C++: The Basics

Dr. Frazer K. Noble
School of Engineering and Advanced Technology
Massey University
Auckland
New Zealand

March 23, 2017

Introduction

The aim of this document is to introduce C++, without going into too much detail. Therefore, we informally present C++ and the basic mechanisms for organising code into a program.

The Basics

C++ is a compiled language. Its source code is processed by a compiler, which produces object files, which are linked together, generating an executable.

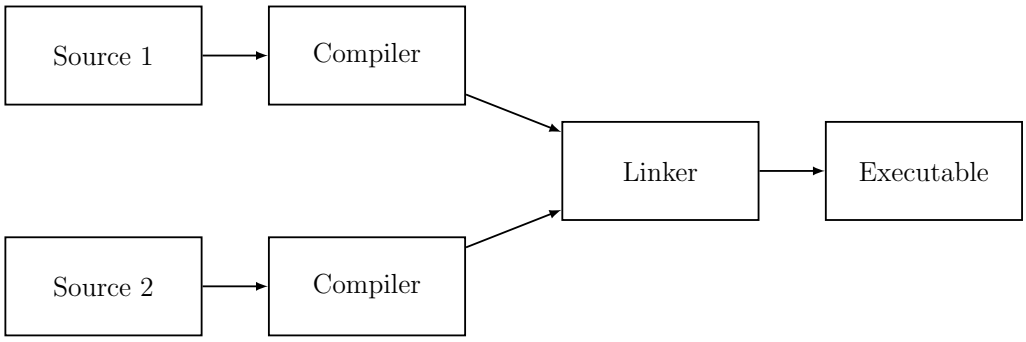

Figure 1: C++ compilation and linking process.

On Windows, an executable has the .exe suffix. Also, an executable generated for Windows is not transferable to Linux or Mac (and vice versa).

With respect to source code, the ISO C++ standard defines to kinds of entities:

1. Core language features. These are built-in types, e.g. **int**, and loops, e.g. **for**.

2. Standard-library features. These are containers, e.g. vector, and input/output (I/O) operations, e.g. cout.

The standard-library is a collection of C++ code, which is provided by every C++ implementation.

C++ is a statically typed language; that is, the type of every entity must be known to the compiler.

Hello World!

The minimal C++ program is

```
int main() {} //The minimal C++ program.
```

This defines a function called main, which has no parameters and does nothing.

The curly braces, {} express grouping. Here, they indicate the start and end of the main() function. The double slash, // begins a comment that extends to the end of the line. A comment is for the user; the compiler ignores it.

Every C++ program must have exactly one main() function. The program starts by executing that function. The **int** value returned by main(), if any, is the program's return value to the "system". Typically, a non-zero value from main() indicates failure.

Here is a program that produces some output:

```
#inlcude <iostream>

int main(int argc, char* argv[]) {
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

The line **#include**<iostream> instructs the compiler to *include* the declarations of the standard stream I/O facilities as found in iostream. Without these, the expression

```
std::cout << "Hello World!" << std::endl;
```

would make no sense. The operator << ("put to") writes its second argument onto its first. In this case, the string literal "Hello World! is onto the standard output stream std::cout.

The std:: specifies that the name cout is to be found in the standard-library namespace.

In general, all executable code is placed in functions and called directly, or indirectly, from main().

Types, Variables, and Arithmetic

Every name and expression has a type. For example, the declaration

```
int inch;
```

specifies that inch is of type **int** (an integer variable).

A *declaration* is a statement that introduces a name into the program. It specifies a type for the named entity:

- A *type* defines a set of possible values and set of operations.
- An *object* is some memory that holds a value of some type.
- A *value* is a set of bits interpreted according to a type.
- A *variable* is a named object

C++ offers a variety of fundamental types. For example,

```
bool //Boolean, possible values are true and false.
char //Character, e.g. 'a', 'b', 'c'.
int //Integer value, e.g. 1, 2, 3.
double //Double-precision, floating-point value, e.g.
    3.14, 2.3, 1.4.
```

Arithmetic operators can be used for appropriate combinations of these types:

```
x+y //plus.
x-y //minus.
x*y //multiplication.
x/y //division.
x%y //remainder (modulus) for integers.
```

So can the comparison operators:

```
x==y //equal.
x!=y //not equal.
x>y //greater than.
x<y //less than.
x>=y //greater than or equal.
x<=y //less than or equal.
```

Note, = is the assignment operator; == is the equality operator.

C++ offers a variety of notations for expressing initialisations, such as =, and a universal form based on curly-brace-delimited initialiser lists:

```
double d = 1.3; //Initialise d with 1.3.
double e {2.6}; //Initialise e with 2.6.

vector<int> v {1,2,3,4,5}; //Initialise a vector of
   int's.
```

The = form is traditional and dates back to C, but if in doubt, use the general {}-form.

When initialising a variable, you don't need to explicitly state its type when it can deduced from the initialiser:

```
auto i {1}; \\Integer value
auto d {2.0}; \\Double value
auto v {1,2,3,4,5}; \\Vector of int's
```

auto is used when we don't have a reason to mention type and is helpful in avoiding redundancy.

Constants

C++ supports two notions of immutability:

const: meaning, roughly, "I won't change this value" and is enforced by the compiler.

constexpr: meaning, roughly, "to be evaluated at compile tile".

For example:

