C++ Syntax Notes: Making Comments in C++ Code

When you look through C++ code, you'll see there are comments written throughout the code that will usually be displayed in a grayed-out font by your code editor.

C++ comments come in two varieties. There are single-line comments, which begin with // and continue until the end of the line:

```
int x = 2; // A single-line comment can appear at the end of a line.
// Single-line comments can also appear by themselves on a line.
```

There are also multi-line comments, which appear in between these open and close markers: /* */

```
int x = 2;

/* This is a multi-line comment.
  It will continue for as many lines as necessary
  until it's closed with the closing marker. */

int y = 3;
```

Be sure to leave notes to yourself in your code where appropriate. That will help you review your own code later, and it's very helpful to other people who need to use your code!

Headers and Source Files: C++ Code Organization

This reading explains a few details to keep in mind about how C++ programs are structured in code. Because a C++ program's source code exists across several separate files, you need to know how the compiler will pull those pieces together.

If you look at the example source code provided so far, you'll see that C++ code files often come with two different file extensions:

.**h** files are "header files". These usually have definitions of objects and declarations of global functions. Recently, some people name header files with a "**.hpp**" suffix instead.

.**cpp** files are often called the "implementation files," or simply the "source files". This is where most function definitions and main program logic go.

In general, the header files contain *declarations* (where classes and custom types are listed by name and type, and function prototypes give functions a type signature) while the source files contain *implementation* (where function bodies are actually defined, and some constant values are initialized). It becomes easier to understand this organizational separation if you know more about how the code is compiled into a program you can run. Let's look at the cpp-class example from the provided code.

The Cube.h header file has this content:

```
/**
* Simple C++ class for representing a Cube.
*
* @author
*   Wade Fagen-Ulmschneider <waf@illinois.edu>
*/

// All header (.h) files start with "#pragma once":
#pragma once
```

```
// A class is defined with the `class` keyword, the name
// of the class, curly braces, and a required semicolon
// at the end:
class Cube {
 public:  // Public members:
   double getVolume();
   double getSurfaceArea();
   void setLength(double length);

 private: // Private members:
   double length_;
};
```

Note the **#pragma once** at the beginning. Instructions beginning with **#** are special commands for the compiler, called preprocessor directives. This instruction prevents the header file from being automatically included multiple times in a complex project, which would cause errors.

This header file also includes the declaration of the Cube class, including listing its members, but the definition doesn't give the full source code of the functions here. For example, the body of the setLength function is not written here! Only the signature of the function is given, to declare its input argument types and return type. Instead, the function will be defined in the Cube.cpp source code file.

Here is the Cube.cpp source code file:

```
/**
* Simple C++ class for representing a Cube.
*
* @author
*   Wade Fagen-Ulmschneider <waf@illinois.edu>
*/

#include "Cube.h"

double Cube::getVolume() {
   return length_ * length_ * length_;
}

double Cube::getSurfaceArea() {
```

```
   return 6 * length_ * length_;
}

void Cube::setLength(double length) {
   length_ = length;
}
```

Notice that the first thing it does is **#include "Cube.h"** to include all the text from the Cube.h file in the same directory. Because the filename is specified in quotes, as "Cube.h", the compiler expects it to be in the same directory as the current file, Cube.cpp. Below we'll see a different way to refer to filenames using < > instead.

