

Programming in C and C++

Lectures 10–12: C++ for Java and C programmers

Alan Mycroft¹

Computer Laboratory, University of Cambridge

Michaelmas Term 2020–2021

¹Notes 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.

What we'll cover

- ▶ Differences between C and C++
- ▶ References versus pointers
- ▶ Overloaded functions and operators
- ▶ Objects in C++; Classes and structs; Destructors; Virtual functions
- ▶ Multiple inheritance; Virtual base classes; Casts
- ▶ Exceptions
- ▶ Templates and metaprogramming

- ▶ 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.

Reference sources

C++ is a big language with many subtleties. The current draft C++20 standard is 1841 pages (457 for the C++ language and 1152 for the C++ Standard Library; the grammar alone is 21 pages)!

<https://isocpp.org/> The ISO standard. Published standards cost money but draft standards are free online, e.g. draft C++20 on <https://isocpp.org/files/papers/N4860.pdf>

<https://cppreference.com> Wiki-book attempt to track standard.

<https://learncpp.com> More-chatty tutorial-style articles.

<https://www.stroustrup.com> Entertaining and educational articles by the creator of C++.

These are useful when wanting to know more about exactly how things (e.g. lambdas, overloading resolution) work, they are not necessary for exam purposes!

How to follow these three lectures

- ▶ These slides try capture the core features of C++, so that afterwards you will be able to read C++ code, and tentatively modify it. The Main ISO C++ versions are: C++98, C++11, C++20; we'll focus on core features—those in C++98.
- ▶ 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.

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,
 - ▶ 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++ Types [big picture]

C++ types are like C types, but additionally:

- ▶ character literals (e.g. `'a'`) are type `char` (but `int` in C)
- ▶ new type `bool` (values `true` and `false`)
- ▶ reference types: new type constructor `&`, so can have
`int x, *y, &z;`
- ▶ `enum` types are distinct (not just synonyms for integers)
- ▶ new type constructor `class` (generalising `struct` in C)
- ▶ names for `enum`, `class`, `struct` and `union` can be used directly as types (C needs an additional `typedef`)
- ▶ member functions (methods) can specify `this` to be `const`.

Many of the above changes are 'just what you expect from programming in Java'.

C++ auto and thread_local

C's storage classes are `auto`, `extern`, `static`, `register`. In C++:

- ▶ `auto` is reused in initialised definitions to mean 'the type of the initialising expression', e.g. `auto x = foo(3);`
- ▶ `thread_local` is an additional storage class, e.g. `static int x = 4; thread_local int y = 5;`
- ▶ `register` is removed since C++17.

C++ booleans

- ▶ type `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 zero becomes `false` (NB: differs from `enum`, see next slide).

C++ enumeration

- ▶ 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(!!)
```

- ▶ Subtlety: Why is 5 'right' (but 8 would be wrong)? Answer: C++ rules to ensure 'bitmaps' work:
 - ▶ 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++ references 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; compare:
- ▶ 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:

```
int i[] = {1,3}; int &refi = i[0]; int *ptri = &i[0];  
refi++; // increments value referenced to 2  
ptri++; // increments the pointer to &i[1]
```

Think of reference types as pointer types with implicit `*` at every use. Subtlety (non-examinable): C++11 added 'rvalue references', e.g.

```
int &&lvr, useful in copy constructors (see later).
```

References in function arguments

- ▶ When used as a function parameter, a referenced value is not copied; for example:

```
void inc(int& i) { i++;}
```

- ▶ Declare a reference as `const` when no modification takes place
- ▶ It can be noticeably more efficient to pass a large struct by reference
- ▶ Implicit type conversion into a temporary takes place for a `const` reference but results in an error otherwise; for example:

```
1 float fun1(float&);  
2 float fun2(const float&);  
3 void test() {  
4     double v=3.141592654;  
5     fun1(v); // Wrong  
6     fun2(v); // OK, but beware the temporary's lifetime  
7 }
```

- ▶ Cf. Fortran call-by-reference

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:

```
1 void f(double);
2 void f(long);
3 void test() {
4     f(1L); // f(long)
5     f(1.0); // f(double)
6     f(1); // Wrong: 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).

