier needed (see figure 2.2 for an example). Anonymous namespaces can also be used to hide declarations from other files if they are `#included`.

Style Guide

Keep your code in its own namespace and use anonymous namespaces for local functions that don't need to be seen elsewhere.

2.3.1 The using keyword

The keyword `using` is used to bring a namespace into the current scope, meaning that you can use it as if you were already inside of it. This also means that it is possible to accidentally pollute namespaces by performing `using` in global scopes, so you should only use `using` in either local scopes or in source files that will not be `#included` in any other file.

Figure 2.3: using namespace examples

```
#include <vector>

namespace mycode
{ int x; }

int main()
{
    // bring std namespace into this scope
    using namespace std;

    // std namespace has been brought into scope
    vector<int> myVector;

    // can still use explicitly
    std::vector<bool> myBoolVector;

    {
        // using is always local to current scope
        using namespace mcode;
        int z = x;
    }

    // scope expired, using does not cover here
    int w = mycode::x;

    return 0;
}
```

One major disadvantage to `using` is that it makes it harder to understand what is happening when looking over code. In the case of long namespaces, these can be aliased to a shorter name by using namespace aliases⁵ (Figure 2.4).

The same rules for scoping apply for alias - it is only valid for the scope it is performed in and should not be done anywhere where it can pollute the global scope.

⁵http://en.cppreference.com/w/cpp/language/namespace_alias

Figure 2.4: namespace aliasing

```
namespace long_name_1
{
    namespace long_name_2
    { int x; }
}

int main()
{
    namespace short=long_name_1::long_name_2;
    return short::x;
}
```

Style Guide

Only use `using` in a local scope or file that will not be `#included` by another file (e.g. keep it inside a `.cpp`). Prefer to avoid using `using` to keep code more readable.

2.4 Classes and Structs

Classes and structs are identical in C++ aside from the default visibility of their members (member functions and variables). Classes are *private* by default, and structs are *public* by default. In idiomatic C++, structs are generally used for lower-level containers where access control is less important. Classes are typically used when it matters what data users have access to and is generally used with anything that is more complicated than a collection of variables.

2.4.1 Access Control

There are three types of access control for classes and structs:

1. `public` - access is available to anyone
2. `protected` - access is only available within the class and to derived children
3. `private` - access is only available within the class

When you inherit from another class, you can specify an access control to be applied to the entire class inherited, which follows rules as seen in figure 2.1.

Table 2.1: Access Control During Inheritance

Parent Access Declaration	Inheritance Access Declaration	Access as seen by derived class
public	public	public
public	protected	protected
public	private	private
protected	public	protected
protected	protected	protected
protected	private	private
private	public	private
private	protected	private
private	private	private

2.4.2 Friendship

In C++, you can get around the access requirements of a class by declaring *friend* classes. Once you declare some other class or function as a friend, it sees all of the data members of the declaring class as public, regardless of what they actually are. This is especially useful when working with iterators, which we often want to conceal the true implementation of a class while allowing easy traversal through data.

The syntax for declaring a friend is identical to the syntax for performing a normal declaration (when you say what something is but don't define it), except that you prefix it with the keyword **friend**:

```
template <class T>
class MyClass
{
    // declare a function as a friend
    friend void printClass( MyClass & );

    // declare some other class as a friend
    friend struct AnotherStruct;

    // declare some templated class as a friend
    // note we don't re-use T here
    template <class U> friend class TemplatedClass;
};
```

2.5 Const

When something is constant, it means that it cannot be changed. There are two big ways that this is used in C++, the first is to make a variable constant,

and the second is to make a class member function constant.

2.5.1 Constant Variables

When a variable is constant, it means that it cannot change its value. This is useful because it can prevent you (the programmer) from accidentally changing a value that should not be changed. It can also let the compiler perform special optimizations since it knows that a value will stay constant. Your attitude towards using the `const` keyword should be to apply it to everything, and treat cases that cannot be made constant as exceptions. In addition to the benefits already listed, many actual libraries and data structures implemented in C++ offer improved performance when operating in a constant fashion.

```
const int x = 3; // value can not be changed
const int * y = &x; // what we point to can't be changed,
                  // although we can change the pointer itself
const int * const y = &x; // both what we point to and the pointer
                          // itself are constant
```

Style Guide

Try to make every variable (including parameters) `const` unless you have a reason to let it stay mutable. Also try to apply this philosophy to class member functions.

2.5.2 Constant Member Functions

Another important way `const` can be used is to make a member function of a class or a struct constant. When used in this context, the constancy applies to any member variables used within that function. The `const` applied to the function says that you will not change the value of any class member variable inside of that function. You can still change local variables inside the function (that are not themselves `const`), but you will be unable to change the value of any member variables.

```
struct Struct
{
    int doSomething() const
    {
        // can still change local variables
        int localVariable = 2;
        localVariable += 1

        // not allowed to change _memberInt
        return _memberInt + 3;
    }
}
```

```

    }

    void increment()
    {
        // not const, can change member variables
        ++_memberInt;
    }

    int _memberInt;
};

```

Style Guide

If a class member function does not need to change any member variables, mark it const.

2.6 Passing by Value vs. Reference

When writing a function, you have two choices on how to construct the arguments. Arguments can accept parameters by either *value* or *reference* (as of C++11 you'd actually have three choices, but you don't need to worry about that for this course).