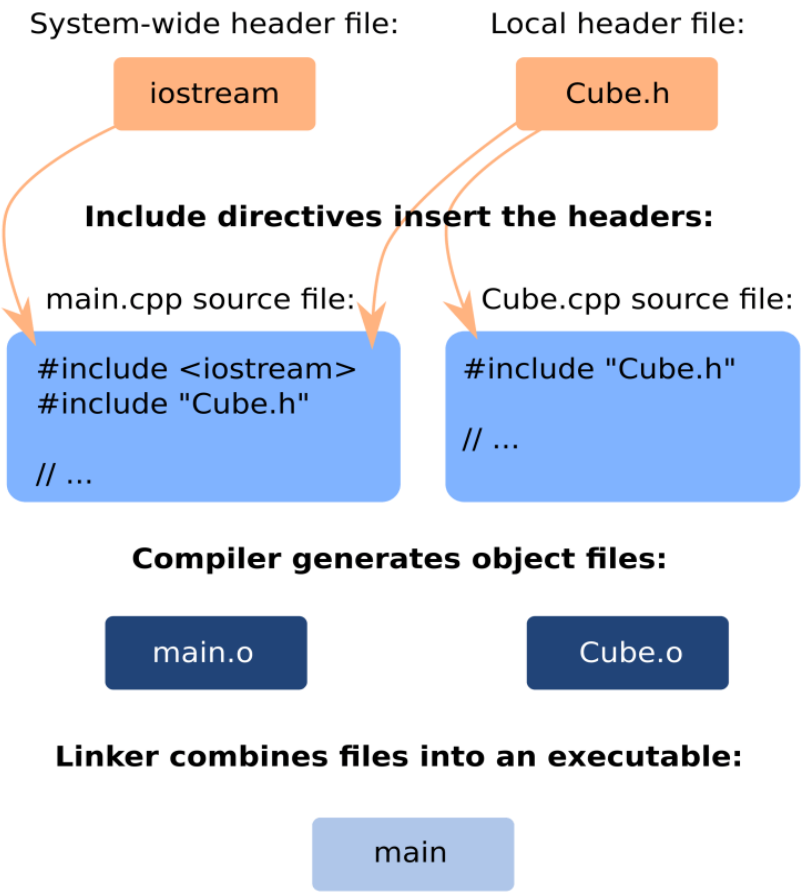
Then, the file gives the definitions for the functions from the class definition before. These full source code listings for the functions that were declared previously are called the implementation. (It is common that function bodies will be implemented in the cpp file and *not* in the h file, but sometimes *short* class function bodies will be defined directly in the h file where they are declared inside the class declaration. The compiler handles that situation in a special way automatically so that it doesn't cause problems in the linking stage, which is described below.)

Finally, let's look at main.cpp:

```
/**
* C++ code for creating a Cube of length 2.4 units.
* - See ../cpp-std/main.cpp for a similar program with print statements.
*
* @author
*   Wade Fagen-Ulmschneider <waf@illinois.edu>
*/

#include <iostream>
#include "Cube.h"

int main() {
 Cube c;

 c.setLength(3.48);
 double volume = c.getVolume();
 std::cout << "Volume: " << volume << std::endl;

 return 0;
}
```

This time there are two #include directives! First, it includes a standard library header from the system directory. This is shown by the use of < >. When you write **#include <iostream>**, the compiler will look for the iostream header file in a system-wide library path that is located outside of your current directory.

Next, it does **#include "Cube.h"** just like in the Cube.cpp file. You have to include the necessary headers in every cpp file where they are needed. However, you shouldn't use #include to literally include one cpp file in another! **There is no need to write #include "Cube.cpp"** because the function definitions in the Cube.cpp file will be compiled separately and then *linked* to the code from the main.cpp file. You don't need to know all the details of how this works, but let's look at a diagram that shows the general idea:



The Cube.cpp files and main.cpp files make requests to include various header files. (The compiler might automatically skip some requests because of **#pragma once** to avoid including multiple times

in the same file.) The contents of the requested header files will be temporarily copied into the cpp source code file where they are included. Then, the cpp file with all of its extra included content will be compiled into something called an **object file**. (Our provided examples keep the object files hidden in a subdirectory, so you don't need to bother with them. But, if you see a file that has a **.o** extension, that is an object file.) Each cpp file is separately compiled into an object file. So, in this case Cube.cpp will be compiled into Cube.o, and main.cpp will be compiled into main.o.

Although each cpp file needs the appropriate headers included for compilation, that has to do with checking type information and declarations. The compiled object files are allowed to rely on *definitions* that appear in the other object files. That's why it's okay that main.cpp doesn't have the Cube function definitions included: as long as main.cpp does know about the type information from the Cube function signatures in Cube.h, the main.o file will be "**linked against"** the compiled definitions in the Cube.o file. (I don't know why we say "link against" instead of "link with" when we talk about this, but that is the usual terminology!) The linker program will also link against system-wide object files, such as for **iostream**. After the compiler and linker programs finish processing your code, you will get an executable file as a result. In this case, that file is simply named **main**.

Fortunately, you won't have to configure the compiler manually in this course. We will provide a **Makefile** to you for each project, which is a kind of script that tells the compiler how to build your program. We'll talk more about that in the next reading lesson.