Scoping and overloading

- ▶ Overloading does not apply to functions declared in different scopes; for example:

```
1 void f(int);  
2  
3 void example() {  
4     void f(double);  
5     f(1); //calls f(double);  
6 }
```

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 v, double=10.0);`
- ▶ Be careful of the lexical interaction between `*` and `=`; for example:
`void f(char*=0); // Wrong: '*=' is assignment`

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();
}
```

```
void f() { //usage
    ...
    Stack::push('c');
    ...
}
```

```
namespace Stack { //implementation
    const int max_size = 100;
    char s[max_size];
    int top = 0;

    void push(char c) { ... }
    char pop() { ... }
}
```

Example

```
1 namespace Module1 {int x;}
2
3 namespace Module2 {
4     inline int sqr(const int& i) {return i*i;}
5     inline int halve(const int& i) {return i/2;}
6 }
7
8 using namespace Module1; //"import" everything
9
10 int main() {
11     using Module2::halve; //"import" the halve function
12     x = halve(x);
13     sqr(x);                //Wrong
14 }
```

(Non-examinable: C++20 adds `module` constructs giving more control over name visibility. Think Java 9 'modules', while namespaces are more like Java 'packages'.)

Using namespaces

- ▶ 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
- ▶ A namespace can be defined more than once
 - ▶ Allows, for example, internal and external library definitions
- ▶ 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;`
- ▶ The global function `main()` cannot be inside a namespace

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:

```
1 extern "C" {  
2   int globalvar; //definition  
3   int f();  
4   void g(int);  
5 }
```

Why do we need this?

- ▶ 'Name munging' for overloaded functions. A C compiler typically generates linker symbol `_'f'` for `f` above, but (in the absence of `extern "C"`) a C++ compiler typically generates `'_Z1fv'`.
- ▶ Function calling sequences may also differ (e.g. for exceptions).

Linking C and C++ code

- ▶ What if I want to write a library in C, and specify it via `mylib.h` which is importable into both C and C++?
- ▶ Use conditional compilation (`#ifdef`) in `mylib.h`, e.g.

```
1 #ifdef __cplusplus
2     extern "C" void myfn(int, bool);
3 #else
4 # include <stdbool.h> // Ensure type bool defined in C
5     extern void myfn(int, bool);
6 #endif
```

Linking C and C++ code

- ▶ Care must be taken with pointers to functions and linkage:

```
1 extern "C" void qsort(void* p, \  
2                 size_t nmemb, size_t size, \  
3                 int (*compar)(const void*, const void*));  
4  
5 int compare(const void*,const void*);  
6  
7 char s[] = "some chars";  
8 qsort(s,9,1,compare); //Wrong
```

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 in C++

C++ classes are somewhat like Java:

- ▶ Classes contain both data members and member functions (methods) which act on the data; they can extend (syntax ':') other classes.
- ▶ Members can be `static` (i.e. per-class)
- ▶ Members have access control: `private`, `protected` and `public`
- ▶ Classes are created with `class` or `struct` keywords
 - ▶ `struct` members default to `public` access; `class` to `private`
- ▶ A member function with the same name as a class is called a constructor
- ▶ Can use overloading on constructors and member functions.

But also:

- ▶ A member function with the same name as the class, prefixed with a tilde (`~`), is called a destructor

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 we 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 `::` (the scope-resolution operator)
- ▶ C++ uses `new` to allocate and `delete` to de-allocate. There is no garbage collector—users must de-allocate heap objects themselves.

Example (emphasising differences from Java)

```
1 class Complex {
2     double re, im; // private by default
3     public:
4     Complex(double r=0.0, double i=0.0);
5 };
6
7 Complex::Complex(double r,double i) : re(r), im(i) {
8     // preferred form, necessary for const fields
9 }
10
11 Complex::Complex(double r,double i) {
12     re=r, im=i; // deprecated initialisation-by-assignment
13 }
14
15 int main() {
16     Complex c(2.0), d(), e(1,5.0);
17     return 0;
18 } // local objects c,d,e are deallocated on scope exit
```

New behaviours w.r.t. Java

In Java constructors are only used to initialise heap storage, and the only way we can update a field of an object is by `x.f = e;`.

In C++ having object values as first-class citizens gives more behaviours. Consider the following, given `class C`