2.6.1 Passing by Value

When you have an argument that accepts a parameter by *value*, it performs a copy of the data when it is passed in. Something is considered to be passed by value if there is no reference (no & symbol) in the declaration.

```

void myFunc( int paramOne, bool * paramTwo, std::vector<double> paramThree );
// all of the above parameters are done by value, since there is no & present

```

Note that since passing by value makes a copy, it can be extremely inefficient if you do not need to make this copy. Consider a function that sets some member variable in a class:

```

struct MyStruct
{
    std::vector<double> x;
    void setX( std::vector<double> xx ) // copy #1
    {
        x = xx; // copy #2
    }
};

```

```

MyStruct s;
std::vector<double> vec(1000);
s.setX( vec );

```

In the above example we want to set `x` in the struct to equal `vec`. If we pass by value here, we copy the entire vector (and all of its 1000 elements) two times - first when we pass it to the function, and then again when we assign it to the member variable. Ideally in this situation, since we are not doing anything to the parameter in the function, we would like to avoid this extra copy and performance hit. This is where passing by reference (specifically const reference) helps us.

Style Guide

If you will be changing the value of a parameter, or if it is smaller than the size (sizeof) a pointer, feel free to pass it by value.

C++11 Tip

In C++11 there is extra optimization that can sometimes occur where valued are *moved* rather than being copied in situations like the one presented above. This is an advanced topic that is outside the scope of the course.

2.6.2 Passing by Reference

When we pass a variable by reference, we are essentially passing a pointer to that variable instead of making a copy of it. This is advantageous as soon as the size of the object becomes larger than the size of a pointer, which will be true for most structs and classes. References give us the flexibility of pointers without having to delve into pointer syntax. When we change a reference, it changes the original value:

```

struct MyStruct
{
    std::vector<double> x;
    void setX( std::vector<double> & xx ) // no copy, using reference
    {
        xx.push_back( 32.2 ); // changes the original vector
        x = xx; // copy #1 (includes the insertion we made)
    }
};

MyStruct s;
std::vector<double> vec(1000);
s.setX( vec );
// vec is now size 1001

```

In the above example, we have eliminated an extra copy that occurred when we passed the vector by value. Since we passed by a reference, any change we make to the parameter also changes the original vector we passed in. If we don't need to change the value of the parameter, and want to avoid making an extra copy, we should use a const reference:

```
struct MyStruct
{
    std::vector<double> x;
    void setX( std::vector<double> const & xx ) // no copy, using const reference
    {
        // cannot modify xx as it is const
        x = xx; // copy #1
    }
};

MyStruct s;
std::vector<double> vec(1000);
s.setX( vec );
// vec is unchanged
```



Style Guide

If you will not be changing the value of an argument and it is larger (sizeof) than a pointer, pass it by const reference.

2.7 Pre and Post-Increment and Decrement

Two very common operators that you will see used in C++ code are the increment and decrement operators (`++` and `--`, respectively). For basic data types, these operators either increment the value by one or decrement the value by one. For user-defined types, such as iterators, the behavior is usually the same but may be slightly different.

Regardless of whether you are using these operators on a basic type or a class, it is important to know the difference between putting the operator before (pre) and after (post) a variable.

When you use pre-increment, the variable is changed immediately, and no copy is made:

```
int x = 3;
int y = ++x;
// y = 4, x = 4
```

When you use post-increment, the variable is not changed until after the entire statement is executed:

```
int x = 3;
int y = x++;
// y = 3, x = 4
```

In order for this post-increment to happen, a copy of the variable is created and used in its place so that the original value can be changed without affecting the behavior of the current statement. This means that there is more overhead associated with the post-increment operator in addition to its unconventional behavior of delaying the increment. You should only use post-increment when you desire this special behavior, and use pre-increment (or decrement) in every other circumstance.

Style Guide

Always use pre-increment (or decrement) unless you desire the specific behavior that post-increment (decrement) gives you.

2.8 Measuring Program Speed

Throughout the semester, you will be learning about theoretical analysis of the speed of various algorithms and the impact of data structures. It is very useful and illuminating to complement the theoretical analysis with practical measurements. Here, we point you to some simple techniques for measuring how long a program takes.

The basic idea behind any time measurement is a stopwatch. You record the CPU time at the beginning of your operation and again at the end. Modern CPUs have a fast clock so you can measure any operation that is at least 1 millisecond. In Section 2.8.2, we show you how to measure faster operations (and improve the precision even on slower measurements).

It is worth mentioning that the measured time is not purely from your program. Most Operating Systems tend to schedule and run many tasks simultaneously. Therefore it is recommended to check if another program is extensively consuming your CPU in the background. Also it is common practice to only measure the processing time of the program and not the I/O portion, because that is often much slower, and not indicative of the data structure or algorithm.

Besides the usual C++ input/output libraries, in this guide we need the `time` library as well. This *include* file has many useful classes to store time. In order to familiarize yourself with this library check out <http://www.cplusplus.com/reference/ctime/>.

2.8.1 Basic Method

In order to programmatically measure the elapsed time, we call the `clock()` function once at the beginning and again after the process is done. This function returns the CPU clock which later can be interpreted as seconds. The code in figure 2.5 demonstrates the procedure⁶.

Figure 2.5: Basic Time Measurement

```
#include <iostream>
#include <ctime>

using namespace std;

int main()
{
    clock_t start, finish;
    double dur;

    start = clock();
    // do something slow...
    finish = clock();

    // Measure the elapsed CPU time.
    dur = (double)(finish - start);
    dur /= CLOCKS_PER_SEC;
    cout << "Elapsed seconds: "
         << scientific << dur << endl;

    return 0;
}
```

The code in Figure 2.5 outputs the time in scientific format. Note that only the slow operation (line 12) is being measured and not the entire program.

2.8.2 Iterative Method

As you may have noticed if you implemented the code above, the reported times are not always the same for the same operation. Additionally, many operations are too fast to be measurable at all like this. To overcome this challenge we can repeat the same operation and then report the average elapsed time. This tends to give a more accurate measurement.

The code in Figure 2.6 demonstrates how we can measure more accurately. We are repeating the same operation 1000 times (or choose a larger constant if the

⁶Taken from here.