C++ Syntax Notes: Basic Operators, If-Else, and Type Casting

Here are some basic syntax notes about C++.

Assignment vs. Equality

Notice that "=" is the assignment operator in C++ and "==" is the comparison operator that checks equality:

```
#include <iostream>

int main() {
 int x = 2; // This line assigns a value.
 int y = 2;
  if (x == y) {
   // Note that we had to use "==" to check equality!
   // That's two equal signs!
   std::cout << "x and y are equal" << std::endl;
 }
 else {
   std::cout << "x and y are not equal" << std::endl;
 }
  return 0;
}
```

Be very careful not to mix these up or make a typo! That is a common mistake that even long-time C++ programmers make. It can cause bugs that are hard to find.

If-Else

The above example also shows one of the most fundamental control structures in C++: The **if** and **else** keywords. The basic structure is always like this:

```
if (condition) {
   // true case
}
else {
   // false case
}
```

You can also chain these together with more if-else combinations. In this situation, the first condition that is true will be the section that is executed:

```
if (condition1) {

}
else if (condition2) {

}
else if (condition3) {

}
else {
   // If none of the other conditions were met...
}
```

It's crucial to note that in the above example, the use of "if... else if... else", in a chain, makes the conditional cases mutually exclusive. Only one of them will execute: the first one that is true. Be very careful to use "else" where appropriate to get this behavior. If you simply put several "if" statements in a sequence, then potentially, several of them will execute, as in this example:

```
int x = 100;
if (x > 0) {
   // This will execute
}
if (x > 10) {
   // This will execute
}
if (x > 50) {
   // This will execute
}
else {
   // This is the only section that won't execute!
}
```

Here, all of the conditional blocks will execute except for the last "else" part: the last "else" at the end only affects the preceding "if," and since it's true that x is greater than 50, that "if" block will execute and the final "else" will not. The important lesson is to always be intentional about where you choose to write "else" (or not), depending on whether you intend cases to be mutually exclusive.

You should also be aware of the strange-looking "ternary operator," also called the "conditional operator," which is actually a combination of two operators, "?" and ":" (question mark and colon). The basic format for this syntax is:

[Boolean-valued condition] ? [expression to evaluate if true] : [expression to evaluate if false]

So, you could compare "A ? B : C" to "if (A) {B;} else {C;}" but that the ternary operator is allowed in situations where "if" is not allowed. This is because "?:" is evaluated at the level of an expression, so it can be a sub-expression within a larger expression, whereas the "if" statement is a top-level flow control statement that can't be nested within an expression. This means that you can put ?: to the right of an assignment = operator, which you cannot do with an "if". Here is an example:

```
// Since (5<10) is true, the expression before the colon will be selected,
which is 1.
int x = 5 < 10 ? 1 : 2;
// Now x is equal to 1.

// Note, the following syntax is NOT allowed in C++, which is why the ternary
operator can be useful in these cases:
int y = if (5<10) {1;} else {2;}
```

This ?: ternary operator is useful in other situations where you can use it creatively to make code more concise and compact, although some people find it confusing to read. We recommend that you do not use it more than necessary, but you will see it occasionally as you explore C++ source code.

Comparison Operators and Arithmetic Operators

There are many other operators in C++ that are compatible with various types. You should get familiar with some of them and be able to tell the difference between a comparison and an assignment.

```
#include <iostream>

int main() {
 if (3 <= 4) {
   std::cout << "3 is less than or equal to 4" << std::endl;
 }
  int x = 3;
 x += 2; // This increases x by 2. It's the same as writing: x = x + 2;
 std::cout << "x should be 5 now: " << x << std::endl;
  return 0;
}
```

You can read about many more operators here: http://www.cplusplus.com/doc/tutorial/operators/

Type Casting

An important concept to understand in C++ is "casting." This is the phenomenon where a value is automatically converted to a different type in order to allow an operation to continue. For example, if you try to add together a floating point **double** value and an integer **int** value, the int will be converted to double first, and then the result of the addition will have type double as well. But if you save a double value to an int variable, the floating point value will be truncated to an integer! For example:

```
#include <iostream>
int main() {
   int x = 2;
   double y = 3.5;
   std::cout << "This result will be a double with value 5.5: " << (x+y) << std::endl;
   int z = x+y; // This expression is calculated as a double, but then it's cast back to int!
   std::cout << "This result will be an int with value 5: " << z << std::endl;
   return 0;
}
```