```
C x;      // how is x initialised? (default constructor)
C y = x;  // how is y initialised? (copy constructor)
x = y;    // what does the assignment do? (assignment operator)
          // what happens to x,y on scope exit? (destructor)
```

For C `structs`, these either perform bit copies or leave `x` uninitialised.

C++ class definitions may need to control the above behaviours to preserve class invariants and object encapsulation.

Constructors and destructors

- ▶ A default constructor is a function with no arguments (or only default arguments)
- ▶ The programmer can specify one or more constructors, but as in Java, only one is called when an object is created.
- ▶ If no constructors are specified, the compiler generates a default constructor (which does as little initialisation as possible).
- ▶ To forbid users of a class from using a default constructor then define it explicitly and declare it `private`.
- ▶ There can only be one destructor
 - ▶ This is called when a stack-allocated object goes out of scope (including when an exception causes this to happen—see later) or when a heap-allocated object is deallocated with `delete`;
 - ▶ Stack-allocated objects with destructors are a useful way to release resources on scope exit (similar effect as Java try-finally) – “RAII: Resource Acquisition is Initialisation”.
 - ▶ Make destructors virtual if class has subtypes or supertypes.

Copy constructor

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

```
1 Complex c(1,2); // note this C++ initialiser syntax;  
2                // it calls the two-argument constructor  
3 Complex d = c;  // copy constructor called
```

- ▶ In the second case, by default object `d` is initialised with copies of all of the non-static member variables of `c`; no constructor is called
- ▶ If this behaviour is undesirable (e.g. consider a class with a pointer as a member variable) define your own copy constructor:
 - ▶ `Complex::Complex(const Complex&) { ... }`
- ▶ To forbid users of a class from copying objects, 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:

```
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:

```
1 Complex& Complex::operator=(const Complex& c) {  
2     ...  
3 }
```

- ▶ 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).

Constant member functions

- ▶ Member functions can be declared `const`
- ▶ Prevents object members being modified by the function:

```
1 double Complex::real() const {  
2     // forbidden to modify 're' or 'this->re' here  
3     return re;  
4 }
```

- ▶ The syntax might appear odd at first, but note that `const` above merely qualifies the (implicit/hidden) parameter `'this'`. So here `this` is effectively declared as `const Complex *this` instead of the usual `Complex *this`.
- ▶ Helpful to both programmer (maintenance) and compiler (efficiency).

Arrays and heap allocation

- ▶ 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:

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

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

```
delete c;
```

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

```
1 Complex* c = new Complex[5];  
2 ...  
3 delete[] c; //Using "delete" is wrong here
```

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

Exercises

1. Write an implementation of a class `LinkedList` 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:

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

Hint: heap allocation & deallocation should occur exactly once!

Operators

- ▶ C++ allows the programmer to overload the built-in operators
- ▶ For example, a new test for equality:

```
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:

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

- ▶ Almost all operators can be overloaded, including address-taking, assignment, array indexing and function application. It's probably bad practice to define `++x` and `x+=1` to have different meanings!

Streams

- ▶ Overloaded operators also work with built-in types
- ▶ Overloading is used to define << (C++'s "printf"); for example:

```
1 #include <iostream>
2
3 int main() {
4     const char* s = "char array";
5
6     std::cout << s << std::endl;
7
8     //Unexpected output; prints &s[0]
9     std::cout.operator<<(s).operator<<(std::endl);
10
11    //Expected output; prints s
12    std::operator<<(std::cout,s);
13    std::cout.operator<<(std::endl);
14    return 0;
15 }
```

- ▶ Note `std::cin`, `std::cout`, `std::cerr`

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
- ▶ For example:

```
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.

