Programming C and C++
Lectures 11&12: C++ for Java and C programmers

Alan Mycroft
Computer Laboratory, University of Cambridge
Michaelmas Term 2018–2019

Notes based, with thanks, on slides due to Alastair Beresford and Andrew Moore

Aims of C++
To quote Bjarne Stroustrup:
“C++ is a general-purpose programming language with a bias towards systems programming that:
▶ is a better C
▶ supports data abstraction
▶ supports object-oriented programming
▶ supports generic programming.”

Alternatively: C++ is “an (almost upwards-compatible) extension of C with support for: classes and objects (including multiple inheritance), call-by-reference, operator overloading, exceptions and templates (a richer form of generics)”.

Much is familiar from Java, but with many subtle differences.

How to follow these two lectures
▶ These slides try capture the core enhancements of C++, so that afterwards you will be able to read C++ code, and tentatively modify it.
▶ But C++ is a very complex language, so these slides are incomplete, even if they uncomfortably large.
▶ For exam purposes the fine details don’t matter, it’s more important to get the big picture, which I’ll try to emphasise in lectures.

The creator of C++, Bjarne Stroustrup, has various entertaining and educational articles on his web page: www.stroustrup.com

What we’ll cover
▶ Differences between C and C++
▶ References versus pointers
▶ Overloading functions and operators
▶ Objects in C++; Classes and structs; Constructors; Virtual functions
▶ Multiple inheritance; Virtual base classes; Casting
▶ Exceptions
▶ Templates and meta-programming

Slides marked ‘[further information]’ will not be lectured.
▶ For exam purposes, focus on ‘big-picture’ novelties and differences between features of C++ and those in C and Java.
▶ For coding, sorry but compilers insist you get it exactly right.

Should I program my application in C or C++?
Or both or neither?
▶ One aim of these lectures is to help you decide.
▶ C and C++ both have very good run-time performance
▶ C++ has more facilities, but note Bjarne Stroustrup’s quote: “C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.”
▶ Even if C++ is a superset of C then mixing code is risky, e.g.
  ▶ you don’t want two conflicting IO libraries being active, e.g.
  ▶ you often program using different metaphors in C and C++
  ▶ C functions may not expect an exception to bypass their tidy-up code
  ▶ Using C-coded stand-alone libraries in C++ is fine.
▶ C++ vs. Java? Speed vs. safety? More vs. fewer features? Java is trying to follow C++ (and C#) by having value types (objects/structs as values not just references).

 Decide C or C++ at the start of a project.

C++ fundamental types [further information]
▶ C++ has all the fundamental types C has
  ▶ character literals (e.g. ‘a’) are now of type char
▶ In addition, C++ defines a new fundamental type, bool
▶ A bool has two values: true and false
▶ When cast to an integer, true→1 and false→0
▶ When casting from an integer, non-zero values become true and false otherwise

C++ enumeration [further information]
▶ Unlike C, C++ enumerations define a new type; for example
  enum flag is_keyword=1, is_static=2, is_extern=4, ...
▶ When defining storage for an instance of an enumeration, you use its name; for example: flag f = is_keyword
▶ Implicit type conversion is not allowed:
  f = 5; //wrong  f = flag(5); //right
▶ The maximum valid value of an enumeration is the enumeration’s largest value rounded up to the nearest larger binary power minus one
▶ The minimum valid value of an enumeration with no negative values is zero
▶ The minimum valid value of an enumeration with negative values is the nearest least negative binary power
References

C++ supports references, which provide an alternative name (alias) for a variable

- Generally used for specifying parameters to functions and return values as well as overloaded operators (more later)
- A reference is declared with the & operator; for example:
  ```
  int i[] = {1,3}; int &refi = i[0]; int &pi = &i[0];
  ```
- A reference must be initialised when it is declared
- The connection between a reference and what it refers to cannot be changed after initialisation; for example:
  ```
  refi++; // increments value referenced to 2
  pi++; // increments the pointer to &i[1]
  ```

Think of reference types as pointer types with implicit * at every use.

