C++

C++ for Fortran programmers

Kevin Cowtan
cowtan@ysbl.york.ac.uk

Part 0: C++ Syntax:
Basics

C++

- C++ is a compiled programming language, like Fortran.
- It contains a range of very similar data types, control statements, and input/output facilites.
- It also includes a range of modern programming features, in particular:
  - Dynamic memory allocation (F90)
  - Namespaces
  - **Object orientation**

C++

```
    integer i
    real n, s, t
    n = 3.0

    s = 1.0
10  continue
      t = s
      s = (t+n/t)/2.0
    if (abs(s-t).lt.0.01) goto 10

    write (*,*)s
```

C++

```
  int i;
  float n, s, t;
  n = 3.0;

  s = 1.0;
loop:
    t = s;
    s = (t+n/t)/2.0;
  if (abs(s-t)<0.01) goto loop;

  std::cout << s;
```

- Basic concepts (variables etc.) very similar:
  - Types have different names.
  - In C++, statements separated by ';', not newline.
  - 'goto' is labelled. (But this is the last time you'll see it!)

C++

```
      real mat(3,3),x(3),y(3)
      integer i,j

C     initialise mat
C     initialise x

      do i = 1,3
       y(i) = 0.0
       do j = 1,3
        y(i) = y(i) + mat(i,j)*x(j)
       enddo
      enddo
```

C++

```
  float mat[3][3],x[3],y[3];
  int i,j;

// initialise mat
// initialise x

  for ( i=0; i<3; i++ ) {
   y[i] = 0.0;
   for ( j=0; j<3; j++ ) {
    y[i] = y[i] + mat[i][j]*x[j];
   }
  }
```

- C/C++ arrays are different:
  - Indices in '**[ ]**'
  - Indices start at zero.
  - Only 1-d, but for multidimensional use an array of arrays!
- Loop syntax different, loop body enclosed in '**{ }**'.
- Comments use //

C++

- Built in types:

	C++
`real`	`float`
`double precision`	`double`
`integer`	`int`
`character, character()`	`char, char[]` but also `std::string`
`complex`	`std::complex`
`logical`	`bool`

The built in types are very similar.

But in addition to the built in types, C++ has a library called STL of much richer extensions: `std::*`,

for example strings, complex numbers, *resizable arrays*.

C++

- Built in types: One *big* difference:
  - In C++ (not C), variables may be declared at any point in a function. It is normal not to declare a variable until you are about to use it. (Improves readability, saves memory)
  - Variables stay in scope until the end of the block ('**{ ... }**') in which they were declared. This can be a function, or the body of a loop/conditional, or just a self contained group of statements.

```
int i = 9;
{
  int j;
  j = i;  //ok
}
i = j; //error
```

Note also: we can initialise on declaration.

C++

- Control statements: '**if**'

	C++
`if ( ... ) statement`	`if ( ... ) statement;`
`if ( ... ) then` `  statements` `else` `  statements` `endif`	`if ( ... ) statement;` `else       statement;`

Conditionals are similar, but in C/C++, only one statement (ending with ';') follows the condition. Hence no 'endif'.

How do we handle conditions containing multiple statements?

Wherever you can use a single statement, you can also use a list of statements enclosed in '{ }'.

C++

- Control statements: '**if**'

Fortran	C++
`if ( i.lt.0 ) i = 0` `if ( i.gt.6 ) then` `  i = 0` `  j = 1` `endif`	`if ( i < 0 ) i = 0;` `if ( i > 6 ) {` ` i = 0;` ` j = 1;` `}`

Note also that the conditions are different:

`.lt.`	`<`
`.gt.`	`>`
`.ne.`	`!=`
`.eq.`	`==`

Beware:

`'='` for assignment

`'=='` for comparison

You will get this wrong!