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:

```
1 class Z {  
2     Complex c;  
3     Complex d;  
4     Z(double x, double y): c(x,y), d(y) {  
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:

```
1 string a("A "), b("string");
2 const char *s1 = a.c_str();      //OK
3 const char *s2 = (a+b).c_str(); //Wrong
4 ...
5 //s2 still in scope here, but the temporary holding
6 //"a+b" has been deallocated
7 ...
8 string tmp = a+b;
9 const char *s3 = tmp.c_str();   //OK
```

[Non-examinable:] C++11 added rvalue references `&&` to help address this issue.

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.

```
1 class Matrix {  
2     ...  
3     friend Vector operator*(const Matrix&, const Vector&);  
4     ...  
5 };  
6 }
```

This code allows `operator*` to access the private fields of `Matrix`, even though it is defined elsewhere. Mental model: granting your lawyer rights to access your private papers.

- ▶ Note that friendship isn't symmetric.

Inheritance

- ▶ C++ allows a class to inherit features of another:

```
1 class vehicle {
2     int wheels;
3 public:
4     vehicle(int w=4):wheels(w) {}
5 };
6
7 class bicycle : public vehicle {
8     bool panniers;
9 public:
10    bicycle(bool p):vehicle(2),panniers(p) {}
11 };
12
13 int main() {
14     bicycle(false);
15 }
```

Derived member function call

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

- ▶ Default derived member function call semantics differ from Java:

```
1 class vehicle {
2   int wheels;
3 public:
4   vehicle(int w=4):wheels(w) {}
5   int maxSpeed() {return 60;}
6 };
7
8 class bicycle : public vehicle {
9   int panniers;
10 public:
11   bicycle(bool p=true):vehicle(2),panniers(p) {}
12   int maxSpeed() {return panniers ? 12 : 15;}
13 };
```

Example

```
1 #include <iostream>
2 #include "example13.hh"
3
4 void print_speed(vehicle &v, bicycle &b) {
5     std::cout << v.maxSpeed() << " ";
6     std::cout << b.maxSpeed() << std::endl;
7 }
8
9 int main() {
10     bicycle b = bicycle(true);
11     print_speed(b,b); //prints "60 12"
12 }
```

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:

```
1 class vehicle {
2     int wheels;
3     public:
4     vehicle(int w=4):wheels(w) {}
5     virtual int maxSpeed() {return 60;}
6 };
```

Virtual functions

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

```
1 bicycle b(true);
2 vehicle v;
3 vehicle* pv;
4
5 user_input() ? pv = &b : pv = &v;
6
7 std::cout << pv->maxSpeed() << std::endl;
8 }
```

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
- ▶ Indirect (virtual) function calls are slower than direct function calls.
- ▶ Question: virtual function calls are compulsory in Java; is C++'s additional choice of virtual/non-virtual calls good for efficiency or bad for being an additional source of bugs?
- ▶ C++ vtables also contain an encoding of the class type: 'run-time type information' (RTTI). Syntax `typeid(e)` gives the type of `e` encoded as an object of `type_info` which is defined in standard header `<typeinfo>`.

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:

```
1 class shape {  
2   public:  
3     virtual void draw() = 0;  
4 }
```

- ▶ It is forbidden 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'.

Example

```
1 class shape {
2 public:
3     virtual void draw() = 0;
4 };
5
6 class circle : public shape {
7 public:
8     //...
9     void draw() { /* impl */ }
10 };
```

Multiple inheritance

- ▶ It is possible to inherit from multiple base classes; for example:

```
1 class ShapelyVehicle: public vehicle, public shape {  
2     ...  
3 }
```

- ▶ Members from both base classes exist in the derived class
- ▶ If there is a name clash, explicit naming is required
- ▶ This is done by specifying the class name; for example:

```
ShapelyVehicle sv;  
sv.vehicle::maxSpeed();
```

Multiple instances of a base class

- ▶ With multiple inheritance, we can build:

```
1 class A { int var; };  
2 class B : public A {};  
3 class C : public A {};  
4 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 accesses to members of **A** within a **D** must be stated explicitly:

```
1 D d;  
2 d.B::var=3;  
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

```
1 class Vehicle {int VIN;};  
2 class Boat : public virtual Vehicle { ... };  
3 class Car : public virtual Vehicle { ... };  
4 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++

- ▶ In C, casts play multiple roles, e.g. given `double *p`

```
1 int i = (int)*p;    // well-defined, safe
2 int j = *(int *)p; // undefined behaviour
```

