ACM C++ Guide

Matt Hancock

Contents

1 Introduction

Welcome to the ACM C++ guide, a manual for quickly learning C++ for mathematical and scientific computing applications. The goal of this guide is not to make you a C++ expert, but to quickly teach you enough of the C++ fundamentals and design patterns to help you off the ground. If you should like to go beyond this guide, a few references are listed below.

This guide is split into sections. The sections are ordered in such a way that topics in later sections often depend on topics from previous ones. You'll find code snippets peppered throughout the sections. At the end of each major section, you'll find exercises for practice.

1.1 A little about the language

Before you dive in, here is a little about the C++ programming language:

C++ is an extension of the C programming language. Both C and C++ are **statically-typed** and **compiled** languages, which means that the **type** of variables used in your source code must declared explicitly and is checked at when the program is compiled (i.e., translated into a machine executable file).

One (among many) difference between C++ and C, however, is that C++ provides many data structures for the object-oriented programming paradigm. This essentially allows the software writer (that's you) to create custom, complex, reusable data structures. The object-oriented paradigm is extremely useful, but we will only touch the surface of it in this guide.

1.2 Additional references

- C++ reference
- C++ tutorials
- C++ wiki
- A compiled list of C++ textbooks from stackoverflow

2 Getting started

The particular programming tools that you choose to use will likely be largely influenced by the operating system that you use. We will use free tools (often developed for GNU / Linux systems) in this guide. These tools are mostly available in other (non UNIX-like) operating systems as well. For example, on Windows, you could use Cygwin, or install a dual boot with some Linux distribution (e.g., Ubuntu). On the other hand, MAC OSX, being a BSD-derived system, has many of the required tools already available (although, a command line utility, Brew, makes building and installing other tools quite simple).

In the following two sections, we'll talk about the two basic types of software that you'll need to begin writing C++ programs.

2.1 Text editors

The text editor that you choose to use should be be any program capable of editing plain text files. However, you may find that it's more productive to write in an editor who offers features such as syntax highlighting, code-completion, bracket-matching, or other features. Here are some popular free text editors:

- Atom is a recently open-sourced GUI editor which some have compared to the very popular non-free editor, Sublime Text.
- Emacs is another powerful editor, which allows for highly optimized workflows.
- Gedit is a nice and simple GUI editor, which is the default in GNOME desktop environment.
- Kate is a another simple GUI editor, which is the default in the KDE desktop environment.
- Vim is a modal editor with a steep learning curve. It offers highly efficient means to edit text, and is available (or it's predecessor, vi) by default on nearly all UNIX-like operating systems.

2.2 Compilers

Second, you'll need a program called a **compiler**. A compiler translates the high-level C++ language into an executable program. In this guide, we will use the `g++` compiler which is freely available through the gnu compiler collection (gcc).

`g++` is a program which you typically call from the command line, which takes as input, your C++ source code file, and produces as output, a binary executable file.

2.3 Writing a program

Let's create our first C++ program, the obligatory "Hello, world!". First, fire-up your text editor and create a file called, `hello.cpp`, with the following contents:

```
1  #include <iostream>
2
3  int main() {
4    std::cout << "Hello, world!";
5    return 0;
6  }
```

Now, to compile the program, execute:

```
g++ hello.cpp
```

Followed by:

```
./a.out
```

By default, `g++` calls the resulting binary executable, `a.out`, but you can specify a different output file name by using the `-o` flag:

```
g++ hello.cpp -o my_executable_file.bin
```

Note that in Windows, executable files typically end in `.exe`. In Unix-like systems, there is no particular file-extension type for executables.

2.3.1 Deconstruction of the "Hello, world!" program

Let's look at this program line-by-line:

1. In line 1, we are including a library called `iostream`. A standard library (one that is included with C++) is usually included inside angular brackets. The `iostream` library is used for printing to the screen or prompting for user input.

2. Line 2 is blank. Whitespace is ignored in C++.
3. In line 3, we are declaring a function called `main` which has a return type of `int` (short for integer). Functions are followed by parentheses, listing their arguments (our `main` funcion has no arguments in this case), followed by the function body. The function body is enclosed by brackets. The function `main` is special - it is the main function (surprise) that is executed when you actually run your program.
4. In line 4, we are giving the `Hello, world!` string to an object called `cout`, which prints the string. Note that `cout` is prefixed by `std`. The syntax, `std::cout`, means that `cout` belongs to the **namespace** called `std`. Namespaces are mechanisms for preventing name conflicts.
5. In line 5, we return the value 0. This signifies to the program that called the executable `hello.cpp` program (the parent program, you could say) that our program exited successfully (i.e., without error).
6. Line 6 closes the the the body of the function, `main`.