Overloaded functions

- Just like Java we can define two functions with the same name, but varying in argument types (for good style functions doing different things should have different names).
- Type conversion is used to find the “best” match
- A best match may not always be possible:

  ```
  void f(double);
  void f(long);
  void test() {
    f(1L); // f(long)
    f(1.0); // f(double)
    f(1); // f(long(1)) or f(double(1))?
  }
  ```

- Can also overload built-in operators, such as assignment and equality.

Applies both to top-level functions and member functions (methods).

Default function arguments

- A function can have default arguments; for example:
  ```
  double log(double v, double base=10.0);
  ```
- A non-default argument cannot come after a default; for example:
  ```
  double log(double base=10.0, double v); // wrong
  ```
- A declaration does not need to name the variable; for example:
  ```
  double log(double base, double v);
  ```
- Be careful of the lexical interaction between * and =; for example:
  ```
  int i[] = {1,3}; int &refi = i[0]; const int max_size = 100;
  char s[max_size];
  int top = 0;
  void push(char c) {
    if (top < max_size) {
      s[top] = c;
      top = top + 1;
    }
  }
  ```

Namespaces

Related data can be grouped together in a `namespace`. Can use :: and `using` to access components. Think Java packages.

```
namespace Stack { // header file
  void push(char);  
  char pop();
}
```

```
namespace Stack { // implementation
  void f() {
    // usage
    char pop();
    Stack::push('c');
  }
}
```

Example

```
namespace Module1 { int x;}
namespace Module2 {
  inline int sq(int i) { return i*i; }
  inline int halve(int i) { return i/2; }
  }
using namespace Module1;  // "import" everything
int main() {
  using Module2::halve;  // "import" the halve function
  x = halve(x);
  sq(x); // Wrong
}
```

Using namespaces [further information]

- A namespace is a scope and expresses logical program structure
- It provides a way of collecting together related pieces of code
- A namespace without a name limits the scope of variables, functions and classes within it to the local execution unit
- The same namespace can be declared in several source files
- The global function `main()` cannot be inside a namespace
- The use of a variable or function name from a different namespace must be qualified with the appropriate `namespace(s)`
  - The keyword `using` allows this qualification to be stated once, thereby shortening names
  - Can also be used to generate a hybrid namespace
- `typedef` can be used: `typedef Some::Thing thing;`
- A namespace can be defined more than once:
  - Allows, for example, internal and external library definitions
Linking C and C++ code

- The directive `extern "C"` specifies that the following declaration or definition should be linked as C, not C++, code:

  ```
  extern "C" int f();
  ```

- Multiple declarations and definitions can be grouped in curly brackets:

  ```
  extern "C" {
  int globalvar; // definition
  int f();
  void g(int);
  }
  ```

Why do we need this?

- ‘Name mangling’ for overloaded functions. A C compiler typically generates linker symbol ‘_f’ for f above, but a C++ compiler typically generates ‘_Zifv’.
- Function calling sequences may also differ (e.g. for exceptions).

Big Picture

So far we’ve only done minor things.

- We’ve seen C++ extensions to C. But, apart from reference types, nothing really new has appeared that’s beyond Java concepts.
- Now for classes and objects, which look the same, but aren’t . . .

Classes and objects: big differences from Java

- Values of class types are not references to objects, but the objects themselves. So we access members with C-style ‘.’ (but using ‘->’ is more convenient when you have pointers to objects).
- We can create an object of class C, either by:
  - on the stack (or globally) by declaring a variable: `C x;`
  - on the heap: `new C()` (returns a pointer to C)
- Member functions (methods) by default are statically resolved. For Java-like code declare them `virtual`.
- Member functions can be declared inside a class but defined outside it using `:\`.
- C++ uses `new` to allocate and `delete` to de-allocate. There is no garbage collector – users must de-allocate heap objects themselves.