It's also the case that various values can be cast to the Boolean **bool** type. So, they can be used as a condition. For example, usually, nonzero numeric values will be considered **true**, and zero will be considered **false**.

```cpp
#include <iostream>
int main() {
 if (0) {
   std::cout << "You won't see this text." << std::endl;
 }
 if (-1) {
   std::cout << "You WILL see this text!" << std::endl;
 }
 return 0;
}
```

You need to be aware that casts are happening invisibly in your code! Sometimes this can cause hard-to-find bugs. If you want to read more details about the casting operation, especially if you are curious about a debugging situation later, see these references:

A more detailed primer on the casting topic: http://www.cplusplus.com/doc/tutorial/typecasting/

In-depth documentation: https://en.cppreference.com/w/cpp/language/implicit_conversion

C++ Syntax Notes: Block Scope, Loops

Block Scope

After viewing the lecture about stack memory, you should already have an intuitive idea about how **block scope** works. The idea is that certain blocks of code, signified by **{ }**, create an inner stack on top of the previous stack, which can hide the pre-existing values. The lifetime of stack variables inside a block is called the variable's **scope**. Variables created on the stack inside the inner block are only in existence for the duration of the block. When the block ends, the inner stack is removed, and the pre-existing values in the outer scope are available again (because they are still in scope!).

Here is an example:

```
#include <iostream>
int main() {
  // You can actually make an inner scope block anywhere in a function.
 // Let's do it to show how scope works.
  // In the initial, outer scope:
 int x = 2;
 std::cout << "Outer scope value of x (should be 2): " << x << std::endl;
  // Create an inner scope and make a new variable with the name "x".
 // This is not an error! We can redeclare x because of the inner scope.
 // Also, make an extra variable named "y".
 {
   int x = 3;
   int y = 4;
   std::cout << "Inner scope vaue of x (should be 3): " << x << std::endl;
   std::cout << "Inner scope vaue of y (should be 4): " << y << std::endl;
 }
  // Now that the inner block has closed, the inner x and y are gone.
 // The original x variable is still on the stack, and it has its old value:
 std::cout << "Outer scope value of x (should be 2): " << x << std::endl;
  // We can't refer to y here, because it doesn't exist in this scope at all!
 // If you uncomment this line, there will be a compile error.
 // std::cout << "This line causes an error because y doesn't exist: " << y <<
std::endl;
  return 0;
}
```

Some keywords like **if** can have a block in { } afterwards, which does create an inner block scope for the duration of the conditional block:

```
#include <iostream>
int main() {
 int x = 2;
 std::cout << "Outer scope value of x (should be 2): " << x << std::endl;
  if (true) {
   int x = 3;
   std::cout << "Inner scope value of x (should be 3): " << x << std::endl;
 }
 std::cout << "Outer scope value of x (should be 2): " << x << std::endl;
 return 0;
}
```

Loops

There are several kinds of loops in C++ that allow you to process data iteratively. We'll just show you a few types here.

for loops

The **for** loop is a common loop type that lets you specify an iteration variable, a range for that variable, and an increment instruction. The syntax is:

for (declaration ; condition ; increment operation) { loop body }

Be careful about whether you are redeclaring the iteration variable at block scope or not! Here's an example:

```
#include <iostream>
int main() {
  // outer scope version of "x" to show the for-loop block scoping
 int x = -1;
  std::cout << "[Outside the loop] x is now (should be -1): " << x << std::endl;
  // The for loop lets us declare a variable in the first part of the
 // loop statement, which will belong to the inner block scope:
 for (int x = 0; x <= 2; x++) {
   std::cout << "[Inside the loop] x is now: " << x << std::endl;
 }
  // The outer scope x is still -1
 std::cout << "[Outside the loop] x is now (should be -1): " << x << std::endl;
  // This version doesn't redeclare x, so it just inherits access to the
 // same x variable from the outer scope. This modifies the outer x directly!
 for (x = 0; x <= 2; x++) {
   std::cout << "[Inside the loop] x is now: " << x << std::endl;
 }
  // In the last iteration where the condition x<=2 was true, x had the value 2.
 // After that iteration, x was incremented one more time and became 3.
 // Then the condition was false and the loop body did not execute.
 // Afterwards, the outer scope x is still 3 because the loop modified it.
 std::cout << "[Outside the loop] x is now (should be 3): " << x << std::endl;
  return 0;
}
```

while loops

Another loop is the simple **while** loop, which has this syntax:

while (condition) { loop body }

```
#include <iostream>
int main() {
 int x = 0;
 std::cout << "This should show 0, 1, 2, 3:" << std::endl;
 while (x <= 3) {
   std::cout << "x is now: " << x << std::endl;
   x++; // increment x by 1
 }
 return 0;
}
```