C++

- Control statements: '**for**'

`do i=1,n` `  statement` `enddo` `do i=1,n` `  statement` `  statement` `enddo`	`for (i=1;i<=n;i++)` C++ `  statement;` `for (i=1;i<=n;i++) {` `  statement;` `  statement;` `}`

Basic loop syntax very different, but it follows the same rules as 'if' for the contents:

The loop contains one statement, which may be

- a simple statement followed by ';' or
- a compound statement in '{ }'.

C++

- Control statements: '**for**'

Do this between each iteration of the loop

Do the loop as long as this is true

Do this before the loop starts

```
for ( i = 1; i <= n; i++ )
```

Note: '`i++`'

is shorthand for '`i += 1`',

which is shorthand for '`i = i + 1`'

In C/C++ we usually count from zero, hence:

```
for ( i = 0; i < n; i++ )
```

C++

- Control statements: other loops

`10  if (!cond) goto 20` `      statement` `      goto 10` `20  continue` `10    statement` `    if (cond) goto 10`	`while ( condition ) {` `  ...` `}` `do {` `  ...` `} while ( condition )`

Note:

'do' and 'while' loops also available.

Can use for single statements or blocks.

C++

- Control statements: '**break**'

```
    do i = 0,n-1
      ...
      if (err) goto 20
      ...
    enddo
20  continue
```

```
for ( i=0; i<n; i++ ) {
  ...
  if (err) break;
  ...
}
```

A '`break`' statement breaks out of the nearest loop (for/do/while) above it.

(It ignores any intervening non-loop blocks).

C++

- Operators and functions:
  - Most operators are the same.
  - Especially:
    - Parentheses
    - Precendence.
    - Trig and log functions.

<table>
<tr><td>+</td><td>+</td></tr>
<tr><td>-</td><td>-</td></tr>
<tr><td>*</td><td>*</td></tr>
<tr><td>/</td><td>/</td></tr>
<tr><td>f**g</td><td>pow(x,y)</td></tr>
<tr><td>mod(i,j)</td><td>i%j</td></tr>
<tr><td>mod(f,g)</td><td>mod(f,g)</td></tr>
<tr><td>.lt.</td><td><</td></tr>
<tr><td>.le.</td><td><=</td></tr>
<tr><td>.gt.</td><td>></td></tr>
<tr><td>.ge.</td><td>>=</td></tr>
<tr><td>.ne.</td><td>!=</td></tr>
<tr><td>.eq.</td><td>==</td></tr>
<tr><td>.and.</td><td>&&</td></tr>
<tr><td>.or.</td><td>||</td></tr>
<tr><td>.not.</td><td>!</td></tr>
<tr><td>abs(i)</td><td>abs(i)</td></tr>
<tr><td>abs(f)</td><td>fabs(f)</td></tr>
</table>

C++

- Arrays:
  - C++ arrays indexed from 0
  - Declare and index with '**[ ]**'
  - Multidimensional arrays are arrays of arrays.
- But:
  - In C++, we only use arrays for special purposes where we know size is fixed. Otherwise we use more flexible objects.

Fortran	C++
`integer i(3)`	`int i[3];`
`real m(10,10)`	`float m[10][10];`
`character c(20)`	`char c[20];`
`i(3) = i(1)`	`i[2] = i[0];`
`m(1,1) = 1.0`	`m[0][0] = 1.0;`

C++

- Functions:

``integer function fact(n)``` ```integer n``` ```integer i,j``` ```j = 1``` ```do i = 2,n``` ``` j = j * i``` ```enddo``` ```fact = j``` ```return``` ```end```	```int fact(int n)``` ```{``` ``` int i,j;``` ``` j = 1;``` ``` for (i=2;i<=n;i++) {``` ``` j = j * i;``` ``` }``` ``` return j;``` ```}```

  - Define argument types in the function definition.
  - Enclose the body of the function in '**{ }**'.
  - Use '`return`' to return result (if any).
    - Arguments are copied – the function cannot change them in the calling program (in general).

