IN2029: Programming in C++

Session 1 – Introduction to C++

Ross Paterson

Department of Computer Science
City, University of London

Autumn term, 2018

This module: more programming, in C++

C++ has evolved over three decades, but is still widely used.
Assuming that you are currently a reasonably skillful Java programmer, by the end of this course you should be able to

- read and modify substantial well-written C++ programs
- create classes and small programs in C++ that are correct, robust, clear and reusable
- use the standard libraries effectively

The language is much too large to cover in one term, so we shall focus on the most useful features for writing application code.

This session: Getting started with C++

This session introduces the philosophy of C++, and some simple non-OO programs.
We will touch on the following features of C++:

- Program structure
- Standard libraries: I/O and strings
- Operator overloading
- Class types

All will be explored in greater detail later.

A bit of language history

1960 Algol 60: block structure, static typing

1967 Simula: Algol plus object-orientation (for simulation)

1970 C: statically typed, with low-level features

1972 Smalltalk: object-orientation (for graphical interfaces), no static types

1985 C++: C plus object-oriented features

1990 *The Annotated C++ Reference Manual*: namespaces, templates, exceptions

1995 Java: C++ greatly simplified (Sun)

1998 C++-98 (ISO standard): standard template library

2000 C♯: another simplified C++ (Microsoft)

2011 C++11: many new features (revised in 2014 and 2017)

Java design criteria

- object orientation
- (moderate) simplicity (fewer variant ways of doing things)
- robustness and security (type-safe, automatic memory allocation)
- architecture-neutral (fairly high level)
- precisely specified, e.g. `int` is always 32 bits
- syntax based on C++

C++ design criteria

- support a variety of programming styles, including object oriented (give the programmer more choices)
- powerful (give the programmer more control)
- enable efficient implementation (shift some implementation concerns to the programmer)
- implementation-dependent specification, e.g. `int` is machine word size
- extension of C (machine-level access)

C++ as a composite language

C++ is a very large language, with many layers that have accumulated over time.
In his book *Effective C++*, Scott Meyers suggests thinking of C++ as a "federation of related languages" with different programming styles:

- C: a low-level language with built-in data types and control structures. Often C features coexist with newer, cleaner versions.
- Object-Oriented C++: classes, inheritance, dynamic binding, etc. (But unlike in Java, one can often avoid these.)
- Template C++: classes and functions can be parameterized by types.
- The Standard Template Library: a library of containers, iterators and algorithms using template classes, but not inheritance.

For the first five weeks, we shall focus on the first and last of these, and introduce the second later.

A small C++ program

```
#include <iostream>

int main() {
    std::cout << "Hello world!\n";
    return 0;
}
```

- A typical program will begin with several **`#include`** lines, referencing system headers like **`<iostream>`**, which contain collections of related definitions (in this case for I/O streams).
- This **`#include`** line brings into scope several names, including **`std::cout`**, the standard output stream. Here **`std`** is the standard namespace, which is attached to all names defined in the standard libraries, e.g. **`std::string`**, **`std::vector`**, etc.

Using the standard namespace

We can avoid putting **std::** in front of every standard name by placing a single **using** declaration at the top, after the **#include** lines:

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello world!\n";
    return 0;
}
```

(Some style guides prefer to put **std::** everywhere.)
We won't be using any namespaces other than **std**.

Functions

In C++ (as in C), one can define functions independent of any class:

```
int square(int n) {
    return n*n;
}
```

A function called **main** is used as the entry point of the program:

```
int main() {
    // whatever you like here
    return 0;
}
```

The return value is reported to the system when the program exits: 0 means success; anything else means something went wrong.

Similarities with Java

Many things in the C part of C++ are much the same as in Java:

- primitive types `int`, `double`, etc
- declarations, expressions, comments
- semicolons to end statements and braces for grouping
- `if`, `for`, `while`, `return`.

For example, this means the same in both languages:

```
int factorial(int n) {
    int prod = 1;
    for (int i = 1; i <= n; ++i)
        prod = prod * i;
    return prod;
}
```

(but in Java this must be a method inside a class)

Text output

This statement writes a string to the standard output (the console):

```
cout << "Hello world!\n";
```

(like `System.out.println("Hello world!");` in Java)

- The **<iostream>** header defines three standard streams:

std::cin	standard input (cf. Java's `System.in`)
std::cout	standard output (cf. Java's `System.out`)
std::cerr	error output (cf. Java's `System.err`)

- The << operator, when applied to an output stream and a string, writes the string to the stream.
- When applied to integers, it performs a left shift (as in Java): the << operator is overloaded.

Outputting several things

```
#include <iostream>

using namespace std;

int main() {
    int i = 3;
    cout << "The value is " << i << '\n';
    return 0;
}
```

The << operator associates to the left, and returns the stream, so the above line is equivalent to

```
    ((cout << "The value is ") << i) << '\n';
```

It is also overloaded for `int` and `char`.

Ending output lines with `endl`

Some authors use `endl` instead of `'\n'`:

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello world!" << endl;
    return 0;
}
```

but this is slightly different. The `endl` manipulator outputs `'\n'` and then flushes the output, so the output line is equivalent to

```
    cout << "Hello world!" << '\n' << flush;
```

But frequent flushing may be costly.

Text input

```
#include <iostream>

using namespace std;

int main() {
    int i;
    cout << "Type a number: ";
    cin >> i;
    cout << i << " times 3 is " << (i*3) << '\n';
    return 0;
}
```

The >> operator reads from an input stream. It is overloaded in a similar way to the << operator.

Using strings

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

using namespace std;

int main() {
    cout << "Please enter your first name: ";
    string name;
    cin >> name;

    cout << "Hello, " << name << "!\n";
    cout << "Your name has " << name.size() <<
        " letters.\n";
    return 0;
}
```

Strings details

```
#include <string>
```

includes a standard header defining a type **std::string**.

```
    string name;
```

declares a variable **name** of type **string** and initializes it to an empty string (different from Java).

```
    cin >> name;
```

reads one word from standard input into **name**.

```
    cout << name.size();
```

A **string** is an object, with a member function (method in Java terms) **size()** that returns a number, the length of the string.

Properties of types

- Like Java, C++ has primitive types, like **`int`** and **`double`**, and also class types, like **`string`**.
- In Java, a variable of class type holds a reference to an object, which has to be created with `new`.
- In C++, a variable of class type holds an object, which is initialized in some default way if you don't give an initial value.

```
string name; // initialized as empty
string another = "Fred";
string name = another; // copies the string
```

- In contrast, local variables of primitive types are not initialized to default values.

Summary

Much of basic C++ is similar to Java, but we've already noticed some differences:

- Different syntax for accessing libraries (`#include` and namespaces)
- Functions can be defined independent of classes.
- More extensive use of overloading
- **Variables of class type contain objects, not references.** (We'll explore the consequences of this over the following weeks.)

Next week, we'll look at using sequential containers (`vector`, `list`, `deque`, etc).