Constructors and destructors

- A default constructor is a function with no arguments (or only default arguments).
- If no constructor is specified, the compiler will generate one.
- The programmer can specify one or more constructors.
- Only one constructor is called when an object is created.
- There can only be one destructor.
  - This is called when a stack-allocated object goes out of scope or when a heap-allocated object is deallocated with `delete`; this also occurs for stack-allocated objects deallocated during exception handling (more later).
  - Stack-allocated objects with destructors are a useful way to release resources on scope exit (similar effect as Java try-finally) – ‘RAII: Resource Allocation is Initialisation’.
  - Make destructors virtual if class has subtypes or supertypes.

Copy constructor

- A new class instance can be defined by initialisation; for example:

  ```
  Complex c(1,2); // not this C++ initialiser syntax
  ```

- In this case, the new class is initialised with copies of all the existing class’s non-static member variables; no constructor is called.
- If this behaviour is undesirable (e.g. consider a class with a pointer as a member variable) define an own copy constructor:

  ```
  Complex::Complex(const Complex&); ...
  ```

- If a copy constructor is not wanted, make the copy constructor a private member function, or in C++11 use `delete`.
  - Note that assignment, e.g. `d = c;` differs from initialisation and does not use the copy constructor – see next slide.
Assignment operator

- By default a class is copied on assignment by over-writing all non-static member variables; for example:

```cpp
1 Complex c(), d(1.0, 2.3);
2 c = d; // assignment
```

- This behaviour may also not be desirable (e.g. you might want to tidy up the object being over-written).

- The assignment operator (operator=) can be defined explicitly:

```cpp
1 Complex& Complex::operator=(const Complex& c) {
2    ...
3    return *this;
4 }
```

- Note the result type of assignment, and the reference-type parameter (passing the argument by value would cause a copy constructor to be used before doing the assignment, and also be slower).

Arrays and the free store

- An array of class objects can be defined if a class has a default constructor

- C++ has a new operator to place items on the heap:

```cpp
1 Complex* c = new Complex(3.4);
```

- Items on the heap exist until they are explicitly deleted:

```cpp
1 delete c;
```

- Since C++ (like C) doesn’t distinguish between a pointer to a single object and a pointer to an the first element of an array of objects, array deletion needs different syntax:

```cpp
1 Complex* c = new Complex[5];
2    ...
3 delete[] c; //Cannot use "delete" here
```

- When an object is deleted, the object destructor is invoked

- When an array is deleted, the object destructor is invoked on each element

Operators

- C++ allows the programmer to overload the built-in operators

- For example, a new test for equality:

```cpp
1 bool operator==(Complex a, Complex b) {
2    return a.real()==b.real() && a.imag()==b.imag();
3    // presume real() is an accessor for field 're', etc.
4 }
```

- An operator can be defined or declared within the body of a class, and in this case one fewer argument is required; for example:

```cpp
1 bool Complex::operator==(Complex b) {
2    return re==b.real() && im==b.imag();
3 }
```

- Almost all operators can be overloaded

The 'this' pointer

- If an operator is defined in the body of a class, it may need to return a reference to the current object

  - The keyword this can be used

```cpp
1 Complex& Complex::operator+=(Complex b) {
2    re += b.real();
3    this->im += b.imag();
4    return *this;
5 }
```

- In C (or assembler) terms this is an implicit argument to a method when seen as a function.

Constant member functions

- Member functions can be declared const

- Prevents object members being modified by the function:

```cpp
1 double Complex::real() const {
2    return re;
3  }
```

- In effect it gives type const Complex* this instead of Complex* this to the (implicit) parameter 'this'.

- Helpful to both programmer (maintenance) and compiler (efficiency).

Exercises

1. Write an implementation of a class LinkList which stores zero or more positive integers internally as a linked list on the heap. The class should provide appropriate constructors and destructors and a method pop() to remove items from the head of the list. The method pop() should return -1 if there are no remaining items. Your implementation should override the copy constructor and assignment operator to copy the linked-list structure between class instances. You might like to test your implementation with the following:

```cpp
1 int main() {
2    int test[] = {1, 2, 3, 4, 5};
3    LinkList l1(test+1, 4), l2(test, 5);
4    LinkList l3=l2, l4;
5    l1.pop();
6    printf("%d %d %d\n", l1.pop(), l3.pop(), l4.pop());
7    return 0;
8 }
```

  Hint: heap allocation & deallocation should occur exactly once!