C++

- Functions: (Subroutines)

```fortran
subroutine fact(n,m)
integer n,m
integer i
m = 1
do i = 2,n
  m = m * i
enddo
return
end
```

```cpp
void fact(int n, int& m)
{
  int i;
  m = 1;
  for (i=2;i<=n;i++) {
    m = m * i;
  }
}
```

  - This is an ugly example!
  - If there is no return value, return type is '`void`'.
  - You can pass arguments to be modified by adding '&' after the type.

    i.e. Don't copy the value, just pass a reference to it.

C++

- Functions: (Subroutines)

```
void fact(int n, int& m)

void fact(const int& n, int& m)
```

  - We can pass a reference even when we don't want to return something through it.
  - The keyword 'const' indicates it will not be changed.
  - We passing a complex *object* (rather than a variable), passing a 'const' reference saves a lot of copying.

C++

- Input/Output:

```
        integer x
        character s(20)
        write (*,*)s," scored ",x
```

```
int x;
char s[20];
std::cout << s << " scored " << x;
```

- `std::cout` does free format output.
- `std::cin` (using '>>') does free format input.
- Use '`#include <iostream>`' at the start of the program.

C++

- Formatted Input/Output:

```
      integer x
      character s(20)
      write (*,10)s,x
 10   format (a20," scored ",i4)
```

```
int x;
char s[20];
printf("%20c scored %4i \n",x,s);
```

  - '\n' means 'new line'. You can put it in any string.
  - Use '#include <stdio>' at the start of the program.

C++

- Main program:

```
            program myprog

            ...

            stop
            end
```

```
int main( int argc, char** argv ) {

   ...

}
```

- In C/C++, we write a function called '`main`'.
- It receives parameters from the OS, which are the number of command line arguments, and the array of arguments.

C++

- Main program:

```
#include <iostream>

int main( int argv, char** argc ) {
  for ( int i = 0; i < argv; i++ )
    std::cout << argc[i] << "\n";
}
```

  - We won't go into the '`char**`' notation, but it translates roughly as '`char argc[][]`'.
  - This program prints out a list of all the command line arguments.

C++

- Putting it all together:
  - Write a program which makes a table of numbers with their factorials.
  - Just add one more technique: since we can define variables anywhere, we can define a loop variable in the 'for' statement.

C++

- Putting it all together:

```
#include <iostream>

int fact(int n)
{
  int j = 1;
  for ( int i=2; i<=n; i++ )
    j = j * i;
  return j;
}

int main( int argc, char** argv ) {
  for ( int i=1; i<=10; i++ )
    std::cout << i << " factorial is "
              << fact(i) << "\n";
}
```

C++

Other features:

- C/C++ have other features, which we will avoid using:
  - Pointers: Refer to a variable by its address in memory.
    - **`int *x; float *y;`**
  - Memory allocation:
    - C-style (malloc/free)
    - C++-style (new/delete)
- C++ is a huge language, with several ways of doing anything. We are focusing on a small, well defined, modern subset, based on the standard library STL.

C++

Part 1: C++ Syntax:
Advanced concepts

C++

Namespaces:

- Problem: if we include many libraries in our program, we will eventually get name collisions.
- One solution is to include the library name in every function. Example:
  - `Cmtz_read_file("filename")`
  - `MMDB_read_file("filename")`
  - `Jpeg_read_file("filename")`
- But it is tedious to retype names when we don't need to (e.g. When a library calls itself).
- Solution: namespaces

C++

Namespaces:

- We can hide functions, data, types etc. inside a namespace, to avoid name clashes.
- They can then have obvious short names, with which they refer to one another.
- Anything else outside the namespace must add the name of the namespace before the function (or whatever) name.
  - But if we use something really frequently, we can declare the whole namespace, or individual members, to be available.