Preview: Modern "range-based" for loops

You might see another very common type of **for** loop in recent C++ code, that has this syntax:

for (temporary variable declaration : container) { loop body }

We'll wait to talk more about that in a later reading lesson.

C++ Syntax Notes: Uninitialized Pointers, Segfaults, and Undefined Behavior

---

This reading discusses some details of pointers and safe memory access. We hope to help you avoid common crashes and bugs in your code.

You should refer back to the lecture on heap memory, which gave an example of an uninitialized variable producing an unpredictable result. Let's go over a few definitions and concepts first.

Segmentation fault (Segfault)

Various kinds of programming bugs related to pointers and memory can cause your program to crash. On Linux, if you dereference an address that you shouldn't, this is often called "segmentation fault," or "segfault." For example, if you dereference a pointer that is set to nullptr, it will almost certainly cause a segfault and crash immediately. This code will segfault:

```
// This code compiles successfully, runs, and CRASHES with a segfault!
int* n = nullptr;
std::cout << *n << std::endl;
```

On other operating systems, the error message may say something different. *Some* C++ compilers may respond to this in other unpredictable ways, but this is one type of error that has a fairly reliable symptom. It is actually an example of what we talk about in the next paragraph.

Undefined behavior

Sometimes it's possible to write a program that compiles, but that is not really a safe or valid program. There are some improper ways to write C++ that are not strictly forbidden by the C++ standard, but the C++ standard doesn't define how compilers are supposed to handle those situations, so your compiler may generate a program that works, or not! This is called "**undefined behavior**." Many beginning programmers make the mistake of thinking that just because their program compiles and runs for them on their system, that it must be valid and safe code. **This isn't true**. "It works for me" isn't an excuse, so be sure to proofread your code, use safe practices, and avoid relying on undefined behavior.

Many times, undefined behavior is caused by the careless use of uninitialized variables. Let's talk about initialization.

Initialization

Initialization is specifying a value for a variable from the moment it is created. Remember that if you don't initialize a pointer, you really should *not* try to dereference it. If you dereference an uninitialized pointer, even just to read from it and not to write to it, you may cause your program to crash, or something else unexpected might happen later. Here are some examples.

Examples with stack memory

This pointer "x" is uninitialized. It contains a seemingly random memory address. (However, this is also **not** a good way to get a source of randomness, even for those situations where we want random numbers!) Dereferencing this pointer would cause undefined (unpredictable) behavior:

```
// Dangerous; this can lead to careless mistakes and crashes!
int* x;
```

This pointer is explicitly initialized to nullptr. Dereferencing this pointer would cause the program to crash, immediately and predictably. (On Linux, this is often called a "segmentation fault," or "segfault.") As we'll discuss below, it's a good practice to set a pointer to nullptr if you aren't setting it to any other value immediately.

```
// Explicitly initializing a pointer to nullptr
int* y = nullptr;
```

You can also initialize a value with the () syntax following the variable name. If the type is a class, the parameters will be given to the class type's corresponding constructor. For built-in types such as int, which are not class types, there are no constructors; however, we can still specify an initialization value this way. Sometimes you'll see initialization done with the {} syntax instead of (). This is a new feature since C++11. It can be a good way to make it clear that you are performing an initialization, not some kind of function call. Later in this course sequence, you'll see some other good ways to use the {} initialization syntax.

```
// Other ways to initialize
int* y2(nullptr);
int* y3{nullptr};
```

Plain built-in types, such as int, that are not initialized will have unknown values. However, if you have a class type, its default constructor will be used to initialize the object. Here, int h has an unknown value, but supposing we have some well-defined class type Box, Box b will be given reasonable default values. (It's the responsibility of the Box class type to ensure that.)

```
// h is uninitialized!
int h;
// b will be default initialized
Box b;
```

Here we create integer "i" on the stack, safely initialized with value 7, and then we create a pointer "z" on the stack, initialized to the address of i. This way, z points to i. It's safe to dereference z.

```
int i = 7;
int* z = &i;
```

Examples with heap memory

When you want to use heap memory, you'll use the "new" operator. As with the stack memory case, you should be wary when you use "new" for a built-in type like int, since these types may not have default initialization. Therefore, you shouldn't assume they'll have an expected default value (such as 0). For those cases, be sure to initialize the value manually. Here are some examples.