3 Data types

As we discussed previously, you must explicitly declare the type of a variable. So, in this section, we'll talk about the main variable types you'll use, namely boolean, integer, floating point types. In the section on 6, we'll discuss how to build our own custom data types.

3.1 Boolean

A boolean data type is either `true` or `false`. There are a number of operators between these types, illustrated in the code snippet below (note that lines starting with `//` are comments are ignored by the compiler):

```
bool a,b,c; // Declare the type of variables a, b, and c.
a = true;
b = false;

// !  is logical negation when applied to a single variable.
c = !a; // c is false.

// && is logical and.
```

```
c = (a && b); // c is false.

// || is logical or.
c = (a || b); // c is true.
```

We don't often use boolean variables by themselves, but rather as a result of comparing two other data types (such as comparing if one integer is less than another integer).

3.2 Integer types

There are a variety of integer types in C++. Below, we illustrate a couple. These can be modified further using the `short` and `long` keywords, changing the number of bytes occupied by the variable (and hence the maximum and minimum size the variable can take on).

```
int a = 6; // initialize a to 6.
unsigned int b = 7; // initialize b to 7.
int c; // declare c to be an integer variable.

a = 6;
b = 7;

c = a / b; // c is 0
c = b / a; // c is 1
c = b % a; // c is 1 (% is the integer remainder or modulo operator)
c = a - b; // c is -1
c = a > b; // c is 0 (boolean gets cast to integer)
c = a < b; // c is 1 (boolean gets cast to integer)
c++;       // c is 2 (++ is shorthand for c = c + 1)

b = a - b; // b is 4294967295 (-1 gets cast to unsigned)
b = b + 1; // b is 0 (b was previously the largest unsigned, so adding one circles
b += 7;    // b is 7 (+= is shorthand for b = b + 7;
```

In the above, we've illustrated the use of signed and unsigned integer types and the operators between them. It is important to take care when you assign a result to a variable that doesn't match the type of the result. In many cases, the result gets implicitly cast to the type of variable being assigned to. The result may or may not match your expectations, as shown above.

3.3 Floating point types

There are two main floating point data types in C++, `float` and `double`, which correspond to IEEE 32- and 64-bit floating point types.

```
#include <iostream>
#include <limits>

int main() {
  float a; //  Declare a single precision float.
  double b; // Declare a double precision float.

  // Print the max value of a float type.
  std::cout << std::numeric_limits<float>::max() << std::endl;
  // prints 3.40282e+38


  // Print the max value of a double type.
  std::cout << std::numeric_limits<double>::max() << std::endl;
  // prints 1.799769e+308


  // Print machine epsilon of a float type.
  std::cout << std::numeric_limits<float>::epsilon() << std::endl;
  // prints 1.19209e-07


  // Print machine epsilon of a double type.
  std::cout << std::numeric_limits<double>::epsilon() << std::endl;
  // prints 2.22045e-16

  return 0;
}
```

3.4 Casting

Sometimes it is useful to explicitly cast one variable type as another. This can be done like the following:

```
int a; double b = 3.14159;

```

```
a = (int) b;

std::cout << a << std::endl;
// prints 3
```

3.5 The const modifier

If the value of some variable should not change, you can use the `const` keyword to protect its status. It is typical to denote `const` variables with all caps. Try to compile the following program:

```
const double PI = 3.14159;

PI = 3.0;
```

You will see an error like, `error: assignment of read-only variable ‘PI’`.

3.6 The typedef keyword

Suppose you have a large numerical experiment, where all your code used floating point of type `double`. Your curious about how the results will be affected by changing the floating point type to single precision `float` type. One solution would be to run a "find and replace" in your editor, but something about that doesn't feel right.

Instead, we can use the `typedef` statement to define types:

```
// Define "int_type" to be a short int.
typedef short int int_type;

// Define "float_type" to be single precision float.
typedef float float_type;

// Define "array_index_type" to be unsigned long int.
typedef unsigned long int array_index_type;

int_type a = -17;
float_type b = 1.14;
array_index_type c = 9;
```

3.7 Pointers

Pointers are variables that hold the **memory address** for a variable of a specific type. Pointers are declared by specifying the variable type, followed by the * symbol, followed by the name of the pointer variable, e.g., `double * x` defines a "pointer to double" variable. The variable, `x`, therefore, does not hold the value of a `double` type, but rather, the memory address for a variable of type, `double`. The memory address for a variable can be obtained by the `&` operator.

```
double * a;
double b = 7;

// This obtains the memory address of ‘b‘.
a = &b;

// Prints some memory address (starts with 0x)
std::cout << a << std::endl;
```