C++

- Namespaces:

```
namespace ccp4 { // begin namespace

  int func1( int n ) {
    return 2*n;
  }

  void func2() {
    int i = func1(6);           // 12
  }

}                  // end of namespace

int func1( int n ) {
  return -n;
}

void main() {
  int i1 = func1( 6 );          // -6
  int i2 = ccp4::func1( 6 );  // 12
}
```

C++

- Namespaces:

```
namespace ccp4 { // begin namespace

  int func1( int n ) {
    return 2*n;
  }

}                // end of namespace

using namespace ccp4;   // use this
using ccp4::func1;      //  or this

void main() {
  int i1 = func1( 6 );        // 12
}
```

C++

Overloading

- Overloading allows us to define several functions with the same name.
- The functions differ only in what arguments they accept. The function to be run is chosen by the compiler on the basis of what arguments are provided.
  - Note: you cannot overload on the basis of return value!

C++

Overloading

- e.g. Three ‘minimum’ functions:

```
// minimum of two integers
int min( int x, int y ) {
  if ( x < y ) return x;
  else         return y;
}
// minimum of two floats
float min( float x, float y ) {
  if ( x < y ) return x;
  else         return y;
}
// minimum of three floats
float min( float x, float y, float z ) {
  if ( x < y && x < z ) return x;
  else if ( y < z )     return y;
  else                  return z;
}
```

C++

Overloading

- Note: we will also encounter something called ‘operator overloading’. This allows use to redefine how operators like ‘+’ work under special cases.

C++

- Looking forward:

  C++ has some better ways of doing even simple, non-object oriented tasks.
  - std::string - an advanced string class.
  - std::vector<> - a resizeable array class.

C++

- Looking forward:

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

std::string s = "hello";
std::string t = s.substr(1,3);
int p = s.find("ell");
std::cout << s << "\n";              // hello
std::cout << s.length() << "\n";     // 5
std::cout << t << "\n";              // ell
std::cout << p << "\n";              // 1
```

  - Strings have many useful methods.

C++

- Looking forward:

```
#include <iostream>
#include <vector>

std::vector<float> x(6);
std::cout << x.size() << "\n";  // 6
x[0] = 1.0;
x[5] = 3.142;
x.append( 2.718 );
std::cout << x.size() << "\n";  // 7
x.resize( 1000 );
std::cout << x.size() << "\n";  // 1000
```

  - You can also insert into the middle and delete from the middle of the array.
  - If this is performance critical, use std::list<> instead.

Part 2: Object Orientation: practice

C++

Object orientation is a programming paradigm. Its aim is to allow the creation of more modular, reusable, maintainable code.

It achieves this by many means, including:

- Allowing the code to better reflect the terms of the problem.
- Separating interfaces (APIs) from implementations.
- Allowing existing implementations to be overridden or extended.
- Allowing algorithms to be generic across many data types, even unknown ones.

C++

We will follow the development of programing language features stepwise to arrive at our first objects.

Next, we will look at the ideas of object orientation in a more general way.

Finally, we will bring these things together.

Starting point: data structures.

C++

- Most languages (except F77) allow us to define new, more complex data types. In OO languages, these are called 'classes'. (In C they are 'struct's).
- A class is a new type, which is added to the language.
- We can declare classes just as we would declare a variable of any built in type.
- A class may contain one or more 'member variables' of any type – including other classes.
  - And a whole lot more... later.

C++

- Example: A class for unit cell parameters.

```
class Cell {
 public:
  double a,b,c;
  double alpha,beta,gamma;
};
...
  Cell c1, c2;  // make 2 cell objects
  c1.a = 10.0;
  c1.b = 15.0;
  c1.c = 20.0;
  c1.alpha = c1.beta = c1.gamma = pi/2;

  c2 = c1
  c2.a = 30.0;
  std::cout << c2.a << c2.b << c2.c; // 30 15 20
```

C++

- Example: A class for unit cell parameters.

