Modern C++: The Good Parts/Printable version

Modern C++: The Good Parts

Contents

Getting a compiler

C++ is a **compiled language**, meaning that those who use it commonly use a **compiler** to convert C++ source code into something more machine-friendly, which is then executed. You will need a compiler to complete the exercises in this book, but fear not! Compilers are freely available and reasonably easy to set up. This page will attempt to help you decide which one(s) to install.

If you expect to learn GUI development with Qt later, you should install Qt Creator (http://qt-project.org/downloads#qt-creator), and make sure to check the box for the latest MinGW. (**Note:** If you're one of my students, get Qt Creator.)

Visual Studio Express for Windows Desktop (http://www.visualstudio.com/downloads/download-visual-studio-vs#d-express-windows-desktop) is a good free option which requires essentially zero setup, other than a very long installation. If you have money to spend, some other version of Visual Studio (http://www.visualstudio.com/downloads/download-visual-studio-vs) can serve you better, but you can make that decision later.

Clang (http://llvm.org/releases/download.html) can be a better option, except that its setup can be very involved, especially if you happen to want the latest version for Windows (in which case read this (http://clang.llvm.org/get_started.html#buildWindows)). Clang is well-known for very useful error messages.

The GNU Compiler Collection (GCC) (https://gcc.gnu.org/install/binaries.html) is another good option. On Windows, MinGW is little more than the compiler while Cygwin is like some Linux for your Windows. If you don't understand that last part, just go with MinGW.

If you don't trust my short list, see the significantly longer one (http://www.stroustrup.com/compilers.html) from the originator of C++, Bjarne Stroustrup.

Once you have a compiler installed, see the next page to make sure it's in working order.

Hello, world!

What are we waiting for? Here's your first C++ program. Save it in your editor (such as Qt Creator or Visual Studio), compile or build it, and run the executable. This will prove whether your compiler is working, which will be rather important for later exercises.

Code

```
#include <iostream>
#include <string>

int main()
{
    std::cout << "Hello, world!\n";
}
```

Explanation

Those first two lines tell the compiler what **includes** (pre-existing code) we need. `int main` is required for any C++ program, and the logic inside the curly brackets is everything that will be done. The particular way that C++ requires us to arrange code is called its **syntax**.

In this case, we send the text "Hello, world!" to the console.

These first two chapters have been kept short to allow for complications with setting up your compiler, but following chapters will pick up the pace.

Vocabulary

includes
: files containing code that we can use in our own program.

syntax
: the specific rules of a language, or basically what the language looks like.

Anatomy of a global greeting

Let's examine our program from last time in more detail.

Code & explanation

```
#include <iostream>
#include <string>
```

These includes are required by the standard, so anyone who wants to offer a proper C++ compiler must provide them. Several includes are required that way, and the set of them is called the **standard library**.

- `<iostream>` allows us to use `std::cout` and `std::cin`. All of our programs will include it, because they're all console programs.
- `<string>` is needed anytime you want a **string** (a value that represents text), which will also be all of our programs.

```
int main()
{
    std::cout << "Hello, world!\n";
}
```

As mentioned before, `int main` is required for all C++ programs. Ignore the parentheses and curly brackets for now; just know that they're necessary.

`std::cout` and `std::cin` are for console *out*put and *in*put, respectively. `std` is the **namespace**, which basically means who gave them to us, and this one is the *standard* namespace - i.e., these are from the standard library. The "c" just means C++ inherited them from its "parent" language, C.

The "\n" at the end of the string is a **line break** or **newline**; it causes further text to be printed on an additional line.

Vocabulary

standard library
: several useful pieces of code required by the C++ standard, shipped with all C++ compilers, all under the namespace `std`.

string
: a value that represents text.

namespace
: a directory of names, or where to look for something by name.

newline
: a character which separates two lines of text. Also called a **line break**. Syntax: `\n`

The world's response

Now, let's allow the world to return the greeting.

Code

```
#include <iostream>
#include <string>

int main()
{
    std::string input;

    std::cout << "Hello, world!\n";

    // Notice the arrows point to the right.
    std::cin >> input;
    std::cout << "The world says: " << input << "\n";
}
```

Explanation

We have a few new things here.

`std::string` means that `input` is a string **variable**. A variable is a named place in memory to store information. Since it's named, you can refer to it again and again, to put something there or to see what's already there. And as a string variable it can store text - and only text. Other kinds of variables exist, and we'll cover some of those in the next chapter.

`//` is a **line comment**, which means everything after it on the same line will be ignored by the compiler.

`std::cin` has an arrow that points in the opposite direction from the arrows of `std::cout`, because information is flowing in the opposite direction.