Figure 2.6: Iterative Time Measurement

```
#include <iostream>
#include <ctime>

using namespace std;

int main()
{
    const int ITERATE = 1000;
    clock_t start, finish;
    double dur;

    start = clock();
    for (int i = 0; i < ITERATE; i++)
    {
        // do something not so slow...
    }
    finish = clock();

    dur = (double)(finish - start);
    dur /= (CLOCKS_PER_SEC * ITERATE);
    cout << "Elapsed seconds: "
         << scientific << dur << endl;

    return 0;
}
```

Figure 2.7: C++11 Time Measurement

```
#include <chrono>
#include <iostream>

int main()
{
    // get the current time
    auto start =
        std::chrono::high_resolution_clock::now();

    for (int i = 0; i < ITERATE; i++)
    {
        // do something not so slow...
    }
    auto end = std::chrono::high_resolution_clock::now();

    // output the elapsed time
    std::cout << "Elapsed seconds: " <<
        std::duration_cast<std::chrono::seconds>(
            end - start ).count()
        << std::endl;
    // we could also duration_cast this to
    // milliseconds or anything else

    return 0;
}
```

operation is very fast). At the end, we divide the total duration by 1000. Of course we are also measuring the `for` loop itself but that is usually negligible.

2.8.3 Using C++11

If you are using C++11, you can easily measure time using functionality built into the standard library in the `<chrono>`⁷ header. Take a look at the code in figure 2.7 for an example of this.

★ C++11 Tip

C++11 has a high resolution clock^a that requires no external libraries and will work on any Operating System (OS) with a compliant C++11 compiler.

^ahttp://en.cppreference.com/w/cpp/chrono/high_resolution_clock/now

⁷<http://en.cppreference.com/w/cpp/header/chrono>

Chapter 3

How to Fix Your Code

This section covers a systematic method to read, understand, and fix the common types of code errors. By now you should have learned how to setup your C++ programming environment. More specifically, you should know how to use a professional IDE to edit code, build it, and debug the executable. There are four types of errors a C++ programmer may encounter:

- **Compile Error:** Also called syntactical error is when the compiler literally does not understand what you typed. It's like a spelling error in human language.
- **Link Error:** Is when the code is understood but the linker cannot find everything it needs. It's like a sentence without a verb.
- **Run-Time Error:** Once your binary (executable) is produced, your program can run. If it crashes, goes into infinite loop, or becomes unresponsive, you're having a run-time error.
- **Logic Error:** Is when the program runs but does not produce the expected output.

To reach enlightenment (as a programmer of course) first you have to understand these errors. Then you can fix them. In order to fix any of these errors, you have to follow the **Four Noble Truths**¹ of code fixing:

Pillar 1: The truth of bug: There is suffering and sleepless nights until the code is complete. This is the first step toward enlightenment so don't be sad.

Pillar 2: The truth of the origin of bug: The first thing you do is to isolate the error. Find which part of the code is causing the bug. For each type of error, there are tools to help you isolate the problem.

¹http://en.wikipedia.org/wiki/Four_Noble_Truths

Pillar 3: The truth of the cessation of bug: After isolating the bug, ask yourself why this is happening. Once you discovered the reason behind the bug, write a testing unit (function) that can deterministically reproduce the bug in an isolated environment.

Pillar 4: The truth of the path leading to the cessation of bug: Now it's time to fix the bug. Meditate on the logic behind the bug. A bug that is fully understood is a fixed bug.

The rest of this section briefly covers standard methods to handle coding errors. We will not cover specific errors and fixes. We rather show you the way to isolate the problem and how to look up more information about it. For obvious reasons, one cannot write a document to cover all possible coding problems.

3.1 C++ Compiler Error

When you try to build a C++ project, you are basically asking the compiler to read your code, understand it, and then produce an executable (=binary). A compiler like GCC² does that in multiple steps: parse, compile, and link³. Once the compiler sees an error in your code, it stops advancing to the next step. Instead it generates lots of compile and warning error messages with exact location of the problem.

```
$ g++ code.cpp -g -o code
code.cpp: In function 'int main()':
code.cpp:25: 'DoublyList' does not
name a type (first use this function)
code.cpp:25: parse error before ';' token
bla bla bla
```

Sometimes there might be lots of these errors for a single project. But fear not, follow these steps:

Step 1: Only read the **first** error message

Step 2: Go to the code file with the error (pillar 2)

Step 3: Look at the line that the compiler is referring to (in our example it's code file `code.cpp` line number 25)

Step 4: These types of errors are usually understandable (pillar 3). If you don't see the problem immediately, look at the surrounding code lines.

Step 5: If you don't understand the error message, seek help! Even if you managed to fix the error, don't jump over pillar 3.

²<http://gcc.gnu.org/>

³GCC is a huge open-source project with around 2 million lines of code. If you like to know about its internals this page is a good start.

Step 6: Re-compile

Step 7: Goto step 1

Things to avoid doing:

- Once you've fixed the first error, don't continue with the second one! Compile again.
- Do not undo your last edits unless the section you recently edited happens to be the faulty code.
- Don't continue writing new code either. First fix the errors. They won't go away if you ignore them.
- Don't comment out the faulty code line. If you see a spelling error in a document do you delete it!?

3.2 C++ Linker Error

Once you've fixed all the compile errors, then the compiler will try to link your modules together. At this step you may get link errors. You will not get any link errors until you fix all your compile errors. These are slightly harder to fix because the linker can't tell you the faulty code line. Additionally, the linker error messages are a little cryptic and very general. It will however tell you the problematic function and/or class. The following is a common link error.

```
$ g++ code.cpp -g -o code
/tmp/cc2Q0kRa.o: In function 'main':
/tmp/cc2Q0kRa.o(.text+0x18): undefined
reference to 'Print(int)'
code: ld returned 1 exit status
```

we know this is a linker error because the last line says "ld returned 1 exit status". If you use your own makefile, you may recognize the object file (`cc2Q0kRa.o`). The following list summarizes some of the common mistakes that may lead to a link error:

- How you build: For example if you compile both `.cpp` files and `.h` files together you usually get a link error. For example this is wrong:
`g++ code.cpp code.h -g -o code`
As a rule of thumb, never `#include` a `.cpp` file and never compile a `.h` file!
- If you define a function in a `.h` file, you have to code it in a `.cpp` file unless it's a pure virtual function in an abstract class. No other exception.
- You have to implement all defined constructors and destructors including the default ones even if they are not doing anything.

