Programming in C and C++

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

Alan Mycroft[1]

Computer Laboratory, University of Cambridge

Michaelmas Term 2019–2020

[1]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.

What we'll cover

- Differences between C and C++
- References versus pointers
- Overloading functions and operators
- Objects in C++; Classes and structs; Destructors; Virtual functions
- Multiple inheritance; Virtual base classes; Casting
- Exceptions
- Templates and meta-programming

- 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.

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`

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 <u>or</u> C++ at the start of a project.

C++ Types [big picture]

C++ types are like C types, but:

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

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

C++ fundamental types

- 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

- 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; compare:
  `int i[] = {1,3}; int &refi = i[0]; int *ptri = &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`
  `ptri++; // increments the pointer to &i[1]`

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

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:

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

- 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:

```
void f(double);
void f(long);
void test() {
  f(1L);  // f(long)
  f(1.0); // f(double)
  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:

```
void f(int);

void example() {
  void f(double);
  f(1); //calls f(double);
}
```

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();
}
```

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

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

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


Example

```
namespace Module1 {int x;}

namespace Module2 {
  inline int sqr(const int& i) {return i*i;}
  inline int halve(const int& i) {return i/2;}
}

using namespace Module1; //"import" everything

int main() {
  using Module2::halve;  //"import" the halve function
  x = halve(x);
  sqr(x);                //Wrong
}
```


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
- 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 munging’ for overloaded functions. A C compiler typically generates linker symbol ‘`_f`’ for `f` above, but a C++ compiler typically generates ‘`__Z1fv`’.
- Function calling sequences may also differ (e.g. for exceptions).

Linking C and C++ code

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

```
extern "C" void qsort(void* p, \
                  size_t nmemb, size_t size, \
                  int (*compar)(const void*, const void*));

int compare(const void*,const void*);

char s[] = "some chars";
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 <u>member functions</u> (methods) which act on the data; they can extend ‘`:`’ other classes.
- Members can be `static` (i.e. per-<u>class</u>)
- 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 <u>constructor</u>
- 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 <u>destructor</u>

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 '`::`'
- 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)

```
class Complex {
  double re, im;  // private by default
 public:
  Complex(double r=0.0, double i=0.0);
};

Complex::Complex(double r,double i) : re(r), im(i) {
  // preferred form, necessary for const fields
}

Complex::Complex(double r,double i) {
  re=r, im=i;  // deprecated initialisation-by-assignment
}

int main() {
  Complex c(2.0), d(), e(1,5.0);
  return 0;
} // local objects c,d,e are deallocated on scope exit
```


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 defined by initialisation; for example;
  `Complex c(1,2); // note this C++ initialiser syntax`
  `Complex d = c;`
- 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 your 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 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:

```
Complex c(), d(1.0,2.3);
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:

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

- 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:

```
double Complex::real() const {
  return re;
}
```

- 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).

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:
  `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 an the first element of an array of objects, array deletion needs different syntax:

```
Complex* c = new Complex[5];
  ...
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

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:

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

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:

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

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

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

- Almost all operators can be overloaded

Streams

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

```
#include <iostream>

int main() {
  const char* s = "char array";

  std::cout << s << std::endl;

  //Unexpected output; prints &s[0]
  std::cout.operator<<(s).operator<<(std::endl);

  //Expected output; prints s
  std::operator<<(std::cout,s);
  std::cout.operator<<(std::endl);
  return 0;
}
```

- 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:

```
Complex& Complex::operator+=(Complex b) {
  re += b.real();
  this->im += b.imag();
  return *this;
}
```

- 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:

```
class X {
  Complex c;
  Complex d;
  X(double a, double b): c(a,b), d(b) {
    ...
  }
};
```

- 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:

```
string a("A "), b("string");
const char *s1 = a.c_str();     //Okay
const char *s2 = (a+b).c_str(); //Wrong
 ...
//s2 still in scope here, but the temporary holding
//"a+b" has been deallocated
 ...
string tmp = a+b;
const char *s3 = tmp.c_str();   //Okay
```


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.