Similar to obtaining the memory address from a regular variable, using the `&` operator, you can use the * symbol before a pointer to access the variable value held at the memory location of the pointer. In this context, the * symbol is called the **dereference operator**. This is probably better understood with a short example:

```
double * a;
double b = 7.3;
double c;

// Now ‘a‘ holds the memory address of ‘b‘.
a = &b;

// ‘*a‘ obtains the value of the variable
// at the memory address held by ‘a‘.
// So, ‘c‘ is 7.3.
c = *a;
```

3.8 Arrays

The length of an array can be fixed or dynamic, and how you declare the array depends on this.

3.8.1 Fixed length arrays

```
double a[5];

a[0] = 1.0;
// etc.
```

3.8.2 Dynamic length arrays

Dynamic length arrays are made possible through pointers:

```
// This allocates memory for 5 double types.
double * a = new double[5];

// Afterwards, you can treat ‘a‘ like a normal array.
a[0] = 1.0;
// etc...

// Whenever you use the ‘new‘ keyword, you must
// delete the memory allocated when you’re done by hand.
delete [] a;

// We can change the size of ‘a‘.
a = new double [10];

a[0] = 2.0;
// etc...

delete [] a;
```

Note that omitting the first `delete` statement will cause no error. However, the memory allocated by the first `new` statement will not be freed, and thus inaccessible. This is bad because the memory cannot be allocated to other resources. You should generally try to avoid manually memory management when possible, but a good tool for debugging memory problems is called valgrind.

4 Control structures

4.1 Conditionals

Often a code block should only be executed if some condition is true. Below, we generate a random number between 0 and 1; print the number; and, print whether or not the number was greater than 0.5.

```
#include <iostream>
#include <stdlib.h>
#include <time.h>

int main() {
  // Seed the random number generator based on the current time.
  srand(time(NULL));

  // rand() produces a random integer between 0 and RAND_MAX.
  double num = rand() / ((double) RAND_MAX);

  std::cout << "num: " << num << "\n";

  if (num < 0.5) {
    std::cout << "num was less than 0.5.\n";
  }
  else {
    std::cout << " was greater than 0.5.\n";
  }

  return 0;
}
```

You can follow `else` immediate by another `if` to have mutiple mutually-exclusive blocks:

```
#include <iostream>
#include <stdlib.h>
#include <time.h>

int main() {
  // Seed the random number generator based on the current time.
  srand(time(NULL));

  // rand() produces a random integer between 0 and RAND_MAX.
  double num = rand() / ((double) RAND_MAX);

  std::cout << "num: " << num << "\n";

  if (num >= 0.75) {
    std::cout << "num was between 0.75 and 1.\n";
  }
  else if (num >= 0.5) {
    std::cout << "num was between 0.5 and 0.75.";
  }
  else if (num >= 0.25) {
    std::cout << "num was between 0.25 and 0.5.";
  }
  else {
    std::cout << "num was between 0 and 0.25";
  }

  return 0;
}
```

The conditions are checked in the order that they're written. So, for example, in the second condition, we don't need to specify `num >= 0.5 && num < 0.75` because we know that this condition will only be checked if the previous was false.

4.2 Loops

We discuss two main structures for iterating – the `for` and `while` loops.

4.2.1 The for loop

The `for` loop requires three specifications – the iteration variable initialization, the termination condition, and the update rule. The body of the loop follows these three specifications. Shown below, we declare an array; assign to its components; and, print the current component to the screen.

```
int length = 11;
double x[length];

```

```
for(int i=0; i < length; i++) {
  // Assign to each array component.
  x[i] = (double) i / (length - 1);

  // Print the current component.
  std::cout << "x[" << i << "] = " << x[i] << std::endl;
}
```

```
x[0] = 0
x[1] = 0.1
x[2] = 0.2
x[3] = 0.3
x[4] = 0.4
x[5] = 0.5
x[6] = 0.6
x[7] = 0.7
x[8] = 0.8
x[9] = 0.9
x[10] = 1
```

You can nest loops, i.e., loops inside of loops inside of . . .