- The function signature has to exactly match the one in the header file except the `virtual` keyword.
- Re-definition Error: If something is defined (coded) twice you get a link error. usually you define your modules once but if you don't properly guard your headers or confuse the compiler, it may look like the function exists in two different places. You may get the same error if you put your function definition in a header file, or `#include` your implementation file in another code or header file. Mostly you can avoid this type of error with a well-written makefile. Note that you can declare your functions multiple times and that's OK.
- Most advanced C++ compilers will not compile your unused module which means you won't see all the link (and compile) errors until you call/reference it. Imagine this scenario: you have just finished coding the class `A` and fixed all its compiler errors. Now you write something like `A newObject;` in your `main` function and it causes new compiler errors about `A`. Is this the variable declaration error? The answer is no!
- If you're coding a template class, "undefined reference" link error can occur even if you follow the mentioned guidelines. There are at least three workarounds for this; (1) rename your class implementation file into a `.h` file then include it in the class header file, (2) pre-define all instances of your template class, and (3) implement all functions inside the class definition. For example the following code will give you "undefined reference" link error.

```
// list.h
template <class T>
class List {
    void foo();
};

// list.cpp
#include "list.h"
template <class T>
void List<T>::foo() { ... }

// main.cpp
#include "list.h"
int main() {
    List<int> i;
    return 0;
}
compile with:
$ g++ -g main.cpp list.cpp -o list
```

We will briefly explain the said three workarounds from most preferred

method to least preferred.

1. **class.cpp to classImpl.h:** Code your class the way you used to. Then at the end, rename your code file into a header file. For example from `classname.cpp` to `classnameImpl.h`. Then `#include` your new implementation header file at the end of the original header file. Remove the old `#include "classname.h"` from the beginning of your implementation header. Since the implementation file is no longer a `.cpp` file you should not compile it anymore. The following example demonstrates all the necessary changes.

```
// list.h
template <class T>
class List {
    void foo();
};
#include "listImpl.h"

// listImpl.h
template <class T>
void List<T>::foo() { ... }

// main.cpp
#include "list.h"
int main() {
    List<int> i;
    return 0;
}
compile with:
$ g++ -g main.cpp -o list
```

2. **Pre-define instances:** Code your class the way you used to. At the end, pre-define all the instances you want at the bottom of your `classname.cpp` file. The downside is that you usually don't know in advance which instances of your template class you may need. The following example demonstrates all the necessary changes.

```
// list.h
template <class T>
class List {
    void foo();
};

// list.cpp
#include "list.h"
template <class T>
void List<T>::foo() { ... }
template class List<int>;
```

```

// main.cpp
#include "list.h"
int main() {
    List<int> i;
    return 0;
}
compile with:
$ g++ -g main.cpp list.cpp -o list

```

3. **Move definitions into class:** Code all your functions right inside the class definition and not in a separate code file. By doing so you can avoid some link errors (including those caused by template classes) but you're also violating every rule in the book about good coding practices! So only use this in emergency situations. The following example demonstrates all the necessary changes.

```

// list.h
template <class T>
class List {
    void foo() { ... }
};

// main.cpp
#include "list.h"
int main() {
    List<int> i;
    return 0;
}
compile with:
$ g++ -g main.cpp -o list

```

- Just because you have a link error in your template class, it does not mean you have a template class link error. It still could be one of the usual link problems.
- If you're using an external library, you need to tell `g++` to link in with the external binary.
- Don't let a miss-configured IDE or a malfunctioning Makefile trick you. If the errors do not make sense try a simple `g++ *.cpp -g -o OutputName`. A clean build might help too.

As mentioned before, this document is not listing all the link errors and certainly it's not providing a packaged solution for any of them. We believe the discussed guidelines can empower a self-motivated C++ beginner with the right tools to fix any compiler error. As you get more advanced with C++, the ratio of time spent on fixing compiler errors as oppose to writing new code will drop

significantly. You will also realize link errors are the easiest to fix/avoid despite the cryptic looking messages.

3.3 Run-Time Error

After fixing all the build issues, you can now run the program. Any error that happens during the execution of the program is a runtime error. It usually leads to a crash (unexpected end of program) or unresponsiveness. Some of usual runtime errors are:

- **Memory access violation** [seg fault]: Happens when the program tries to access memory that does not own. Usually happens if you don't allocate memory (pointer), try to access freed memory, or go beyond your array bounds.
- **Memory corruption** [the nightmare]: If your program successfully accesses (read or write) part of memory that it should not, it will cause memory corruption. Unfortunately this does not immediately crash the program. Sometimes it feels like the bug is moving to different places. Also sometimes `gdb` prints out question marks (??) instead of the trace.
- **Infinite loop** [where FB resides]: Happens when your algorithm does not deterministically terminate in a finite time. The easy cases are when your for-loop needs a quick fix. It can also be a recursive function with a faulty base case or a buggy iterator class. Usually the program runs out of memory (heap or stack) and crashes with a different error message.
- **Stack overflow** [yes it's an error]: It's when the program is out of stack memory. A similar case is when the computer is out of heap memory. These kinds of errors are not bad by themselves. However, given that today's computers have so much memory, running out of memory is a sign of a problem.
- **Unhandled exception** [don't (...)]: It's when the program crashes and throws an unhandled exception back to the OS. Your OS may print the exception error message. But `gdb` definitely shows all the information attached to the exception. Don't try to catch the exception if you don't know how to handle it or recover from it. Some exceptions are meant to be thrown back to user.
- **Deadlock** [hardly]: Unless you're writing system code or a multi-threaded app, then errors like deadlock, race condition, and thread interruptions are unlikely to happen.