- ▶ In C++ the role of constructors and casts overlap. Given `double x` consider (slide 25 defines `Complex`):

```
1 Complex c1(x,0);    // C++ initialisation syntax
2 Complex c2 = Complex(x); // beware (two constructors?)
3 Complex c3 = x;     // OK, but 'explicit' would forbid
4 int i0 = (int)x;    // classic C syntax
5 int i1(x);         // C++ initialisation syntax
6 int i2 = int(x);   // C++ constructor syntax for cast
7 int i3 = x;        // implicit cast
```

- ▶ `c3` is OK—the `Complex` constructor can take one argument. Declare the constructor `explicit` if you want to disallow `c3` (but not `c2`). Compare `i3`, some languages might forbid this.

Casts from a class type

What if I want to write either of the following:

```
1 Complex c;  
2 double d1 = (double)c; // explicit cast  
3 double d2 = c;         // implicit cast
```

These are faulted by the type checker.

Answer: overload operator `double operator()` for class `Complex`:

```
1 Class Complex {  
2     ...  
3     operator double () const { return re; }  
4 }
```

Adding qualifier `explicit` requires casts to be explicit, allowing `d1` but forbidding `d2`.

Casts in C++ (new forms)

Downsides of C-style casts:

- ▶ hard to find (and classify) using a text editor in C or Java.
- ▶ they do no checking (cf. Java downcasts)

C++ encourages the more-descriptive forms:

- ▶ `dynamic_cast<T>(e)`: like Java reference casts: run-time checks when casting pointers within an inheritance hierarchy. This uses RTTI.
- ▶ `static_cast<T>(e)`: nearest to C—best efforts at compile time, e.g. `static_cast<int>(3.14)`.
- ▶ `reinterpret_cast<T>(e)`: to explicitly flag re-use of bit patterns.
- ▶ `const_cast<T>(e)`: remove `const` (or `volatile`) from a type, to modify something the type says you can't!

Pointer casts and multiple inheritance

C-style casts `(C1 *)p` (and indeed `static_cast<C1 *>(p)`) are risky in an inheritance hierarchy 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 require an addition or indirection).

Java single inheritance means that storage for a base class is always at offset zero in any subclass, making casting between references a no-op (albeit with a run-time check for a downcast).

Exercises

1. If a function `f` has a static instance of a class as a local variable, when might the class constructor be called?
2. Write a class `Matrix` which allows a programmer to define 2×2 matrices. Overload the common operators (e.g. `+`, `-`, `*`, and `/`)
3. Write a class `Vector` which allows a programmer to define a vector of length two. Modify your `Matrix` and `Vector` classes so that they inter-operate correctly (e.g. `v2 = m*v1` should work as expected)
4. Why should destructors in an abstract class almost always be declared `virtual`?

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

C++ has no try-finally (use local variables having destructors – RAI).

Throwing exceptions

- ▶ Exceptions in C++ 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:

```
1 void f() { ... throw MyError(); ... }
2 ...
3     try {
4         f();
5     }
6     catch (MyError) {
7         //handle error
8         throw; //re-throw error
9     }
```

Conveying information

- ▶ The “thrown” type can carry information:

```
1 struct MyError {
2     int errorcode;
3     MyError(i):errorcode(i) {}
4 };
5
6 void f() { ... throw MyError(5); ... }
7
8 try {
9     f();
10 }
11 catch (MyError x) {
12     //handle error (x.errorcode has the value 5)
13     ...
14 }
```

Handling multiple errors

- ▶ Multiple catch blocks can be used to catch different errors:

```
1 try {
2     ...
3 }
4 catch (MyError x) {
5     //handle MyError
6 }
7 catch (YourError x) {
8     //handle YourError
9 }
```

- ▶ The wildcard syntax `catch(...)` catches all exceptions but discouraged in practice (what have you caught?)
- ▶ Class hierarchies can be used to express exceptions. BUT, they need RTTI for the following code to work (the virtual function in `SomeError` causes it to have a vtable—and hence RTTI):

```
1 #include <iostream>
2
3 struct SomeError {virtual void print() = 0;};
4 struct ThisError : public SomeError {
5     virtual void print() {
6         std::cout << "This Error" << std::endl;
7     }
8 };
9 struct ThatError : public SomeError {
10     virtual void print() {
11         std::cout << "That Error" << std::endl;
12     }
13 };
14 int main() {
15     try { throw ThisError(); }
16     catch (SomeError& e) { //reference, not value
17         e.print();
18     }
19     return 0;
20 }
```

Exceptions and local variables [important]

- ▶ 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 Acquisition is Initialisation".