The pointer "q" is created and initialized with the address for a newly-allocated integer on the heap.

However, the integer that q points to does not have a predictable value. It depends on the compiler. We shouldn't rely on this integer to have any particular value at the beginning.

```
int* q = new int;
```

You can specify initialization parameters at the end of the "new" expression. This will create pointer "r" initialized with newly-allocated memory for an integer with the value "0" explicitly.

```
int* r = new int(0);
```

There are a lot of other special situations in C++ where different factors may slightly change how an object is initialized. You don't need to get into all those details. If you're unsure, the easiest thing to do is make sure that you explicitly initialize your variables. For a more exhaustive reference on initialization, you can refer to this page: https://en.cppreference.com/w/cpp/language/initialization

Resetting deleted pointers to nullptr

Now that we've reviewed initialization, especially for pointers, let's talk about why you should manually reset pointers to nullptr when you're done with them.

Note that using "delete" on a pointer frees the heap memory allocated at that address. However, deleting the pointer does not change the pointer value itself to "nullptr" automatically; you should do that yourself after using "delete", as a safety precaution. For example:

```
// Allocate an integer on the heap:
int* x = new int;
// Now x holds some memory address to a valid integer.
// Do some kind of work with the integer.
// We'll just set that integer to 7:
*x = 7;
// Now delete the pointer to deallocate the heap memory:
delete x;
// This destroys the integer on the heap and frees the memory.
// But now x still holds the memory address!
// Set x to nullptr for safety:
x = nullptr;
```

The idea here is that by manually setting x to nullptr after "delete x", we help prevent two kinds of problems:

1. We don't want to delete the same allocated address more than once by mistake, which could cause errors. Using "delete" on nullptr does nothing, so this way, if we accidentally try to delete the same address twice, nothing further happens.

2. We must never dereference a pointer to deallocated memory. This could cause the program to crash with a segfault, exhibit strange behavior, or cause a security vulnerability. However, attempting to dereference a nullptr will almost always cause a segfault and terminate the program immediately. This is actually better than causing a silent security vulnerability or another problem that is harder to detect! Therefore, it makes sense to set the deleted pointer to nullptr, thus ensuring that if we dereference it carelessly, then it will cause a very obvious runtime error that we can fix.

You should also note that we only need to use delete and nullptr with pointers, of course. Variables in stack memory don't need to be manually deleted. Those are automatically deallocated when they go out of scope. (Please refer to the other reading lesson on scope.)

In summary, remember that if you use "new," you will also need to "delete," and after you delete, you should set to nullptr.

Growing beyond pointers

As you go further in your lessons on C++, at first you may be frustrated that the use pointers and raw memory is very tedious. However, class types can be designed handle all the new and delete operations for you, invisibly. As you create your own robust data structures, and as you use libraries such as the C++ Standard Template Library, you will find that you very rarely have to use "new" and "delete" anymore.

C++ Syntax Notes: The Modern Range-Based "for" Loop

Modern Range-Based "for" Loops

In recent versions of C++, there is a version of the **for** loop that automatically iterates over all of the things in a container. This is very useful when used with a standard library container, because you don't have to worry about trying to access memory outside of a safe range, for example: the loop will automatically begin and end in the right place.

for (temporary variable declaration : container) { loop body }

There's an important detail about the temporary variable. If you declare an ordinary temporary variable in the loop, it just gets a copy of the current loop item by value. Changes you make to that temporary copy won't affect the actual container!