In section 3.1 we only focused on compiler **errors** and not the warnings. In order to see all the compiler warnings, compile your code with `-Wall` switch:

```
g++ *.cpp -g -Wall -o OutputName
```

It's unlikely but sometimes warnings can become future crashes! It's not a bad idea to look over them and understand the warnings before jumping into debugging.

3.3.1 Basic Debugging

Certainly you can re-use most of what you've learned with compiler errors here too. Make a guess about problematic section of the code, then try to narrow it down by `cout`-ing the program state (variables) at different places. Also learn how to use `assert` in C++. Comes handy. Most memory access problems can be isolated with a simple `assert`. If you're dealing with other types of runtime error or if the program is too long to debug with `cout` and `assert`, then it's probably a good time to learn `gdb`. With `gdb` you can isolate the problematic code section very quickly (pillar 2).

3.3.2 Advanced Debugging

Thanks to advancements in Operating Systems and new debugging tools, it's fairly easy to isolate the faulty code (pillar 2). However you should lower your expectation from the debugging tool/IDE to help you fix the problem beyond isolation⁴.

GDB Guide

Some of most common `gdb` commands are reviewed here. This is just a scratch on surface. You should put some time to master either `gdb` or your IDE debugger before diving into serious C++ programming.

Step 1: Compile your code with `-g` switch. For example: `g++ *.cpp -g -o OutputName`

Step 2: Instead of directly running your program (`./OutputName`) pass it as an argument to the debugger: `gdb OutputName`. This will put you in the debugger shell. It's like an interactive terminal.

Step 3: If you want to set a breakpoint type the command `break ClassName::FunctionName`. The debugger will pause the execution of the program at the location of breakpoints. When the program is paused you can inspect the stack, check the value of variables, and a lot more. For example if you're getting a seg fault on a private pointer inside a class, a good starting point is to set a breakpoint on the class constructor(s) to check if everything is being initialized correctly.

⁴There are other tools that can monitor programs for potential crashes and generate helpful reports. You will learn about them during the semester. If you cannot wait, lookup *valgrind*.

Step 4: To run the program type the command **r**

Step 5: The following commands are useful when the program is in paused state (breakpoint):

print VariableName: Prints the value of the variable. You have to be in the right scope.

n: Runs the next line of code and then pause again. It's useful to advance the execution by one line and see the changes.

s: Similar to **n**, runs the next line of code. If the code line is a function call, it will step into the function instead of stepping over it.

c: Continues the execution of the program. It basically takes the program out of the paused state until the next breakpoint.

bt: Prints the stack. This is very useful because it shows all the function calls in the current state of the program in the order that they have called each other. Each line shows you a frame number and the associated function name. When a function calls another one, this creates a new frame on the stack.

frame n: Moves you to the n^{th} frame in the current paused state. By moving on the stack from one frame to the other, you can examine the local variables in each function.

q: Quits the debugger shell.

help: This one is self-explanatory.

print VariableName=value: Changes the value of the variable.

Step 6: If debugger detects that the program is crashing, it will pause the execution automatically. In this case, go back to previous step and examine the program state in order to isolate the bug (pillar 2).

Step 7: Write a small testing function which, if called, can deterministically reproduce the bug. This is called unit testing (pillar 3).

This section briefly covered basic steps to understand a run-time crash and how to isolate the bug. There are certainly much more to **gdb** that this document can cover. The interested reader is encouraged to practice debugging with simple C++ programs.

Side Note

In software development, in order to improve user experience, programmers usually like to reduce run-time problems into compile-time problems. To do so, they use **unit testing** and **code reviewing** (by software or other programmers). Unit testing is when you write code to test another program. Whenever a program crashes, we find the cause of the crash. Then we write a function that can deterministically reproduce the bug. For example if you're coding a sort function, a testing function would be to check the output of the sort function to make sure the same numbers are outputted in increasing order. In future builds,

we run all previous testing functions (units) to check the program against all previously known bugs. Code reviewing is when a different programmer reviews your code to find potential bugs. There are intelligent software packages too that can detect common programming pitfalls from source code.

3.4 Logic Error

Going back to the definition of a logic error, a program has a logic problem if it does not crash but returns wrong output. There is a gray area between run-time and logic errors. The question is do you consider a wrong output any different than a crash? Does throwing an exception a bad output or a crash? What if the program is returning a wrong output because of a memory corruption? These kind of questions are language independent and it's more about algorithm design than programming.

You may think that a program that has a logic error is better than a program that crashes. However, looking from user's perspective, at least with a program that crashes you know the output, if produced, is wrong. In case of a logic error, the user (and probably the programmer) may not even know that a logic bug exist and may trust the correctness of the output. In a sense you don't even know if there is a bug until you check all the outputs. So even pillar 1 is not apparent. After you checked your program against the basic testing units, your bug is hopefully reduced to a run-time or compile time error which is what we know how to deal with.

3.5 References

- Common C++ Compiler Errors
- Compiler, Linker and Run-Time Errors
- Dealing with Compiler Errors
- GDB Tutorial

Chapter 4

Collected Questions and Answers

This chapter contains some useful questions and answers collected from students previously taking the course.

4.1 Templates

4.1.1 Linker Error

Question

I'm getting a linker error (undefined reference to XXX) when compiling some template code. Why is this?

Answer

First off I want to say that you **never** include source files under normal circumstances. You only include header files.

