Introduction to C++

Rob Hackman and Xiao-bo Li

Winter 2024

University of Alberta

Table of Contents

The C++ Programming Language

The C++ language was created by Bjarne Stroustrup, originally intended to provide object-oriented programming features to the C language ("C with classes")

C++ today is vastly different than the language at conception, and C is also very different today than it was when C++ was first created.

At one point every valid C program was a valid C++ program, and C++ could be said to be a superset of C. Today neither of these claims are true.

C++ and C

C++ and C share a lot in common. However, good modern C++ code will look very little like C code, and following C paradigms in C++ is a surefire way to write *bad* C++ code. Some things we learned from C that will carry over however:

- Lots of basic syntax
- Same basic types
- The memory model, and concept of memory management
- The preprocessor and compilation model
- Strong static type system

What's different is... pretty much everything we want to do.

Compiling C++ programs

We have been using `gcc` to compile our C programs. We'll use `g++` to compile our C++ programs.

- `gcc` — the GNU C compiler
- `g++` — the GNU C++ compiler

All of the compiler flags we've learned about so far for `gcc` will also work with `g++`, we'll also be compiling all of our programs with the option `-std=c++17` which specifies we're compiling with the 2017 C++ standard. Now your programs should all be compiled with flags

```
g++ -std=c++17 -Wall -Wvla -Werror
```

C Libraries in C++

The libraries we used in C are available in C++, however the include name changes. For any header we included from C such as `<stdio.h>`, `<stdlib.h>`, and `<string.h>` we instead include them with the name `<cstdio>`, `<cstdlib>`, and `<cstring>`. Namely we remove the `.h` and prepend the letter `c`.

However, this information is only for your knowledge as in this class we will not use any C libraries in our C++ programs.

If we can't include these libraries how do we do our basic things like read input and write output? How do we allocate memory if we can't access malloc and free in `stdlib.h`? How do we work with strings? We do it the C++ way.

Table of Contents

C++ I/O

Input and output in C++ uses a *stream* abstraction. The header for our basic I/O operations is `<iostream>`.

`<iostream>` has defined in it three objects named `std::cout`, `std::cin`, and `std::cerr`. These objects represent the standard output, standard input, and standard error streams our program is connected to.

In C++ we read data from an input stream with the input operator (>>), and write data to an output stream with the output operator (<<).

Hello World — the C++ way

```cpp
#include <iostream>
int main() {
  std::cout << "Hello world" << std::endl;
}
```

Reading input in C++

```cpp
#include <iostream>
int main() {
  int n;
  std::cin >> n;
  for (int i = 0; i < n; ++i) {
    std::cout << "na";
  }
  std::cout << " Batman!" << std::endl;
}
```

The input and output operators

The input and output operators >> and << shown earlier are for reading data from or writing data to a stream.

A good way to remember which operator is which is that they "point" in the direction the data is flowing.

C++ just uses the type of the operand to determine what is being read in, so we don't have to specify what we'd like to read in like with `scanf`.

The `std` namespace

The examples above all had to refer to entities from the standard library with `std::` prefixed before them.

`::` is the scope resolution operator, the left-hand operand should be a scope where an entity exists and the right-hand operand is the name of the entity. The left hand operand is usually a namespace or a class.

`std` is the name of the scope for the standard namespace. In C++ namespaces are used so that names pulled in from a library don't have to pollute the global namespace. Sometimes we may want to be able to access entities from the `std` namespace without the scope resolution operator.

The `using` directive

You can pull identifiers from a namespace into your global or local scope with the `using` directive. When identifiers are pulled into your scope you may use them without specifying their source.

```
#include <iostream>
using namespace std;

int main() {
  int x;
  cin >> x;
  cout << "You entered: " << x << endl;
}
```

In the above code we can reference `cin`, `cout`, and `endl` without prefacing them with `std::` — since we place the `using` directive in our global scope they have been pulled in to our global scope.

Failed reads in C++

in C the `scanf` function would return the number of items read in, and the programmer would have to check that return value to determine whether or not the read failed.

How do we check for failed reads from a stream in C++?

Failed reads in C++

in C the `scanf` function would return the number of items read in, and the programmer would have to check that return value to determine whether or not the read failed.

How do we check for failed reads from a stream in C++?

The input and output stream types are abstract data types that provide a huge amount of functionality, including ability to check for failed reads/writes.

Failed reads in C++

in C the `scanf` function would return the number of items read in, and the programmer would have to check that return value to determine whether or not the read failed.

How do we check for failed reads from a stream in C++?

The input and output stream types are abstract data types that provide a huge amount of functionality, including ability to check for failed reads/writes.