```
class Cell {
 public:
  double a,b,c;
  double alpha,beta,gamma;
};
```

  - We define a new type, called 'Cell'. (Begins with capital letter, by convention)
  - It has 6 members (all double), which are publicly accessible (see later).
  - Note: at this stage it is not unlike a Fortran common block. But, we can only have one instance of a common block in a program, but we can have many instances of a class.

C++

- Example: A class for unit cell parameters.

```
Cell c1, c2;  // make 2 cell objects
c1.a = 10.0;
c1.b = 15.0;
c1.c = 20.0;
c1.alpha = c1.beta = c1.gamma = pi/2;
```

  - We create two 'objects' of class '`Cell`': `c1` and `c2`.
    Just like making two int-s or float-s.

  - We access the parameters of `c1` by using the name of the object, a dot, and then the name of the member.

  - The other object, `c2`, is unaffected.

C++

- Example: A class for unit cell parameters.

```
c2 = c1
c2.a = 30.0;
std::cout << c2.a << c2.b << c2.c; // 30 15 20
```

  - We can copy all the members of `c1` into `c2` using a simple assignment.

  - We then change just one of the members. `c1` is unaffected.

  - Objects can also be passed to or from functions as arguments or return values.

C++

Terminology:

- An **object** is an instance of a **class**.
- A **class** defines the type of an **object**.
- A class contains **members:**
  - Member variables. (*members*)
  - Member functions. (*methods*)

C++

Member functions (methods):

- A member function is a function which is defined as part of a class.
- It is used on an object of that class, and operates on the member variables of that object and any arguments passed to the function.
- The result of the function may be returned as a return value, or may modify the member variables, or both.

C++

Member functions (methods):

```
class Cell {
 public:
  double a,b,c;
  double alpha,beta,gamma;

  double volume() {
    return a*b*c*sin(beta);  // wrong!
  }
};
```

- The function '`volume`' is defined inside the class.
- We don't need to pass the cell parameters, because they are already members of the class.

C++

Member functions (methods):

```
...
  std::cout << c1.volume();  // 10x15x20 = 3000
  std::cout << c2.volume();  // 30x15x20 = 9000
```

- We call the member function by giving the name of the object, a dot, the name of the function, and then any arguments.

- We get a different result depending on which object we use, because they have different parameters.

C++

Class design:

- Once we have member functions, we can then 'hide' the member variables.
- Instead, we provide accessor functions to allow the member variables to be read or modified.
- The result is a class which has a well defined external interface which completely hides the internal implementation. We can then change how the class works internally without affecting the rest of the program.

This is a huge benefit! Use it!

C++

Class design:

```
class Cell {
 public:
  double a() { return a1; }  // accessors
  double b() { return b1; }
  double c() { return c1; }
  double alpha() { return alpha1; }
  double beta()  { return beta1;  }
  double gamma() { return gamma1; }
  double volume() {
    return a1*b1*c1*sin(beta1);  // wrong!
  }
 private:
  double a1,b1,c1,alpha1,beta1,gamma1;
};
```

- We make the member variables private.
- There is no performance cost - the compiler optimizes.

C++

Class design:

```
class Cell {
 public:
  double a() const { return a1; }  // accessors

 private:
  double a1,b1,c1,alpha1,beta1,gamma1;
};
```

- Methods can be declared as ‘const’,
  i.e. They don’t change the state of the object.
  (The compiler will check this for us).
- Doing this for one accessor doesn’t make much difference, but if we do it *everywhere* then the compiler can do a huge amount of checking for us.

C++

Class design:

- Now that the members are private, how do we set the contents of the object? 3 options...
  - We can define a constructor.
  - We can define 'set' accessors ('setters').
  - For simple cases where the representation is unambiguous, we can return references to the members.

C++

Class design:

```
class Cell {
 public:

  Cell() {}

  Cell( double a, double b, double c,
        double alpha, double beta, double gamma ) {
    a1 = a; b1 = b; c1 = c;
    alpha1 = alpha; beta1 = beta; gamma1 = gamma;
  }

 private:
  double a1,b1,c1,alpha1,beta1,gamma1;
};
```

- Constructor has same name as class.
- Also define null constructor, so we can create an object without initialising it.

C++

Class design:

```
...
  // construct and initialise
  Cell c1( 10.0, 15.0, 20.0, pi/2, pi/2, pi/2 );
  // construct uninitialised
  Cell c2;
  // construct on-the-fly and assign
  c2 = Cell( 30.0, 15.0, 20.0, pi/2, pi/2, pi/2 );
```

- We can use the new constructor in two ways -
  - On declaring the object.
  - On the fly, to create a cell object in the middle of an expression.

C++

Class design:

```
class Cell {
 public:
  void set_cell_parameters
      ( double a, double b, double c,
        double alpha, double beta, double gamma ) {
    a1 = a; b1 = b; c1 = c;
    alpha1 = alpha; beta1 = beta; gamma1 = gamma;
  }

  void set_a( double a ) {
    a1 = a;
  }

 private:
  double a1,b1,c1,alpha1,beta1,gamma1;
};
```

- 'Set' accessors may set one or many members.
- Can also perform calculations.

C++

Class design:

– One last detail: Destructors

```
class Cell {
 public:
  // Constructor
  Cell( double a, double b, double c, ... ) {
    ...
  }

  // Destructor
  ~Cell() {
    ...
  }

  ...
};
```

- Called automatically when a class is destroyed.
- Clean up any memory allocation, i.e. Do not use!

C++

Class design: Summary

- We can build up compound objects containing collections of built-in data types and other objects.
- We can use these objects wherever we would use a built-in type.
- The objects can also contain relevant functions for manipulating the data.
- In the interests of maintaining a stable API, it is best to only allow access to the members of the object through accessor functions.

C++

Part 3: Object Orientation: theory

C++

Object orientation:

- By making objects which describe the objects in our problem domain, we achieve a better match between the program and the problem.
- Interactions between objects in the real world become methods of objects in the program.
- Even in completely abstract problems, we still have a benefit: the abstraction of the API.

C++

Object orientation:

- How do we chose objects? One approach:
  - Write down a description of the problem, then underline all the nouns.
  - Simple things may become properties of objects (i.e. Member variables).
  - More complex things may become objects.
  - Others are abstract or uninteresting or processes and will be ignored or take other forms.
- e.g. Crystal, cell, space-group, coordinate, map, density, FFT, likelihood.

C++

Object Orientation concepts:

- Encapsulation:
  - Hiding the details of data representation behind some interface. (We've seen this already).
- Inheritance:
  - Allows us to customise existing classes to suite our purposes.
- Polymorphism:
  - Allows us to make methods and classes which will work on any of a range of different data types.

C++

Inheritance:

- Inheritance provides an efficient way to re-use code.
  - If we want to make a new class which is similar to an existing class, we can use ‘inheritance’ to do this without touching the original class.
  - The new class is called a ‘sub-class’ or ‘derived-class’. It inherits from its ‘super-class’, or ‘base-class’.
  - The new class has all the members and methods of the parent class, plus any more that are defined.

C++

Inheritance:

- e.g. The unit cell class: a real example:
  - Phil wants to use my Cell class, but he needs a different orthogonalization convention in reciprocal space.
  - So he makes a new class, Cell_cambridge, which inherits from my Cell class.
  - My class has a method which returns the reciprocal orthogonalization matrix.
    - Phil overrides this method with a new method of his own.
    - He can also add completely new methods, and members.

C++

Inheritance:

- e.g. The unit cell class: a real example:

```
class Cell {
 public:
   ...
   Mat33 mat_reci_orth() const { return ... ; }
   ...
};

class Cell_cambridge : public Cell {
 public:
   Mat33 mat_reci_orth() const { return ... ; }
};
```

  - The ‘derivation’ comes after the class name.
  - Any changes are included in the class body.

C++

Inheritance:

- Properly used, inheritance aids code re-use:
  - Define ‘base classes’ which are generic.
  - Derive more specific classes, have more details.
  - e.g.