Let's first discuss why you got your original error: Let's pretend you have three files: main.cpp, list.h, and list.cpp. Let's pretend that list.h declares a template linked list class and list.cpp provides the implementation. In main.cpp, you instantiate an instance of your list, let's say of strings. So you have:

```
// main.cpp:  
#include <list.h>  
#include <string>
```

```
int main() {
    List<std::string> aStringList;
    aStringList.push_back( "Hello, World!" );
    return 0;
}
```

You then tell the compiler to compile this with something like: `g++ main.cpp list.cpp` and it yells at you with some linker error like the one originally posted.

Why does this happen?

Each source file the compiler knows about is compiled in its own translation unit and it only knows about its own contents and the contents that other source files export. Source files export all globally visible variables and functions. Normally when you write some non templated code, it gets fully compiled since all the types are known and it can be exported to other translation units (other source files). When you have template code, that template is just a blueprint of what the compiler has to do when it gets a type. It is not until you instantiate your template by saying "I want a List of strings" (List) that the compiler goes ahead and tries to actually generate the code where it replaces with `T = std::string`.

The problem here is that your `main.cpp` can only see things that were exported or that are visible from header files it includes. `List.cpp` doesn't know that you want a list of strings, so it never generated code for List. You are now asking for this code and the compiler doesn't know how to generate it since it was never exported and it can't see into the other translation unit to instantiate the code. This is why you generally put template code in a header file so you can include it elsewhere and it can be seen.

The other option, as Kaveh pointed out, is that you can keep things in a source file and then tell the compiler to explicitly instantiate certain versions of your class, e.g. with a string. The downside to this is that if you want to use a type that you didn't explicitly instantiate, you are back to the problem if the linker not knowing where the code is because the compiler can't generate it.

4.2 Exceptions

4.2.1 Throwing a Runtime Exception

Question

I was wondering how runtime exceptions are implemented into code because the textbook specifies that runtime exceptions should not be thrown because the client cannot handle it.

Answer

Exceptions are a way of signaling that a non-recoverable error has occurred in some segment of code. Without exceptions this is done by having functions return error codes and then always checking these return codes against various known errors. This C style means that people have to always check the return codes, which is annoying, and structure their code such that it properly cleans up depending on where the error occurs. There's also no way to enforce that users of your code actually check for errors.

In C++, we can use exceptions. Exceptions can be any type, including primitives (int, char, double, etc) or classes/structs. You say an exception has occurred by using throw:

```
// in some function
if( index < 0 || index >= _size )
    throw std::out_of_range( "Index was out of range" );
// the rest of my function
```

A throw statement immediately exits the current function (to be more precise it immediately jumps to the first catch statement that matches it), similar to a return statement. It means that no lines of code after the throw are executed in the current function. So if you have something like:

```
int * data = new int[1000];
throw std::exception();
delete[] data;
```

data will never be deleted, because that line of code cannot be reached.

Exceptions are used to signal things which the current context cannot recover from - imagine the situation above where you are asking for a certain index in your container and it is out of bounds. There is nothing your container can do about this, the data doesn't exist and it can't do anything to make it exist, so it throws an exception.

So if we can't recover from it, we throw the exception with the hope that someone else can detect this error and do something about it. This is what try/catch statements are for. A try block is put around some code that could throw an exception that you can actually do something about by catching that exception in the catch block.

```
void printList()
{
    size_t i = 0;
    try
    {
        while( true )
        {
```

```

        std::cout << myList.get( i++ ) << std::endl;
    }
}
catch( std::out_of_range const & e )
{
    // do nothing with the error other than catch it
    // since we have nothing to print out here
}
}

```

You should only use try/catch in places where you can detect and recover from an error. If you can't recover from it, you just let the exception keep unwinding the stack and breaking out of functions until it reaches main. When an exception breaks out of main, it will crash your program by default, which is perfectly reasonable behavior if an error happens that nothing in your code can handle.

This section is a high level overview of some concepts that will ultimately help you better understand and fix common mistakes novice C++ programmers make. You will still make mistakes, but at least you'll be able to understand what they mean.

4.2.2 Exceptions in While Loops

Question

If I throw an exception in my while loop, would it automatically break out of the while loop if my catch is implemented after my loop?

Answer

When you throw an exception, it will unwind the stack until it hits a try/catch block. If it never hits a try/catch block, it will break out of main and hit the default exception handler, whose behavior is to immediately terminate your program. When I say unwind the stack, I mean the memory stack that local variables and parameters use.

In your case, if you had a loop inside of a try/catch block, then the exception would back up until the opening part of the try, and then go down to the catch block. Any variables allocated before the exception is thrown that were on the stack will be properly discarded - anything you put on the heap and didn't get to delete will be leaked.

Think of throwing exceptions kind of like having a return statement, except that instead of going back to where you called from, they go to the closest catch that they will fit into.

4.3 References

4.3.1 Passing a Pointer by Reference into a Function

Question

How do I pass a pointer by reference into a function?

Answer

So I take it what you are doing is passing your head pointer to some function and you would like that function to modify that pointer and have that persist.

There are two ways to do this, one is more idiomatic C++ and the other is more of a C style.

Remember that when you pass something to a function, it is always passed by value unless you mark it as being passed by reference. Things that are passed by value are copied into the argument and will not propagate outside of the function.

```
void modify_ptr( int * ptr )
{
    ptr = &something;
}
```

Will not change the value of `ptr` outside of `modify_ptr`. However, we could choose to pass by reference (C++ style) if we want to change this:

```
void modify_ptr( int * & ptr )
{
    ptr = &something;
}
```

This will make the change to `ptr` actually change the value used when calling the function.