```
class Matrix {
  ...
  friend Vector operator*(const Matrix&, const Vector&);
   ...
  };
}
```

  This code allows `operator*` to access the private fields of `Matrix`, even though it is defined elsewhere.
- Note that friendship isn't symmetric.

Inheritance

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

```
class vehicle {
  int wheels;
public:
  vehicle(int w=4):wheels(w) {}
};

class bicycle : public vehicle {
  bool panniers;
public:
  bicycle(bool p):vehicle(2),panniers(p) {}
};

int main() {
  bicycle(false);
}
```


Derived member function call

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

- Default derived member function call semantics differ from Java:

```
class vehicle {
  int wheels;
public:
  vehicle(int w=4):wheels(w) {}
  int maxSpeed() {return 60;}
};

class bicycle : public vehicle {
  int panniers;
public:
  bicycle(bool p=true):vehicle(2),panniers(p) {}
  int maxSpeed() {return panniers ? 12 : 15;}
};
```

Example

```cpp
#include <iostream>
#include "example13.hh"

void print_speed(vehicle &v, bicycle &b) {
  std::cout << v.maxSpeed() << " ";
  std::cout << b.maxSpeed() << std::endl;
}

int main() {
  bicycle b = bicycle(true);
  print_speed(b,b); //prints "60 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:

```cpp
class vehicle {
  int wheels;
 public:
  vehicle(int w=4):wheels(w) {}
  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:

```cpp
bicycle b(true);
vehicle v;
vehicle* pv;

user_input() ? pv = &b : pv = &v;

std::cout << pv->maxSpeed() << std::endl;
}
```


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`'.

Example

```
class shape {
public:
  virtual void draw() = 0;
};

class circle : public shape {
public:
  //...
  void draw() { /* impl */ }
};
```


Multiple inheritance

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

```
class ShapelyVehicle: public vehicle, public shape {
 ...
}
```

- Members from <u>both</u> 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:

```
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 `C(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>`.

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

But there is no try-finally (use local variables having destructors).

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:

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


Conveying information

- The "thrown" type can carry information:

```
struct MyError {
  int errorcode;
  MyError(i):errorcode(i) {}
};

void f() { ... throw MyError(5); ... }

try {
  f();
}
catch (MyError x) {
  //handle error (x.errorcode has the value 5)
  ...
}
```


Handling multiple errors

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

```
try {
  ...
}
catch (MyError x) {
  //handle MyError
}
catch (YourError x) {
  //handle YourError
}
```

- Every exception will be caught with `catch(...)`
- Class hierarchies can be used to express exceptions:


```
#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 { throw ThisError(); }
  catch (SomeError& e) { //reference, not value
    e.print();
  }
  return 0;
}
```

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 Allocation is Initialisation".

Templates

- Templates support meta-programming, 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)
- 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 <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);
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 `int`s
- 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`

Creating a stack template

- A class template is defined as:

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

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


```
template<class T> class Stack {

  struct Item { //class with all public members
    T val;
    Item* next;
    Item(T v) : val(v), next(0) {}
  };

  Item* head;

  Stack(const Stack& s) {}            //private
  Stack& operator=(const Stack& s) {} //

public:
  Stack() : head(0) {}
  ~Stack();        // should generally be virtual
  T pop();
  void push(T val);
  void append(T val);
};
```



```
#include "example16.hh"

template<class T> void Stack<T>::append(T val) {
  Item **pp = &head;
  while(*pp) {pp = &((*pp)->next);}
  *pp = new Item(val);
}

//Complete these as an exercise
template<class T> void Stack<T>::push(T) {/* ... */}
template<class T> T Stack<T>::pop() {/* ... */}
template<class T> Stack<T>::~Stack() {/* ... */}

int main() {
  Stack<char> s;
  s.push('a'), s.append('b'), s.pop();
}
```


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<class T,int i> class Buf { T b[i]; ... };`
- A template can even use one template parameter in the definition of a subsequent parameter: `template<class T, T val> class A { ... };`
- A templated class is not type checked until the template is instantiated: `template<class T> class B {const static T a=3;};`
  - `B<int> b;` is fine, but what about `B<B<int> > bi;`?
- Template definitions often need to go in a header file, since the compiler needs the source to instantiate an object

Default parameters

- Template parameters may be given default values

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

int main() {
  Buffer<int> B; //i=128
  Buffer<int,256> C;
}
```


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)

```
#include <iostream>
class A {};

template<class T> struct B {
  void print() { std::cout << "General" << std::endl;}
};
template<> struct B<A> {
  void print() { std::cout << "Special" << std::endl;}
};

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


Templated functions

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

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

- The type of the template is inferred from the argument types: `int a[] = {2,1,3}; sort(a,3);` ⟹ `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


```
#include <iostream>

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]) //type T must support "operator>"
       tmp = a[j], a[j] = a[j+1], a[j+1] = tmp;
}

int main() {
  const unsigned int len = 5;
  int a[len] = {1,4,3,2,5};
  float f[len] = {3.14,2.72,2.54,1.62,1.41};

  sort(a,len), sort(f,len);
  for(unsigned int i=0; i<len; i++)
    std::cout << a[i] << "\t" << f[i] << std::endl;
}
```

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

```
#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
                  // at compile time!
};
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:
   `template<class T> T Stack<T>::pop();` and
   `template<class T> Stack<T>::~Stack();`
2. Provide an implementation for:
   `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?