Streams

- Overloaded operators also work with built-in types

- Overloading is used to define << (C++’s "printf"); for example:

```cpp
1 #include <iostream>
2 int main() {
3    int main() {
4        const char* s = "char array";
5        std::cout << s << std::endl;
6        std::cout.operator<<(std::endl);
7    }
8 }
```

  ```cpp
11    //Unexpected output: prints &s[0]
12    std::cout.operator<<(std::endl);
13    //Expected output: prints s
14    std::cout.operator<<(std::endl);
15 }
```

  ```cpp
20    return 0;
21 }
```

  ```cpp
22    Note std::cin, std::cout, std::cerr
```

Class instances as member variables

- A class can have an instance of another class as a member variable

- How can we pass arguments to the class constructor?

- New C++ syntax for constructors:

```cpp
1 class X {
2    Complex c;
3    Complex d;
4    X(double a, double b): c(a,b), d(b) {
5        ...
6    }
7 }
```

- This notation must be used to initialise const and reference members

- It can also be more efficient
Temporary objects

- Temporary objects are often created during execution
- A temporary which is not bound to a reference or named object exists only during evaluation of a full expression (BUGS BUGS BUGS!)
- Example: the C++ string class has a function c_str() which returns a pointer to a C representation of a string:

```c++
#include <iostream>
#include "example13.hh"

void print_speed(vehicle &v, bicycle &b) {
    std::cout << pv->maxSpeed() << std::endl;
}

int main() {
    bicycle b(true);
    print_speed(b,b); //prints "60 12"
}
```

Friends

- If, within a class C, the declaration friend class D; appears, then D is allowed to access the private and protected members of C.
- A (non-member) function can be declared friend to allow it to access the private and protected members of the enclosing class, e.g.

```c++
friend Vector operator*(const Matrix&, const Vector&);
```

Derived member function call

I.e. when we call a function overridden in a subclass.

- Default derived member function call semantics differ from Java:

```c++
virtual int maxSpeed() {return 60;}
```

Virtual functions

- Non-virtual member functions are called depending on the static type of the variable, pointer or reference
- Since a pointer to a derived class can be cast to a pointer to a base class, calls at base class do not see the overridden function.
- To get polymorphic behaviour, declare the function virtual in the superclass:

```c++
virtual int maxSpeed() {return 60;}
```

Virtual functions

- In general, for a virtual function, selecting the right function has to be run-time decision; for example:

```c++
virtual int maxSpeed() {return 60;}
```

Enabling virtual functions

- To enable virtual functions, the compiler generates a virtual function table or vtable
- A vtable contains a pointer to the correct function for each object instance
- The vtable is an example of indirection
- The vtable introduces run-time overhead (this is compulsory in Java; contemplate whether C++'s additional choice is good for efficiency or bad for being an additional source of bugs)
Abstract classes

- Just like Java except for syntax.
- Sometimes a base class is an un-implementable concept
- In this case we can create an abstract class:
  ```
  class shape {
    public:
    virtual void draw() = 0;
  }
  ```
- It is not possible to instantiate an abstract class:
  ```
  shape s; //Wrong
  ```
- A derived class can provide an implementation for some (or all) the abstract functions
- A derived class with no abstract functions can be instantiated
- C++ has no equivalent to Java ‘implements interface’.

Multiple instances of a base class

- With multiple inheritance, we can build:
  ```
  class A {};
  class B : public A {};
  class C : public A {};
  class D : public B, public C {};
  ```
- This means we have two instances of A even though we only have a single instance of D
- This is legal C++, but means all references to A must be stated explicitly
  ```
  D d;
  d.B::var=3;
  d.C::var=4;
  ```

Virtual base classes

- Alternatively, we can have a single instance of the base class
- Such a “virtual” base class is shared amongst all those deriving from it
  ```
  class Vehicle { int VIN; }
  class Boat : public virtual Vehicle { ... };
  class Car : public virtual Vehicle { ... };
  class JamesBondCar : public Boat, public Car { ... };
  ```
- Multiple inheritance is often regarded as problematic, and one of the reasons for Java creating interface.