You could also do this in a more C style by passing a pointer to your pointer:

```
void modify_ptr( int ** ptr )
{
    *ptr = &something;
}
int * my_ptr;
modify_ptr( &my_ptr );
```

4.3.2 Member Variable is Private

Question

Since my `listElement`'s next pointer is a private variable, how do I access it in my `LinkedList`'s `insert` and `remove` functions? I tried using a `getNext()` function in `listElement`, but that didn't seem to work. The error is that "lvalue required as left operand of assignment"

```
template <class T>
listElement <T> * listElement<T>::getNext()
{
    return next;
}
template <class T>
void LinkedList<T>::insert(int position, const T & data)
{
    // some code
    listElement <T> *newPtr = new listElement <T>;
    // more code
    newPtr->getNext() = tail;
    // more code
}
```

Answer

Let's first discuss the error you were getting: "lvalue required as left operand of assignment." This is saying that you need a left hand value on the left hand side of an assignment - helpful right? What this means in this context is that you were doing something like:

```
someListElementPtr->getNext() = someOtherListElementPtr;
```

This doesn't work because the left hand side of this assignment is actually a temporary value with no name - your function returns a pointer, which you don't assign into any variable, and then you try to assign something to that variable. What happens immediately after this assignment? You just assigned to something that is in the process of evaporating into thin air. What you probably wanted to do was to actually change the next pointer. To do this you need to either return a pointer to the thing you want to change, or more easily, return a reference:

```
listElement <T> * & listElement<T>::getNext();
```

This now says: return a reference to a `listElement`;T; pointer. References refer to things with actual names, which means they can be assigned to. The whole

lvalue/rvalue is a bit of an advanced topic, but this should help you understand why you got that error.

Now on to actually coming up with a good solution to this: one way is to do what I just mentioned - return a reference to your pointer instead of the pointer itself. If you do this, any changes you make to the value returned will change the pointer in your object.

Another way to solve this is to use a friend declaration. Friend declarations are a way of letting C++ know that some classes/functions are allowed to ignore your careful public/protected/private access control, and see everything as public. A friend declaration would go in the class you want to open up to another class, something like:

```
template <class T>
class ListElement
{
    public:
        template <class U> friend class LinkedList;
        // the rest of your class
};
```

What this says is that ListElement will be friends with some class called LinkedList that has a template parameter (we use a different name (not T) in case the template argument differs), and that friend class gets to see any member functions or variables as public.

4.4 Virtual Functions

4.4.1 Constructors and Virtual

Question

I know what virtual function is, but I am still not exactly sure when to use it. For example, in "shape.h", why destructor is virtual function but constructor is not?

Answer

Constructors can never be virtual. Every class is responsible to construct itself and if necessary call the base class constructor. However, it's a good idea to make destructors virtual to make sure the derived class destructor gets called. See this post for some more information¹.

¹<http://stackoverflow.com/questions/461203/when-to-use-virtual-destructors>

Constructors have no need of being virtual because when you construct an object, you need to know its type explicitly. Consider an example of a hierarchy with a Shape that has a Rectangle child:

```
Shape * aShape = new Rectangle();
```

In this situation we know we are constructing a Rectangle and there is no ambiguity about what type we have. We explicitly construct the Rectangle and there is no need at runtime to figure out what we want to construct.

Remember that with inheritance, a Rectangle IS A Shape, so we can have a Rectangle pointer that assigns to a Shape pointer. However, let's look at what happens when we delete this:

```
delete aShape;
```

aShape has type Shape pointer, but if we were to just call Shape's destructor that wouldn't be enough, because aShape actually points to a Rectangle object. So we need virtual destructors here so that dynamic binding can step in and at run time tell us where we need to go (which is to Rectangle's destructor). At runtime C++ will check what kind of Shape aShape points to, and then call the appropriate function (destructor in this case).

Note that none of this is an issue if we aren't using pointers to access our objects (references count here too, since behind the scenes they are just pretty pointer dereference/address of operations):

```
Rectangle aRectangle;
Shape & someRef = aRectangle;
someRef.draw(); // calls Rectangle's draw,
// not Shape's (assuming draw was virtual)
```

In summary the whole point of declaring a function virtual is so that you can have pointers to related (in class hierarchy terms) types and still end up calling the correct function.

4.5 Iterators

4.5.1 General Iterator Confusion

Question

Though its pretty late for this assignment I would still like to know this for the future assignments. As I was looking through the posts about iterators I was a little confused about one thing. Iterators seem great for taking internal data and showing it to the outside world. Suppose we want to get data from the outside and put it in some objects, what role do iterators play there, especially

if its complicated objects where we are iterratorating through users and we want to change specific parts within the User wall or bag of friends?

Answer

Iterators can be used for the purpose you mention in some data structures. Normally a class will have two types of iterators - a const version and a non const version. With the non-const version, you can use it to modify data you are iterating over. This doesn't solve the question of how you can insert or remove elements yet.

Many classes will implement insertion or removal functions that take an iterator as a parameter. Remember that the iterator refers to a specific element in your data structure, so it can be used to delete a specific element if written properly. In addition, it can be thought of as giving an index for insertion, and normally insertion functions that take an iterator will insert immediately before or after that item. For some data structures these operations don't make much sense. Consider a Heap, which enforces a special ordering of the elements since it must maintain the heap property, so inserting anywhere wouldn't be supported.

You can see examples of this on `std::vector`, check out the following two² references³.

For the assignment and wanting to change specific parts of a User wall or something, remember that your iterator should return a reference to the type of items it is iterating over. Once you have an item, you can then call functions on it like you normally would. Consider:

```
// Iterate over all wall posts for every user and change the
// author to something
UserList users;
for( UserList::Iterator it = users.begin(); it != users.end(); ++it )
{
    // the type of *it is User. Let's pretend we have a function
    // getWall() that returns a reference to the wall
    for( User::Iterator postIt = it->getWall().begin();
        postIt != it->getWall().end(); ++postIt )
    {
        // the type of *postIt is WallPost
        postIt->setAuthor( "Jon Snow" );
    }
}
```

Note that in my example above, I have overloaded the `->` operator so that I don't have to do the ugly looking `(*iterator).functionCall()`.