After `"The world says: "`, there's another arrow, and then another. This just appends more text to the output.

Exercises

- Extend the conversation between your program and the "world". Don't bother with whether the program is responding correctly to what is typed.

Vocabulary

variable
a named place in memory to store information.

line comment
causes the compiler to ignore the rest of a line. Syntax: `//`

Number crunching

The machine in front of you is really just a very fancy calculator, so let's make it calculate.

Code

```
#include <iostream>
#include <string>

int main()
{
    std::string input;
    int a, b;

    std::cout << "Enter two integers to multiply.\n";

    // Take input.
    std::cin >> input;
    // Parse it as an int.
    a = std::stoi(input);

    // Take input.
    std::cin >> input;
    // Parse it as an int.
    b = std::stoi(input);

    std::cout << a << " * " << b << " = " << a * b << "\n";
}
```

Explanation

So first we have a `std::string` variable named "input" and two `int` variables named "a" and "b". An `int` is an integer or whole number, so it can't have a decimal point.

Important: The = (equals sign) *does not mean equality*. In C++ it's the **assignment operator**; it puts the value on the right into the variable on the left, overwriting any previous value of that variable.

`std::stoi` is a **function**, and the name stands for **s**tring **to i**nt. It takes a string (in this case, the value of `input`) and converts it to an integer. If this isn't possible, or you enter a very large number, the program will crash and you'll probably see something about an "unhandled exception". How to prevent this issue will be addressed a few chapters later.

You can read about `std::stoi` at cplusplus.com (http://www.cplusplus.com/reference/string/stoi/) or cppreference.com (http://en.cppreference.com/w/cpp/string/basic_string/stol). It has some friends (http://www.cplusplus.com/reference/string/#functions), too (http://en.cppreference.com/w/cpp/string/basic_string#Numeric_conversions).

The multiplication operator in C++ is * (asterisk). Division is / (forward slash).

Try entering 3.14 as one of the integers. Oops! `std::stoi` finds the valid integer 3 and ignores the invalid input after it. You'll get the same result for 3sdjgh. The next few chapters should allow you to write a better integer parser.

Exercises

- Modify the above program so it accepts decimal points (floating-point numbers (https://en.wikipedia.org/wiki/Floating_point) or `float` values). Hint: Check the links on this page.

Vocabulary

int
: an integer or "whole number". Cannot represent fractions.

assignment operator
: puts the value on the right into the variable on the left, overwriting any previous value of that variable. Syntax: =

function
: a named piece of code. More on these later.

float
: a floating-point number or "decimal number". (That last one is a little confusing.) Can represent some fractions.

Now it gets interesting.

Up to now all of our programs have been very straightforward: Start at the top and read each line once until you get to the bottom. That changes now, because that's not really a very useful way to write a program. Useful programs usually repeat some things, or decide to not do some. This chapter covers how to select a course of action on the fly.

Code

```
#include <iostream>
#include <string>

int main()
{
    std::string input;
    int a, b;
    int greater;
    // This one starts out with a value.
    bool canDecide = true;

    std::cout << "Enter two integers to compare.\n";

    // Take input.
    std::cin >> input;
    // Parse it as an int.
    a = std::stoi(input);

    // Take input.
    std::cin >> input;
    // Parse it as an int.
    b = std::stoi(input);

    // This is equality, and it might be true now and false later.
    if(a == b)
    {
        // = does not mean equality, which means we can do this.
        canDecide = false;
    }
    else if (a < b)
    {
        greater = b;
    }
    else
    {
        greater = a;
    }

    if(canDecide)
    {
        std::cout << greater << " is the greater number.\n";
    }
    else
    {
        std::cout << "The numbers are equal.";
    }
}
```

Explanation

`bool` is short for Boolean (https://en.wikipedia.org/wiki/Boolean_data_type), which means yes or no, true or false. In this case the bool variable "canDecide" is initialized to the value `true`. Basically, the program assumes it "canDecide" which number is greater, until it sees evidence to the contrary.

Overwriting bools in this way can be very useful.

`==` is the equality operator; it determines whether two values are equal, and returns true or false. Note: It does not change the value of anything.

Anything that looks like `if(...){...}` is an **if statement**. `a == b` is an **expression** which evaluates to a bool. An expression is some code which can be resolved to a value. The curly brackets just after `if(a == b)` contain code which will run if a equals b.

`else if` means "Otherwise, if..." and is only tested if the **condition** (the bool) is false for the first if statement. `else` by itself doesn't test any condition, but only runs if none of the if statements in its chain have been run.

When some logic is selected for execution in this way, "control" is said to "flow" into the logic. So `if` statements are considered **control flow** statements or **control structures**, a set which contains some other constructs to be introduced later.