Below, is an example of a double loop for creating and accessing matrix data stored in a flat array. The matrix data is stored in row-major order. This means the first `n_cols` elements of the array named, `matrix`, will contain the first row of the matrix, the second `n_cols` elements of `matrix` will contain the second row, etc. . .

```
int n_rows = 4;
int n_cols = 3;

// Row-major matrix array.
double matrix [n_rows*n_cols];

// temporary index.
int k;

for(int i=0; i < n_rows; i++) {
  for(int j=0; j < n_cols; j++) {
    // Convert the (i,j) matrix index to the "flat" row-major index.
    k = i*n_cols + j;

    // Assign a value of 1.0 to the diagonal,
    // 2 to the off-diagonal, and 0 otherwise.
    if (i == j) {
      matrix[k] = 1.0;
    }
    else if ((i == (j+1)) || (i == (j-1))){
      matrix[k] = 2.0;
    }
    else {
      matrix[k] = 0.0;
    }
  }
}


// Print the matrix elements.
for(int i=0; i < n_rows; i++) {
  for(int j=0; j < n_cols; j++) {
    k = i*n_cols + j;

    std::cout << matrix[k];
    if (j != (n_cols-1)) {
      std::cout << ", ";
    }
  }

  if (i != (n_rows-1)) {
    std::cout << "\n";
  }
}
```

```
1, 2, 0
2, 1, 2
0, 2, 1
0, 0, 2
```

4.2.2 The while loop

A `while` loop iterates while a condition is met. Essentially, it is a `for` loop without an update variable. In the following example, we approximate the geometric series:

$$1 = \sum_{n=1}^{\infty} \left(\frac{1}{2}\right)^n$$

The loop exits when the absolute error,

$$\text{absolute error} := 1 - \sum_{n=1}^{N} \left(\frac{1}{2}\right)^n$$

is less than some specified value, which we call ‘tol‘ (for tolerance).

```
double sum = 0.0;
double base = 0.5;
double pow = base; // initialize to base^1
double tol = 1e-4;
int iter = 1;

while((1-sum) >= tol) {
  // Add ‘pow‘ to ‘sum‘.
  sum += pow;
  // Update ‘pow‘ by one power of ‘base‘.
  pow *= base;

  std::cout << "Iteration: " << iter << ", Sum: " << sum << "\n";
  // Update the ‘iter‘ val by 1.
  iter += 1;
}
```

```
Iteration: 1, Sum: 0.5
Iteration: 2, Sum: 0.75
Iteration: 3, Sum: 0.875
Iteration: 4, Sum: 0.9375
Iteration: 5, Sum: 0.96875
Iteration: 6, Sum: 0.984375
Iteration: 7, Sum: 0.992188
Iteration: 8, Sum: 0.996094
```

```
Iteration: 9, Sum: 0.998047
Iteration: 10, Sum: 0.999023
Iteration: 11, Sum: 0.999512
Iteration: 12, Sum: 0.999756
Iteration: 13, Sum: 0.999878
Iteration: 14, Sum: 0.999939
```

4.2.3 The break keyword

The `break` keyword provides a mechanism for exiting the direct parent loop for which the `break` statement is placed. For example:

```
for(int i=0; i < 3; i++) {
  while(true) {
    std::cout << "Entering infinite loop number " << (i+1) << "\n";
    break;
  }
  std::cout << "We escaped the infinite loop!\n";
}
```

```
Entering infinite loop number 1
We escaped the infinite loop!
Entering infinite loop number 2
We escaped the infinite loop!
Entering infinite loop number 3
We escaped the infinite loop!
```

The previous example is contrived, but there are situations, where you might find the break statement within an infinite loop useful. Of course, you should avoid this sort of thing if there is a more straight-forward approach.

4.3 Exercises

1. Given integers, n and k, write a program to compute the binomial coefficient, $\binom{n}{k}$.

2. The series, $\displaystyle\sum_{n=1}^{\infty} \frac{1}{n^2}$, converges to $\displaystyle\frac{\pi^2}{6}$. Create a program that approximates this series up to some specified tolerance, printing the absolute error at each iteration.

3. Fix numbers, a and b. Let $x_0 = a$ and $x_N = b$. Let $\Delta x = \frac{b-a}{N}$ and $x_i = a + i \cdot \Delta x$, for $i = 0, 1, \ldots, N$. The left endpoint Riemann sum approximation to the integral, $\displaystyle\int_a^b x^2 dx$, is given by $\displaystyle\sum_{n=1}^{N} (x_i)^2 \Delta x$. Write a program with $a = 0$ and $b = 1$, which successively halves Δx (starting from the initial value of $\Delta x = 0.5$) until the absolute error between the approximation and the true integral value is less than some specified tolerance. Record the absolute error at each iteration.

4. Maybe do something with a matrix.

5 Functions

6 Object-oriented programming

7