Note: A failed read will also write 0 to the variable that was being read into in C++11 and up.

The `fail` and `bad` bits

Each stream in C++ stores two bits that indicate when an error has occured:

- fail — this bit is set when a read failed, usually from a logical problem that is recoverable (e.g. reading an int when the next item in the stream is not an int).
- bad — this bit is set when a serious error has occured leaving the stream in a broken state, much less common and not very recoverable.

When either of these bits are set for a stream, any attempts to read or write to that stream will not be performed.

The `fail` method

We can call the `fail` function on a stream which returns `true` if the fail *or* bad bit is set for that stream, and `false` otherwise.

```
cin.fail();
```

Functions we call on specific pieces of data are called *methods*, you've seen them in Python before. Unlike in C, C++ has methods. More on them later.

We access a particular method of a piece of data the same way we accessed a member of a structure in C, with the member-of operator (the dot `.`)

Reading all integers from stream

We can then use the `fail` function to read all integers from the input stream until we fail.

```
int main() {
  int x;
  int total = 0;
  cin >> x;
  while (!cin.fail()) {
    total += x;
    cin >> x;
  }
  cout << "Total was: " << total << endl;
}
```

Implicit conversions

Abstract data types like the stream types can provide *implicit conversion* functions. These functions allow for the type to be implicitly converted to another in a meaningful way.

The stream abstraction in C++ allows for streams to be implicitly convered to `bool`s, when used as a `bool` a stream just produces the negation of its `fail` function. So this program is equivalent to the previous one.

```
int main() {
  int x;
  int total = 0;
  cin >> x;
  while (cin) {
    total += x;
    cin >> x;
  }
  cout << "Total was: ";
  cout << total << endl;
}
```

Chaining output operators

We've already shown chaining print operators such as

```
cout << "Total is: " << total << endl;
```

This line included three output operators. The output operators are resolved left to right and *return back the stream operand*. The operand must return the stream itself, or else the chaining wouldn't be possible. So the above is equivalent to

```
(((cout << "Total is: ") << total) << endl);
```

And each bracketed expression evaluates to the stream operand, in this case `cout`, for use in the next expression.

Chaining input operators

We can also chain together input operators.

```
int x, y;
cin >> x >> y;
```

The behaviour is the same, this performs the input operations from left to right which each expression evaluating to the stream operand itself which is then used as the left hand operand for the subsequent expression.

Combining I/O operator return value and stream implicit conversion

Being clever we can combine the fact that the I/O operators return back the stream itself with the fact that streams can be implicitly converted to a bool determining if a read was successful or not, and rewrite our program.

```
int main() {
  int x = 0;
  int total = 0;
  while (cin >> x) {
    total += x;
  }
  cout << "Total: " << total << endl;
}
```

Reading all ints

What if we want to read all integers from the input stream, skipping over any non-integer inputs as we did in C?

How can we read from a stream that has its `fail` bit set due to a failed read?

How can we skip over offending characters in the input stream?

.

How can we discern a read that failed for seeing a non-integer input from a read that failed due to reaching EOF?

The `clear` method

We can't read from a stream that has its `fail` or `bad` bit set, and the `cin` stream will have the `fail` bit set after we attempt to read an integer and see something else.

The `clear` method can be called on a stream to reset the `fail` and `bad` bits to off.

Clearing the stream only resets these bits, it does nothing to ensure that future operations will succeed. In our case, if our last read failed because the next input wasn't an integer then the next read will fail too unless we remove the offending character(s) from the stream.

The `ignore` method

In C when we wanted to pluck characters from the front of our input stream we had to manually read characters, in C++ streams have a method `ignore` which removes the immediate next character from that stream.

Each call to `ignore` will remove exactly one character from the front of that stream — useful when we want to, for example, ignore inputs that are not integers.

Read all ints, attempt 1

So with `clear` and `ignore` we can attempt to read all integers from the stream, regardless of whether or not there are other characters in the stream.

```
int main() {
  int x = 0, total = 0;
  while (true) {
    if (cin >> x) {
      total += x;
    } else {
      cin.clear();
      cin.ignore();
    }
  }
}
```

Read all ints, attempt 1

So with `clear` and `ignore` we can attempt to read all integers from the stream, regardless of whether or not there are other characters in the stream.

This program loops forever, however, even when receiving EOF. Everytime a read fails it clears the stream's fail and bad bits, tries to ignore the next character, and read again. If that read failed because of EOF there is nothing more to read.

```
int main() {
  int x = 0, total = 0;
  while (true) {
    if (cin >> x) {
      total += x;
    } else {
      cin.clear();
      cin.ignore();
    }
  }
}
```

The `eof` method

We can check if we've reached the end of a stream with the `eof` function.