Templates

- ▶ Templates support metaprogramming, where code can be evaluated at compile time rather than run time
- ▶ Templates support generic programming by allowing types to be parameters in a program
- ▶ Generic programming means we can write one set of algorithms and one set of data structures to work with objects of any type
- ▶ We can achieve some of this flexibility in C, by casting everything to `void *` (e.g. `sort` routine presented earlier), but at the cost of losing static checking.
- ▶ The C++ Standard Library makes extensive use of templates
- ▶ C++ templates are similar to, but richer than, Java generics.

Templates – big-picture view (TL;DR)

- ▶ Templates are like Java generics, but can have both type and value parameters:

```
template <typename 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: ‘metaprogramming’.
- ▶ Top-level functions can also be templated, with ML-style inference allowing template parameters to be omitted, given

```
1 template<typename T> void sort(T a[], const unsigned& len);  
2 int a[] = {2,1,3};
```

then `sort(a,3) ≡ sort<int>(a,3)`

- ▶ The rest of the slides explore the details.

An example: a stack [revision]

- ▶ The stack data structure is a useful data abstraction concept for objects of many different types
- ▶ In one program, we might like to store a stack of `ints`
- ▶ In another, a stack of `NetworkHeader` objects
- ▶ Templates allow us to write 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`

[Just like Java so far.]

Template for Stack

- ▶ A class template is defined in the following manner:

```
template<typename T> class Stack { ... }
```

or equivalently (using historical pre-ISO syntax)

```
template<class T> class Stack { ... }
```

- ▶ Instantiating such a `Stack` is syntactically like Java, so (e.g.) we can declare a variable by `Stack<int> intstack;`
- ▶ Note that template parameter `T` can in principle be instantiated to any C++ type (here `int`). Java programmers: note Java forbids `List<int>` (generics cannot use primitive types); this is a good reason to prefer syntax `template <typename T>` over `template <class T>`.
- ▶ 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

```
1 // example16.hh
2
3 template<typename T> class Stack {
4
5     struct Item { //class with all public members
6         T val;
7         Item* next;
8         Item(T v) : val(v), next(0) {}
9     };
10    Item* head;
11    // forbid users being able to copy stacks:
12    Stack(const Stack& s) {} //private
13    Stack& operator=(const Stack& s) {} //private
14 public:
15    Stack() : head(0) {}
16    ~Stack(); // should generally be virtual
17    T pop();
18    void push(T val);
19    void append(T val);
20 };
```

```
1 // sample implementation and use of template Stack:
2
3 #include "example16.hh"
4
5 template<typename T> void Stack<T>::append(T val) {
6     Item **pp = &head;
7     while(*pp) {pp = &((*pp)->next);}
8     *pp = new Item(val);
9 }
10
11 //Complete these as an exercise
12 template<typename T> void Stack<T>::push(T) { /* ... */}
13 template<typename T> T Stack<T>::pop() { /* ... */}
14 template<typename T> Stack<T>::~~Stack() { /* ... */}
15
16 int main() {
17     Stack<char> s;
18     s.push('a'), s.append('b'), s.pop();
19 }
```

Template richer details

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

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

- ▶ A template can take several parameters:

```
template<typename T,int i> class Buf { T b[i]; ... };
```

- ▶ A template parameter can be used to declare a subsequent parameter:

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

- ▶ Template parameters may be given default values

```
1 template <typename T,int i=128> struct Buffer{
2     T buf[i];
3 };
4
5 int main() {
6     Buffer<int> B; //i=128
7     Buffer<int,256> C;
8 }
```

Templates behave like macros

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