Casts in C++

These need quite a bit of care, hence syntactic variants offering additional checks:

- classical C-style casts `type expr`, these do mainly the same as C. Take care casting between pointers when multiple inheritance or virtual bases are used; the compiler must be able to see the inheritance tree otherwise it might not compile the right operation (casting to a superclass might involve an addition or indirection, not just the no-op in Java).
- New C++ constructor syntax: `int('a')` or `G(expr)`.
- New C++ more descriptive forms: `dynamic_cast<T>(e)`, `static_cast<T>(e)`, `reinterpret_cast<T>(e)` and `const_cast<T>(e)`. The former is closest to Java object-reference casts, and generates code to do run-time tests of compatibility. Too much detail for this course.
- New C++ form: `typeid(e)` gives the type of e encoded as an object of `type_info` which is defined in standard header `<typeinfo>`.

Exceptions

Just like Java, but you normally throw an object value rather than an object reference:

- Some code (e.g. a library module) may detect an error but not know what to do about it; other code (e.g. a user module) may know how to handle it
- C++ provides exceptions to allow an error to be communicated
- In C++ terminology, one portion of code throws an exception; another portion catches it.
- If an exception is thrown, the call stack is unwound until a function is found which catches the exception
- If an exception is not caught, the program terminates

But there is no try-finally (use local variables having destructors).
exceptions are just normal values, matched by type
A class is often used to define a particular error type:
`class MyError {};`
An instance of this can then be thrown, caught and possibly re-thrown:

```cpp
void f() { ... throw MyError(); ... }
...
try {
  f();
} catch (MyError) {
  // handle error
  throw; // re-throw error
}
```

Exceptions and local variables
When an exception is thrown, the stack is unwound.
The destructors of any local variables are called as this process continues.
Therefore it is good C++ design practice to wrap any locks, open file handles, heap memory etc., inside stack-allocated object(s), with constructors doing allocation and destructors doing deallocation. This design pattern is analogous to Java's try-finally, and is often referred to as "RAII: Resource Allocation is Initialisation".

Templates – big-picture view (TL;DR)
Templates are like Java generics, but can have both type and value parameters:

```
template <class T, int max>
class Buffer { T[max] v; int n;};
```
You can also specify 'template specialisations', special cases for certain types (think compile-time pattern matching).
This gives lots of power (Turing-powerful) at compile time: 'meta-programming'.
Top-level functions can also be templated, with ML-style inference allowing template parameters to be omitted, given:

```
template<class T> void sort(T a[], const unsigned& len);
t int a[] = {2,1,3};
then sort(a,3) ≡ sort<int>(a,3)
The rest of the slides are just 'further information'.
```

Handling multiple errors
Multiple catch blocks can be used to catch different errors:

```
t try {
  f();
} catch (MyError x) {
  // handle error
  x.errorcode has the value 5
}
```
Every exception will be caught with `catch(...)`
Class hierarchies can be used to express exceptions:

```
struct ThisError : public SomeError {
  virtual void print() {
    std::cout << "This Error" << std::endl;
  }
};
struct ThatError : public SomeError {
  virtual void print() {
    std::cout << "That Error" << std::endl;
  }
};
```

The "thrown" type can carry information:

```
#include <iostream>
struct SomeError {virtual void print() = 0;};
struct ThisError : public SomeError {
  virtual void print() {
    std::cout << "This Error" << std::endl;
  }
};
struct ThatError : public SomeError {
  virtual void print() {
    std::cout << "That Error" << std::endl;
  }
};
int main() {
  try {
    f();
  } catch (MyError x) {
    // handle error
    x.errorcode has the value 5
  }
  return 0;
}
```

Exception handling in C++ continues
In one program, we might like to store a stack of `int`
In another, a stack of `NetworkReader` objects

`std::stack` allows us to store a single `generic` stack implementation for an unspecified type `T`
What functionality would we like a stack to have?

```
bool isEmpty();
void push(T item);
T pop();
...```
Many of these operations depend on the type `T`
Creating a stack template

- A class template is defined as:

```cpp
template<class T> class Stack {
    struct Item { //class with all public members
        T val;
        Item* next;
    }
    int head;
    char* intstack;
    void print() { std::cout << "Special" << std::endl; }
};
```

- Where class T can be any C++ type (e.g. int)
- When we wish to create an instance of a Stack (say to store ints) then we must specify the type of T in the declaration and definition of the class:

```cpp
Stack<int> intstack;
```

- We can then use the object as normal: `intstack.push(3);`
- So, how do we implement `Stack`?
  - Write T whenever you would normally use a concrete type
  - Java programmers: note Java forbids `List<int>` (generics cannot use primitive types).

```cpp
template<class T> class Stack {
    struct Item { //class with all public members
        T val;
        Item* next;
    }
    int head;
    void print() { std::cout << "Special" << std::endl; }
};
```

Default parameters

- Template parameters may be given default values

```cpp
template<class T, int i=128> struct Buffer{
    T buf[i];
};
```

```cpp
int main() {
    B<> b1;
    B<int> b2;
    b1.print(); //Special
    b2.print(); //General
}
```

Template richer details

- A template parameter can take an integer value instead of a type:

```cpp
template<int i> class Buf { int b[i]; ... }
```

- A template can take several parameters:

```cpp
template<class T, int i> class Buffer{T buf[i]; ... }
```

- A template can even use one template parameter in the definition of a subsequent parameter:

```cpp
template<class T, T val> class A { ... }
```

- A template class is not type checked until the template is instantiated:

```cpp
template<class T> class Stack { int stack.push(3); ... }
```

Specialisation

- The class T template parameter will accept any type T
- We can define a specialisation for a particular type as well (effectively type comparison by pattern-matching at compile time)

```cpp
# include "iostream"
class A {}; 

template<class T>
void sort(T a[], const unsigned int& len) { 
    T tmp;
    for(unsigned int i=0;i<len-1;i++)
        for(unsigned int j=0;j<len-1-i;j++)
            if (a[j] > a[j+1])
                tmp = a[j], a[j] = a[j+1], a[j+1] = tmp; 
    }
```

```cpp
# include "iostream"

template<class T>
void sort(T a[], const unsigned int& len) { 
    T tmp;
    for(unsigned int i=0;i<len-1;i++)
        if (a[j] > a[j+1])
            tmp = a[j], a[j] = a[j+1], a[j+1] = tmp; 
    }
```

Templated functions

- A top-level function definition can also be specified as a template; for example (think ML):

```cpp
template<class T> void sort(T a[], const unsigned int& len) { 
    T tmp;
    for(unsigned int i=0;i<len-1;i++)
        if (a[j] > a[j+1])
            tmp = a[j], a[j] = a[j+1], a[j+1] = tmp; 
    }
```
Overloading templated functions

- Templated functions can be overloaded with templated and non-templated functions
- Resolving an overloaded function call uses the "most specialised" function call
- If this is ambiguous, then an error is given, and the programmer must fix by:
  - being explicit with template parameters (e.g. `sort<int>(...)`
  - re-writing definitions of overloaded functions

Template specialisation enables meta-programming:

Meta-programming example

```cpp
#include <iostream>

template<unsigned int N> struct fact {
    static const int value = N * fact<N-1>::value;
    char v[value]; // just to prove the value is computed
};

template<> struct fact<0> {
    static const int value = 1;
};

struct fact<7> foo; // a struct containing char v[5040] and a const.
int main() {
    std::cout << sizeof(foo) << ", " << foo.value << std::endl;
}
```

Templates are a Turing-complete compile-time programming language!

Exercises

1. Provide an implementation for:
   ```cpp
template<class T> T Stack<T>::pop(); and
template<class T> Stack<T>::~Stack();
```

2. Provide an implementation for:
   ```cpp
   Stack(const Stack& s); and
   Stack& operator=(const Stack& s);
   ```

3. Using meta programming, write a templated class `prime`, which evaluates whether a literal integer constant (e.g. 7) is prime or not at compile time.

4. How can you be sure that your implementation of class `prime` has been evaluated at compile time?