This function returns `true` when an attempted read on a stream has failed due to seeing EOF.

When a read from `cin` fails we can use `cin.eof()` to check if it failed due to seeing EOF or not.

Read all ints attempt 2

```
int main() {
  int x = 0, total = 0;
  while (true) {
    if (cin >> x) {
      total += x;
    } else {
      if (cin.eof()) break;
      cin.clear();
      cin.ignore();
    }
  }
  cout << "Total is: " << total << endl;
}
```

The I/O Manipulators

Sometimes we want to change the behaviour of our streams. We can do so with the I/O manipulators available in the `<iomanip>` header.

To modify a stream with an I/O manipulator we use the given I/O manipulator as an operand to the appropriate I/O operator for the stream we want to modify.

Printing numbers in hex

For example we can use `std::hex` to specify that we want `cout` to print numbers in hexadecimal.

```
std::cout << std::hex << 237 << std::endl;
std::cout << 1997 << std::endl; // still in hex!
std::cout << std::dec;
std::cout << 777 << std::endl;
```

Printing numbers in hex

For example we can use `std::hex` to specify that we want `cout` to print numbers in hexadecimal.

```
std::cout << std::hex << 237 << std::endl;
std::cout << 1997 << std::endl; // still in hex!
std::cout << std::dec;
std::cout << 777 << std::endl;
```

Note: sending an I/O manipulator to a stream mutates that stream itself. Once std::hex has been sent to an output stream all integers it outputs will be hexadecimal. In order to switch back to printing decimal integers we have to send the opposing `std::dec` manipulator.

Reading whitespace

By default the input operator skips over leading whitespace, even when reading individual characters.

```
int main() {
  char c;
  while (std::cin >> c) std::cout << c << c;
  std::cout << std::endl;
}
```

This program will print out every (non-whitespace) character the user inputs twice, and end with a newline. What if we also wanted to print out the whitespace characters the user types?

The `std::noskipws` and `std::skipws` manipulators

The input manipulators `std::noskipws` and `std::skipws` modify the input stream they're sent to such that it doesn't/does skip over leading whitespace respectively.

```
int main() {
  char c;
  std::cin >> std::noskipws;
  while (std::cin >> c) std::cout << c << c;
  std::cout << std::endl;
}
```

Be careful though! Make sure to send `std::skipws` to the output stream after you're finished if you want it to revert back to skipping leading whitespace. If you're not skipping whitespace then even a few spaces before an integer can mean a read of an integer fails!

Practice Problem

Write a C++ program that mimics the `wc` utility when run without any command line arguments.

Table of Contents

Managing Memory

The memory model we learned in C is still relevant in C++, and so are the liftimes of our data.

Now however we will use the type aware operators `new` and `delete` for allocating and freeing heap memory.

The `new` operator

Operator `new` in C++ takes one operand which is the type you'd like to allocate on the heap and evaluates to the pointer to that newly heap-allocated data.

```
// Allocates an int on the heap
int *p = new int;
// Allocates an array of 100 ints on the heap
int *arr = new int[100];
```

New is type aware so you no longer specify the number of bytes you'd like, you just specify the type you'd like. The type you'd like to have allocated could be an array.

The `delete` operator

Operator `delete` in C++ takes one operand which is a pointer to heap-allocated memory you'd like to free and frees that heap-allocated data.

```
// Allocates an int on the heap
int *p = new int;
// Allocates an array of 100 ints on the heap
int *arr = new int[100];
delete p; // Right
delete arr; // WRONG!!!
```

Warning: There is an array form of operator `delete` that is required to be used when freeing an array that was allocated.

Array form operator `delete`

```
// Allocates an int on the heap
int *p = new int;
// Allocates an array of 100 ints on the heap
int *arr = new int[100];
delete p; // Right
delete[] arr; // Right
```

Make sure whenever freeing an array that was allocated on the heap that you use the array form of `delete`.

Table of Contents

Multiple functions — the problem

```
int max(int a, int b) {
  return a > b ? a : b;
}
```

In C if we want a version of the above function that works for `floats` we'd have to write a new function, but we couldn't also name it `max`. We'd have to make sure both functions are named different, so something like:

```
int maxInt(int a, int b) {
  return a > b ? a : b;
}
int maxFloat(float a, float b) {
  return a > b ? a : b;
}
```

Function overloading

C does not allow two functions named the same thing as it views it as a redefinition. C++ allows two functions to have the same name so long as they differ in number or ordered type of parameters.

```
int max(int a, int b) {
  return a > b ? a : b;
}
float max(float a, float b) {
  return a > b ? a : b;
}
```

The above is valid in C++ but not C. Giving new definitions to a function for new types is called *overloading*.