```
template<typename T> class B {const static T a=3;};
```

- ▶ `B<int> b;` is fine, but what about `B<B<int> > bi;`?

Historically, template expansion behaved like macro expansion and could give rise to mysterious diagnostics for small errors; C++20 adds syntax for `concept` to help address this.

- ▶ Template definitions often need to go in a header file, since the compiler needs the source to instantiate an object

Java programmers: in Java generics are implemented by “type erasure”. Every generic type parameter is replaced by `Object` so a generic class compiles to a single class definition. Each call to a generic method has casts to/from `Object` inserted—these can never fail at run-time.

Template specialisation

- ▶ The `typename 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)

```
1 #include <iostream>
2 class A {};
3
4 template<typename T> struct B {
5     void print() { std::cout << "General" << std::endl;}
6 };
7 template<> struct B<A> {
8     void print() { std::cout << "Special" << std::endl;}
9 };
10 int main() {
11     B<A> b1;
12     B<int> b2;
13     b1.print(); //Special
14     b2.print(); //General
15 }
```

Templated functions

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

```
1 template<typename T> void sort(T a[],  
2                               const unsigned int& len);
```

- ▶ The type of the template is inferred from the argument types:

```
int a[] = {2,1,3}; sort(a,3);  $\implies$  T is an int
```

- ▶ The type can also be expressed explicitly:

```
sort<int>(a,3)
```

- ▶ There is no such type inference for templated classes

- ▶ Using templates in this way enables:

- ▶ better type checking than using `void *`
- ▶ potentially faster code (no function pointers in vtables)
- ▶ larger binaries if `sort()` is used with data of many different types

```
1 #include <iostream>
2
3 template<typename T> void sort(T a[], const unsigned int& len) {
4     T tmp;
5     for(unsigned int i=0;i<len-1;i++)
6         for(unsigned int j=0;j<len-1-i;j++)
7             if (a[j] > a[j+1]) //type T must support "operator>"
8                 tmp = a[j], a[j] = a[j+1], a[j+1] = tmp;
9 }
10
11 int main() {
12     const unsigned int len = 5;
13     int a[len] = {1,4,3,2,5};
14     float f[len] = {3.14,2.72,2.54,1.62,1.41};
15
16     sort(a,len), sort(f,len);
17     for(unsigned int i=0; i<len; i++)
18         std::cout << a[i] << "\t" << f[i] << std::endl;
19 }
```

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 metaprogramming

Template metaprogramming means separating compile-time and run-time evaluation (we use `enum` to ensure compile-time evaluation of `fact<7>`).

```
1 #include <iostream>
2
3 template<unsigned int n> struct fact {
4     enum { value = n * fact<n-1>::value };
5 };
6
7 template <> fact fact<0> {
8     enum { value = 1 };
9 };
10
11 int main() {
12     std::cout << "fact<7>::value = "
13         << (unsigned int)fact<7>::value << std::endl;
14 }
```

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

Exercises

1. Provide an implementation for:

```
template<typename T> T Stack<T>::pop(); and  
template<typename T> Stack<T>::~~Stack();
```

2. Provide an implementation for:

```
Stack(const Stack& s); and  
Stack& operator=(const Stack& s);
```

3. Using metaprogramming, 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?

Miscellaneous things [non-examinable]

- ▶ C++ annotations `[[thing]]` – like Java `@thing`
- ▶ C++ lambdas: like Java, but lambda is spelt `[]`. E.g.

```
1 auto addone = [](int x){ return x+1; }
2 std::cout << addone(5);
```

Lambdas have class type (like Java), and the combination of `auto` and overloading the `operator()` makes everything just work.

Placing variables between the `[]` enables access to free variables: default by rvalue, prefix with `&` for lvalue, e.g. `[i,&j]`

- ▶ C++20 lets programmers define operator `<=>` (3-way compare) on a class, and get 6 binary comparisons (`==`, `<`, `<=` etc.) for free.
- ▶ use keyword `constexpr` to require an expression to be compile-time evaluable—helps with template metaprogramming.
- ▶ use `nullptr` for new C++ code—instead of `NULL` or `0`, which still largely work.