Exercises

- Write a calculator, asking the user for two floats and an operator and then printing the result.

Vocabulary

bool
: one of two possible values, `true` or `false`. Also called a **condition**.

expression
: some code which can be resolved to a value.

if statement
: runs if its condition is true. Syntax: `if(condition){ }`

control flow
: which code runs, and in what order.

control structure
: a statement which modifies control flow. Also called a **control flow statement**.

Switching things up

In the previous chapter, you received your first assignment: to write a simple calculator. At the time, if statements were your only way to make a decision; this chapter introduces the **switch statement**, which works similarly to the if statement but is more suited to problems like the calculator.

Code

Here's a calculator built around a switch statement:

```
#include <iostream>
#include <string>

int main()
{
    std::string input;
    float a, b;
    char oper;
    float result;

    std::cout << "Enter two numbers and an operator (+ - * /).\n";

    // Take input.
    std::cin >> input;
    // Parse it as a float.
    a = std::stof(input);

    // Take input.
    std::cin >> input;
    // Parse it as a float.
    b = std::stof(input);

    // Take input.
    std::cin >> input;
    // Get the first character.
    oper = input[0];

    switch (oper)
    {
    case '+':
        result = a + b;
        break; // DON'T
    case '-':
        result = a - b;
        break; // FORGET
    case '*':
        result = a * b;
        break; // THESE
    case '/':
        result = a / b;
        break; // !!!
    }

    std::cout << a << " " << oper << " " << b << " = " << result << "\n";
}
```

Explanation

`std::stof` (http://en.cppreference.com/w/cpp/string/basic_string/stof) is similar to `std::stoi`, except it converts to float.

`input[0]` is the first character in the string `input`. Anytime you see square brackets with a number (or variable) between them, the number is zero-based. This means that what people normally call the "first" character is at index 0, the "second" is at index 1, and so forth.

The switch statement has exactly the same effect as the if-else chain you wrote for your own calculator. Notice that every `case` label has a matching `break` statement; be careful that this is true of any switch statement you write, because otherwise you might get some very surprising behavior. Specifically, without a break statement, control will flow right past any labels, causing the logic under them to be executed in multiple cases.

Exercises

- Explore what happens when you divide one int by another, and how that differs from doing the same with two floats, and an int and a float. Discuss your findings with your instructor.

Vocabulary

switch statement
: uses its expression to select which `case` label to jump to.

Getting loopy

For this chapter, we have a calculator willing to perform more calculations than you are willing to input.

Code

```
#include <iostream>
#include <string>

int main()
{
    std::string input;
    float a, b;
    char oper;
    float result;

    // The ultimate truism, this never stops being true.
    while(true)
    {
        std::cout << "Enter two numbers and an operator (+ - * /).\nOr press Ctrl+C to exit.\n";

        // Take input.
        std::cin >> input;
        // Parse it as a float.
        a = std::stof(input);

        // Take input.
        std::cin >> input;
        // Parse it as a float.
        b = std::stof(input);

        // Take input.
        std::cin >> input;
        // Get the first character.
        oper = input[0];

        switch (oper)
        {
        case '+':
            result = a + b;
            break;
        case '-':
            result = a - b;
            break;
        case '*':
            result = a * b;
            break;
        case '/':
            result = a / b;
            break;
        }

        std::cout << a << " " << oper << " " << b << " = " << result << "\n";
    }
}
```

Explanation

This program will run until your terminal stops running it (because you press Ctrl+C, or click the X). That's because the **while loop**, every time control reaches its bottom, jumps back up to the top - *if* its condition is still true, which it always will be in this case.

If we weren't in a context where the user could easily halt the loop, the `while(true)` would be a very bad idea. Instead of `true`, you can put any Boolean expression (like `a == b`) or `bool` variable between the parentheses.

Suppose we wanted to allow up to ten **iterations** (passes through the loop) instead of an unlimited number. We could do this:

```
int i = 0;
while(i < 10)
{
    //...
    i++;
}
```

And that would work just fine.

`i++` is the same thing as `i = i + 1`, and results in `i` being one greater than it was. It's also the same as `i += 1`.

But C++ provides a prettier way to write that, the **for loop**:

```
for(int i = 0; i < 10; i++)
{
    //...
}
```

For the first iteration, `i` will have the value 0. For the last iteration, it will have the value 9. It will never (usefully) have the value 10 because, just like with our equivalent while loop above:

1. the iteration runs;
2. then `i` is incremented;
3. then `i < 10` is checked, and if it's false the loop is exited.

Nonetheless, the loop does run 10 times.