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> int_list;
   int_list.push_back(1);
   int_list.push_back(2);
   int_list.push_back(3);
   for (const int& x : int_list) {
      // This version uses references, so it doesn't make any temporary copies.
      // However, they are read-only, because they are marked const!
      std::cout << "This item has value: " << x << std::endl;
      // This line would cause an error:
      //x = 99;
   }

   return 0;
}
```

Expected output:

```
This item has value: 1
This item has value: 2
This item has value: 3
If that worked correctly, you never saw 99!
```

If you make the temporary variable of a reference type, you can actually modify the current container item instead of just getting a copy. This modified example shows how:

```
#include <iostream>
#include <vector>
int main() {
  std::vector<int> int_list;
 int_list.push_back(1);
 int_list.push_back(2);
 int_list.push_back(3);
  for (const int& x : int_list) {
   // This version uses references, so it doesn't make any temporary copies.
   // However, they are read-only, because they are marked const!
   std::cout << "This item has value: " << x << std::endl;
   // This line would cause an error:
   //x = 99;
 }

 return 0;
}
```

Expected output:

```
This item has value: 99
This item has value: 99
This item has value: 99
Everything was replaced with 99!
```

There are more advanced ways to use this, too. For example, if you are iterating over large objects in a container, then even if you don't want to modify the objects, you might want to use a **reference to a constant** as the loop variable type to avoid making a temporary copy of a large object, which could otherwise be slow.

```cpp
#include <iostream>
#include <vector>
int main() {
  std::vector<int> int_list;
  int_list.push_back(1);
  int_list.push_back(2);
 int_list.push_back(3);
  for (const int& x : int_list) {
   // This version uses references, so it doesn't make any temporary copies.
   // However, they are read-only, because they are marked const!
   std::cout << "This item has value: " << x << std::endl;
   // This line would cause an error:
   //x = 99;
 }

 return 0;
}
```

Expected output:

```
This item has value: 1
This item has value: 2
This item has value: 3
```

This will probably be very useful to you in the future if you continue to use advanced C++.

What are unsigned integers?

Sometimes you'll see another integer type in C++ code, "unsigned" integer, that cannot represent negative values. Instead, unsigned integers have an increased upper positive value range compared to signed integers of the same memory usage. (Signed integers devote a bit of memory just to be able to represent a negative sign.) Unsigned integers are sometimes used in special cases to make use of memory extremely efficiently; for example, a data storage format might use them in order to be more compact. But mixing unsigned and signed integers in your code can cause unexpected problems. Let's walk through a few examples with unsigned integers.

Unsigned type syntax

The normal "int" syntax creates a signed int by default. But, you could also write "signed int" or "signed" to get the same type:

```
int a = 7;
// signed int a = 7;
// signed a = 7;
```

If you write "unsigned int" or "unsigned", you can create a variable with the unsigned integer type:

```
unsigned int b = 8;
// unsigned b = 8;
```

Issues with unsigned arithmetic

Unsigned ints can't represent negative values. Instead, they sacrifice the ability to represent negative numbers in exchange for a higher upper positive limit to the value range that they can represent, using the same number of bits as the comparable signed integer. However, the underlying bit representation that unsigned integers use for these very large values is actually the same as the representation that signed integers use for their negative value range. **This means that a negative signed int may be re-interpreted as a very large positive unsigned int, and vice versa.** This can cause strange behavior if you mix signed and unsigned ints. If you do arithmetic between signed and unsigned ints, then you can get undesired results if negative values should logically have arisen.

Issues with addition

Addition with unsigned values may be no problem, as long as you don't exceed the maximum range. This is 7+8 and shows 15 as expected

```
std::cout << (a+b) << std::endl;
```

However, you do need to be careful if you are approaching the upper limit for a signed int even if you are using unsigned ints. Signed ints have a lower maximum value, so if you get an unsigned int greater than that limit and then cast it to signed int, it will be interpreted as a negative number you did not expect.

Issues with subtraction

Now let's look at issues that can arise if you do subtraction with unsigned ints and try to imply negative values that can't be represented. We'll create some unsigned ints:

```
unsigned int x = 10;
unsigned int y = 20;
```

Now writing (y-x) is 20-10 in unsigned arithmetic, which results in 10 as expected:

```
std::cout << (y-x) << std::endl;
```

By contrast, the next example attempts (x-y) which is 10-20, but the unsigned arithmetic can't represent -10, and instead results in a very large positive value. The output is 4294967286, which is close to the maximum for an unsigned 32-bit integer.