Operator overloading

In C++ we can overload operators as well as functions! That means we can give our own meaning to operators in C++ for types for which they're not already defined.

Which types would not have operators defined for them already? Certainly any types that we, the programmer, create!

Vector in 3D Space

Let's define a type, `Vec3D` to represent a vector in a 3D space over the integers and give it some useful operations. We'd like to be able to

- Scale a `Vec3D` by an integer scalar
- Add two `Vec3D`s together
- Multiply two `Vec3D`s together to get their dot product

So our type will look something like this

```
struct Vec3D {
  int x, y, z;
};
```

How do we overload operators to work with it?

Vec3D behaviour

Ideally we want to be able to write code like the following with our Vec3D type.

```
int main() {
  Vec3D v;
  // Note just Vec3D is fine in C++!
  v.x = 1; v.y = 2; v.z = 3;
  Vec3D w = v*2;
  Vec3D u = v + w;
  cout << w.x << " " << w.y << " " << w.z << endl;
  // 2 4 6
  cout << u.x << " " << u.y << " " << u.z << endl;
  // 3 6 9
  cout << v*w << endl; // 28
}
```

Operating overloading syntax

The syntax for operator overloading is simple — we define a function to represent the code for the operator.

The syntax is the exact same as it is for any other function, the only difference is the name of the function.

When overloading an operator you define a function whose name is `operator` followed by the operator you'd like to overload. For example:

```
Vec3D operator*(Vec3D v, int scalar);
```

Scalar multiplication and `Vec3D`

```
Vec3D operator*(Vec3D v, int scalar) {
  Vec3D ret;
  ret.x = v.x*scalar;
  ret.y = v.y*scalar;
  ret.z = v.z*scalar;
  return ret;
}

int main() {
  Vec3D v;
  v.x = 1; v.y = 2; v.z = 3;
  Vec3D w = v*2; // Works
  Vec3D u = 3*v; // Doesn't work!
}
```

Operator overloading and parameter order

When overloading operators the order of the parameters dictates the order of the operands. We only defined scalar multiplication where a `Vec3D` is the left-hand operand and the int scalar is the right-hand operand.

To allow the user to multiply an int by a `Vec3D` we need to overload the operator again.

Scalar multiplication and `Vec3D`

```
Vec3D operator*(Vec3D v, int scalar) {
  Vec3D ret;
  ret.x = v.x*scalar;
  ret.y = v.y*scalar;
  ret.z = v.z*scalar;
  return ret;
}
Vec3D operator*(int scalar, Vec3D v) {
  return v*scalar; // Reuse code we already wrote
}
int main() {
  Vec3D v;
  v.x = 1; v.y = 2; v.z = 3;
  Vec3D w = v*2; // Works
  Vec3D u = 3*v; // Works now!
}
```

Adding `Vec3Ds`

```
Vec3D operator+(Vec3D lhs, Vec3D rhs) {
  Vec3D ret;
  ret.x = lhs.x+rhs.x;
  ret.y = lhs.y+rhs.y;
  ret.z = lhs.z+rhs.z;
  return ret;
}
```

This function allows for us to add two `Vec3Ds` together.

Dot product of `Vec3Ds`

```
int operator*(Vec3D lhs, Vec3D rhs) {
  return lhs.x*rhs.x + lhs.y*rhs.y + lhs.z*rhs.z;
}
```

This function allows for us to multiply two `Vec3Ds` together to get a dot product.

Overloading the input operator

If we can overload operators, then can we overload the input and output operators to tell C++ how to read in/write out one of our types?

Overloading the input operator

If we can overload operators, then can we overload the input and output operators to tell C++ how to read in/write out one of our types?

Yes!

Overloading the input operator

If we can overload operators, then can we overload the input and output operators to tell C++ how to read in/write out one of our types?

Yes!

Let's try the input operator — what would the function signature look like?

Overloading the input operator

If we can overload operators, then can we overload the input and output operators to tell C++ how to read in/write out one of our types?

Yes!

Let's try the input operator — what would the function signature look like?

```
??? operator>>(??? in, Vec3D v);
```

Overloading the input operator — an observation

We don't know what type `cin` (or other input streams) are — but there's still an interesting observation to be made here.

We want this function to mutate the `Vec3D` that we give it. We've observed the input operator mutating other variables, i.e.

```
int x;
cin >> x; // x is changed
```

But how could a function we call change our local variable without being given its address? That is, why don't we have to do:

```
cin >> &x;
```