Exercises

under construction

Vocabulary

while loop
continues looping until its condition is false.
iteration
one pass through a loop. Also, the action of looping or "iterating".
for loop
more suited to counting than a while loop is.

At last, functions

Functions are so important to programming that teaching control structures has been rather tedious without them, so with great relief let's see how to make our own.

Code

```
#include <iostream>
#include <string>

// Calls to this function evaluate to float.
float readFloat()
{
    std::string temp;
    std::cin >> temp;
    return std::stof(temp);
}

// This one returns char.
char readChar()
{
    std::string temp;
    std::cin >> temp;
    return temp[0];
}

// Another function returning float.
float evaluate(float a, char oper, float b)
{
    switch (oper)
    {
    case '+':
        return a + b;
    case '-':
        return a - b;
    case '*':
        return a * b;
    case '/':
        return a / b;
    }
}

int main()
{
    float a, b;
    char oper;

    // The ultimate truism, this never stops being true.
    while(true)
    {
        std::cout << "Enter two numbers and an operator (+ - * /).\nOr press Ctrl+C to exit.\n";

        // Parse two floats from standard input.
        a = readFloat();
        b = readFloat();
        // And a character.
        oper = readChar();

        std::cout << a << " " << oper << " " << b << " = " << evaluate(a, oper, b) << "\n";
    }
}
```

Explanation

Woohoo, the code just got much easier to read, once you understand functions.

As the comments kind of explain, each function has a **return type** before its name. A function's return type will be the type of any calls to that function. A **function call** is an expression which causes a function to be run, such as `std::stof(temp)`. When the control flows over a `return` statement, the containing function exits. The function's **caller** then receives the **return value** as the value of the function call and continues from that point.

`return` can stand in for `break` in a `switch`. The point is that control doesn't "fall through" from one case to another. C++ does allow that to happen, but it can be very confusing and it's rarely useful.

By now, you've probably noticed that `main` has a return type of int. It's the only non-void function that you don't need to `return` from; by default, it returns 0 when control reaches the bottom. 0 indicates that the program was successful, or ran without problems; any other value tells whoever ran the program that something went wrong. These values are called "exit codes", but you won't need to specify one anytime soon.

Functions also have **parameters**, the variables declared between their parentheses. Any function call must provide a value for each of the function's parameters; these values are called **arguments**.

Finally, a function must appear in the source before any call to it. However, C++ offers an important loophole to this that we'll cover later.

Exercises

under construction

Vocabulary

return type
  the type of value that any call to a specific function will be resolved to.

function call
  an expression which causes a function to be run.

caller
  the function which contains a specific function call.

return value
  the value that a specific function call will be resolved to.

void
  a return type meaning that nothing is returned.

parameter
  a variable in a function which is assigned by each call to that function.

argument
  the value of a parameter for a specific call to a function.

Let's take a breather.

Up to now, we've only covered what has been used in our programs, but that's leaving some irritating gaps. This chapter is to fill those gaps.

Loops

do..while

There's another kind of loop we haven't mentioned:

```
do
{
    //...
}
while(condition);
```

As you might have guessed, this is very similar to a while loop. A **do..while loop** reverses the two things done by an ordinary while loop, which are:

1. check condition, exit if false;
2. run loop body.

A while loop repeats those two steps over and over until the condition tests false. A do..while loop, in contrast, repeats them like this:

1. run loop body;
2. check condition, exit if false.

The effect is that a do..while loop guarantees one iteration, whereas a while loop might not run its body at all. This is useful, in particular, when you're reading data from somewhere else, like a file, and whether you should continue depends on what you've read.

break

Loops can be a whole lot more complicated than "start..repeat until X..exit". That complexity can be a very bad thing, but it can also be very useful.

For example, completing a loop is not the only way out. Remember the `break` statements used in switches? Well, they can also exit loops. If you have a loop in a loop in a loop (another example of nasty complexity), `break` will only exit the smallest one it's in. If you have a switch in a loop, `break` inside the switch will only exit the switch.

If you like, imagine that the language starts from the `break` statement, searches backward through the code, and exits the first loop or switch it finds.

Here's an example of `break` in a loop:

```
#include <iostream>
#include <string>

int main()
{
    std::string input;
    int count;

    std::cout << "Welcome to the lister! How many items would you like to list? > ";
    std::cin >> input;
    count = std::stoi(input);

    std::cout << "If you change your mind, enter STOP after the last item.\n\n";
    for(int i = 0; i < count; i++)
    {
        std::cout << (i + 1) << ". ";
        std::cin >> input;
        if(input == "STOP")
        {
            break;
        }
    }
}
```

`continue`

Switches

Functions