²<http://en.cppreference.com/w/cpp/container/vector/insert>

³<http://en.cppreference.com/w/cpp/container/vector/erase>

4.6 Inheritance

4.6.1 Initialization Lists

Question

I'm having trouble with the constructor for a class that inherits another class. What is wrong?

Answer

Use initialization lists! There's a good reason I've been repeating this for a while at lab, and this is precisely one of the cases where normal assignment won't work in a constructor.

When your constructor runs, by the time it reaches the opening curly brace, it has already allocated and initialized all of your member variables (plus allocated space for itself). Remember that when a class is constructed, it must first construct its parent before constructing itself. So:

```
struct Derived : public Parent
{
    int x;
    double y;
};
```

```
Derived::Derived()
{
    // this will first construct the Parent, then us,
    // meaning allocating space for our int x and double y
}
```

if you want to explicitly invoke a constructor other than the default constructor for your parent class, you need to use initialization list syntax:

```
Derived::Derived() : Parent( arguments, go, here ),
    x( some_initial_value ),
    y( another_initial_value )
{
    // everything in the initialization list has been
    // allocated and initialized before we reach this line
}
```

4.6.2 Is-a string

Question

I was having trouble with correct syntax while implementing my class that inherits from `std::string` like in the homework description, so I looked it up and saw an overwhelming number of posts saying to never inherit from `string` (and therefore no help on actually how to do it). If we choose to use has-a instead will we be counted off?

Answer

One reason it isn't done is that it just isn't the style (not idiomatic C++) and a lot of writing C++ revolves around writing code that fits in with a currently accepted style of "good code."

Some good reasons are detailed here covering both the is-a and as-a cases. The bulk of the problems arise from the fact that essentially none of the standard types are designed for polymorphism, so none of the functions are virtual. When you inherit from something, your new type is convertible to the type you inherited from, meaning it can be passed to functions expecting the base type. Since none of your functions are virtual, you can't actually override anything in this situation. Additionally, there are issues with various constructors and copy constructors that won't get properly called when converting between the derived and base types.

In idiomatic C++, the most commonly used style is "has-a."

The way hash functions are done in the standard library is with template specialization. This means that there is a class called `template <class T> std::hash` that is never defined generically, only in terms of specific types T. Here's an example of that:

```
template <class T>
struct MyHasher; // We'll never define this in terms of just T

template <> // no unknown template parameters
struct MyHasher<std::string>
// this is called a template specialization where T = std::string
{
    MyHasher()
    {
        // any code needed to set up any state associated with the hash
    }

    size_t operator()( std::string const & stringToHash ) const
    {
```

```

        // some math magic to return a number. note that this number
        // is independent of your table size. you'd have to factor
        // that part in your data structure and not in this function.
    }
};

```

You would use this like the following:

```

template <class Key, class Value>
void MyHashTable<Key, Value>::insert(
    Key const & k, Value const & v ) // or wrapped in some node
{
    // pretend we have some member variable of
    // type MyHasher<Key> named _hashFunction;
    size_t hash = _hashFunction( k ) % _size; // or something
    // whatever
}

```

4.6.3 Protected Variables

Question

Why can't I access protected variables in my superclass from a subclass? Do I need to include something after I inherit from a parent?

Answer

You are probably not declaring your inheritance to be public. Remember that classes default to private, and structs to public. So when you do:

```

class MyCoolClass
{
    protected:
        int x;
};

// note the lack of any public/private/protected keyword
class SubClass : MyCoolClass
{
    SubClass()
    {
        x = 3; // ERROR: x is private in this context
    }
};

```


You can't get access to `x`, even though it is protected in `MyCoolClass`. This is because by default, if you don't specify an access level for the inheritance of a class, it will default to private, making everything and anything in the parent class private to the derived class. You need to declare the inheritance as either public or protected to get access to protected members.

First things first: you cannot inherit from `ArrayList` because `ArrayList` is not a type. Since you templated `ArrayList`, it must always be accompanied by its template parameter to fully define the type, so you should be inheriting from `ArrayList<T>`. Secondly, when you use templates in inheritance, or have multiple inheritance, you need to help the compiler figure out what you mean when you say you want a variable that comes from a parent class. You can do this in one of three ways:

Let's assume our base class is something like:

```
template <class T> class Parent { protected: int x; };
```

The first way to access `x` is to use the `this` pointer to disambiguate the source of `x`:

```
template <class T>
class Child: public Parent<T>
{
    public:
        Child()
        {
            this->x = 3; // Explicitly state I am using a copy of x
                       // that can be reached from myself
        }
};
```

The second way is to explicitly say where `x` is coming from:

```
template <class T>
class Child: public Parent<T>
{
    public:
        Child()
        {
            Parent<T>::x = 3; // I'm changing the x I got from Parent<T>
        }
};
```

The last way is to say that you will be using a parent's variable (or function) in your subclass:

```
template <class T>
class Child: public Parent<T>
{
```

```

protected:
    // I'll use Parent<T>'s x as a protected variable here
    using Parent<T>::x;
public:
    Child() { x = 3; }
};

```

Note that with the `using` syntax, you can actually change the access level from protected back to public (but never from private to something else, because you can never see private outside of the class it is defined in). All of this extra verbose syntax only applies if you use multiple inheritance or templated inheritance.

4.7 Const

4.7.1 error: passing 'const Wall' as 'this' argument of 'WallPost Wall::get(int)' discards qualifiers

Question

I seem to run into this error quite a bit, and don't completely understand what's going on. In general ,what is going wrong here?

Answer

You are getting this error because you are trying to access a const method from an instance of a class that is itself not const. Something is considered const if you either declare it as such or pass it by const reference to a function. For example:

```

struct myStruct
{
    void nonConstFunction();
    void someFunction() const;
};

void doSomething( myStruct const & obj )
{
    obj.someFunction(); // ok since someFunction is marked as const
    obj.nonConstFunction(); // NOT ok, discards const qualifiers
}

```

In general just remember that const objects can only call const methods.