```
std::cout << (x-y) << std::endl;
```

Casting values back to signed int

If we explicitly cast the result back to a signed integer, then we might get something usable again. The output of this is -10:

```
std::cout << (int)(x-y) << std::endl;
```

You can also do a casting operation to convert to signed int just by assigning an unsigned result to a signed int variable. This also outputs -10:

```
int z = x - y;
std::cout << z << std::endl;
```

However, casting a result is not always the best way to handle this! Instead, you may want to create temporary working copies of unsigned values as signed types. That way, you can do operations as usual and manually check whether a value is negative or positive, in those cases where it should not be negative. For example, if you are assigning an unsigned int value to a signed int, you can check whether the signed int shows a negative value. If it is negative, that means the unsigned value was too large to be accurately cast to signed. Here's a contrived example:

```
int test_val = (x-y);
if (test_val < 0) {
   // Note: The standard error stream (cerr) will be displayed in the terminal.
   // You can also handle it separately from the standard output stream (cout)
   // if you are logging things to files.
   std::cerr << "Warning: unsigned value cast to signed int resulted in a negative value" <<
std::endl;
}
```

Making direct comparisons between signed and unsigned ints can also cause issues. This next line may give a warning or error. We have commented it out for now:

```
// std::cout << (a<b) << std::endl;
```

Container sizes are often unsigned

We often refer to a generic data structure class as a "container". The C++ Standard Template Library (STL) provides many of these, such as std::vector. Many of these class types have a size() member variable that returns the number of items the container currently holds. It's important to note that in many cases, this will give you the size in an unsigned integer type. (The exact byte size of the specific unsigned integer type may vary.) So, although you should probably try to avoid using unsigned integers in your code except in very special circumstances, you will run into cases where you are comparing a signed int (perhaps an iteration counter) with an unsigned integer size. So, be prepared to safely handle unsigned ints!

Here is an example where the compiler will warn you it's comparing the signed integer i with the unsigned integer returned by size():

```
std::vector<int> v = {1,2,3,4};
for (int i=0; i < v.size(); i++) {
   std::cout << v[i] << std::endl;
}
```

You could handle this using various casting methods described above to make the warning message go away. You could simply use an unsigned int, or indeed std::vector<int>::size_type itself, as the type for the counter i. (However, this may cause you more trouble if you are doing common arithmetic operations on i for one reason or another, because again, you are introducing an unsigned integer to your code. Let's suppose for now that your containers won't need to contain an astronomical number of items.)

Now, consider the danger of trying to *subtract* from the unsigned int that represents the size. If we wrote the code in the following way, what could go wrong?

```
std::vector<int> v = {1,2,3,4};
for (int i=0; i <= v.size()-1; i++) {
   std::cout << v[i] << std::endl;
}
```

This code seems to work as expected, but only because the size is nonzero. If we had any situation where this loop might process an empty vector, the program would crash:

```
std::vector<int> v;
for (int i=0; i <= v.size()-1; i++) {
   std::cout << v[i] << std::endl;
}
```

Here, the vector v is initialized empty by default, so its size() is reported as 0. That means the expression "v.size()-1" will evaluate to -1 *interpreted as an unsigned integer*, in this case meaning a very large positive integer. So, on the first loop iteration, "0 <= (a very large positive integer)" will be evaluated as true, and the loop body will execute, evaluating v[0], even though there is no first item in v to access. As a result, this code will most likely cause a segfault and crash.

So, here, it is most reasonable to write it the "i < v.size()" way perhaps, but these other tricks would also solve this issue. Let's think about why:

```
// Casting to signed int first helps to ensure that the result
// of subtraction will truly be a signed negative value when size is 0:
for (int i=0; i <= (int)v.size()-1; i++) {
   // ...
}

// Rewriting the algebra to perform addition instead of subtraction
// helps to avoid going below 0:
for (int i=0; i+1 <= v.size(); i++) {
   // ...
}
```

In practice, for this code to actually be fully robust, you'd also need to check i against the maximum limit for an integer of its type, and make sure that the container never grew that large either. However, these are separate concerns. For now, just be sure to observe when you are converting to or from an unsigned integer type, and watch out for the potential problems it can cause.

Conclusion

Be careful when you are dealing with unsigned integers! In general, you should only use unsigned integers when you absolutely need to, such as if you are trying to maximize the range of positive values you can store in the same amount of memory. For everyday high-level programming purposes, signed integers may be all you need. If you need to ensure positive values during your code execution, you can write safety checks into the code to monitor values and issue warnings or errors if a certain range is exceed