```
const int kNum = 1; //kNum is a named constant
int var = 1; //var is not a constant
constexpr double max1 = 1.4*square(kNum); //OK if
    square is a constant expression
constexpr double max2 = 1.4*square(var); //Not OK,
    var is not a constant
const double max3 = 1.4*square(var); //OK, may be
    evaluated at run time.
```

For a function to be usable in a *constant expression*, it must be defined constexpr. For example:

```
constexpr doube square(const int &var) {return var*
    var;}
```

Tests and Loops

C++ provides a conventional set of statements for expressing selection and looping. For example, here is a function that checks if a number is equal to the value 2:

```
bool check() {

  std::cout << "Type a number: ";

  char num {};
  std::cin >> num;

  if (num == '2') {
    return true;
  } else {
    return false;
  }
}
```

To match the << ("put to") operator, the >> ("get from") operator is used for input; std::cin is the standard input stream.

Pointers, Arrays, and Loops

An array of elements of type **char** can be declared like this:

```
char v[6]; //array of six characters.
```

Similarly, a pointer can be declared like this:

```
char* p; //pointer to a character.
```

In declarations, [ ] means "array of", and * means "pointer to". All arrays have 0 as their lower bound, so v has six elements, v[0] to v[5]. A pointer can hold the address of an object of the appropriate type:

```
char* p = &v[3];
char x = *p;
```

In this expression, * means "contents of" and & means "address of". We can express this graphically.

Consider copying ten elements from one array to another:

```
void copy() {
  int v1[10] = {0,1,2,3,4,5,6,7,8,9};
  int v2[10];

  for (auto i = 0; i < 10; i++) { //Copy elements of
     v1 to v2
    v2[i] = v1[i];
```

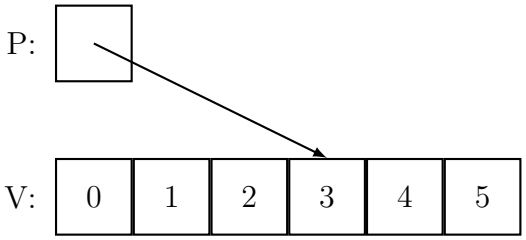

Figure 2: C++ pointers and arrays.

```
    }
}
```

The **for** loop can be read as "set i to zero; while i is less than 10, copy v1's i'th element to v2. When applied to an integer variable, the increment operator, ++, adds 1. An alternative is to use the range-for-statement, **for** (**auto** x : v) {}, which traverses sequence v and can be read as "for each element in v, do something".

User-Defined Types

Structures

The first step in building a new type is often to organise the elements it needs into a data structure, a **struct**:

```
stuct Vector {
  int size; //Number of elements
  double* element; //Pointer to elements
};
```

A variable of type Vector can be defined like this:

```
Vector v;
```

As it is, v isn't very useful; we need to give v some elements to point to. For example, we can construct a Vector like this:

```
void vector_init(Vector& v, int s) {
  v.elem = new double[s]; //allocate an array of s
      doubles
  v.sz = s
}
```

That is, v's elem memeber gets a pointer produced by the **new** operator and v's sz member gets the number of elements. The & in Vector& indicates

that we pass v as a non-**const** reference; that way, vector_init can modify it.

A simple use of Vector is as follows:

```
double read_and_sum(int s) {

  Vector v;
  vector_init(v, s);
  for(int i=0; i<s; i++) {
    std::cin >> v.elem[i];
  }

  double sum = 0;
  for(int i=0; i<s; i++) {
    sum += v.elem[i];
  }

  return sum;

}
```

Classes

A *class* separates a data type's interface (used by all) from its implementation (which has access to hidden, or inaccessible, data). The class is defined to have a set of *memebers*, which can be data, function, or type members. The interface is defined by the **public** members of a class, and **private** members are only accessible through that interface. For example,

```
class Vector{
public:
  Vector(int s) :elem{new double[s]}, sz{s} {} //
      Construct a Vector.
  double& operator[](int i){ return elem[i]; } //
      Element access: subscripting
  int size() const { return sz; }
private:
  double *elem;
  int sz;
};
```

Given that, we can define a variable of our new type Vector:

```
Vector v[6];
```

We can illustrate a Vector graphically:

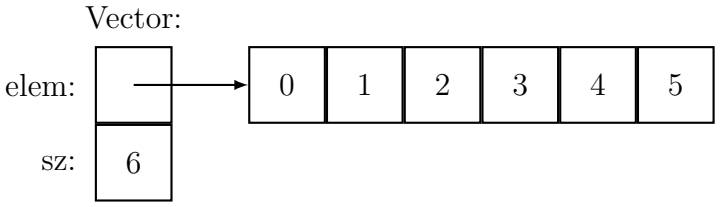

Figure 3: Illustration of a Vector.

Basically, the Vector object is a "handle" containing a pointer to the elements (elem) plus the number of elements (sz). Here, the access to elem and sz is only through the **public** members, i.e. Vector(), **operator**[](), and size().

Conclusion

In this document, we have introduced C++. We have described the basics of compilation, presented simple programs, described C++'s types, variables, and arithmetic; described simple tests and loops, and pointers and arrays; and introduced user-defined types, such as structures and classes.

Bibliography

[1] B. Stroustrup. *The C++ Programming Language: 4th Edition*. Always learning. Addison-Wesley, 2013.