```
Class Atom
  { double x,y,z,occ; };

Class Atom_isotropic : public Atom
  { double u_iso; };

Class Atom_anisotropic : public Atom
  { double u11,u22,u33,u12,u13,u23; };
```

C++

Inheritance:

- Properly used, inheritance aids code re-use:
  - At first, spotting how to use inheritance can be tricky.
  - One (inefficient) approach while learning:
    - Write all the specific classes
      (using cut and paste for any shared code).
    - Look for any members and method code which can be shared.
    - Implement a base class containing the common features.
  - Even for experienced programmers, class design evolves after the first implementation.

C++

Polymorphism:

- Polymorphism is the ability of classes, methods, and functions to work on a range of different data types, even types which did not exist when the library was written.
- Two forms:
  - Runtime polymorphism: You can use a derived class wherever you can use its base class.
  - Templates [*C++/Java only*]: You can write *template* classes and functions which can take **any** type of data.

C++

Polymorphism: (runtime)

- e.g. Suppose our `Atom` class implements a method '`density_at_xyz(x,y,z)`', for a stationary atom.
  - `Atom_isotropic` and `Atom_anisotropic` will override this method with methods appropriate to atoms with thermal motion.
  - A method for calculating electron density might take a list of atoms, not caring what type of atom is involved.
  - A later developer may then add another atom type (e.g. `Atom_disordered`) with a clever method for calculating density. The electron density calculation will still work!

C++

Polymorphism: (runtime)

- Problems: there are (small) memory and performance overheads:
  - Therefore in C++, runtime polymorphism only occurs for classes which have *virtual* methods, and then only those methods of a class which are explicitly declared as ‘virtual’ (i.e. Can be overridden).
  - Polymorphism only occurs when handling a reference *(or pointer)* to a class which contains virtual methods.
- It’s useful, but in C++, it has limitations.
  - (Clipper uses it, but you don’t need to know any more.)

C++

Polymorphism: (templates)

- Templates are a second form of polymorphism, implemented in C++, which has no performance or memory overheads.
- Template polymorphism occurs at compile time.
- Template polymorphism works on any class (which has the right sort of API), whether or not there is an inheritance relationship involved.

C++

Polymorphism: (templates)

- e.g. `std::vector`: a resize-able array of data of some type.
  - Incredibly useful: use it whenever you want an array whose size isn't absolutely immutable.
  - Makes memory allocation (and therefore memory leaks) obsolete.
  - There are also a whole range of related types, e.g. Singly and doubly linked lists, associative arrays, etc.
- Part of STL: the Standard Template Library.

C++

Polymorphism: (templates)

- e.g. `std::vector`:

```
int n = 10;
std::vector<int> i;
std::vector<float> f( 6 );
std::vector<double> d( n, 1.0 );
std::vector<Cell> c;
```

  - The type of data comes in angle brackets <> after the class name. This tells the compiler to compile a vector for that type of member.
  - The data type can be any built-in or user defined type or class.
  - We can optionally define initial size, and value.

C++

Polymorphism: (templates)

- e.g. `std::vector`:

```
// get the current size
int old_size = d.size();

// resize the list
d.resize( 20 );
```

  - The std::vector class has a method ‘size’ which returns the current size of the list.
  - The std::vector class has a method ‘resize’ which changes the current size of the list.

C++

Polymorphism: (templates)

- e.g. `std::vector`:

```
// sum the values in a list
double sum = 0;
for ( int i = 0; i < d.size(); i++ )
  sum = sum + d[i];
```

  - The std::vector class has a method which looks like standard array subscription (i.e. overrides the bracket operator ‘[]’), which allows us to get at the data is if it were in a normal array.

C++

Polymorphism: (templates)

- e.g. `std::vector`:

```
// add to the end of the vector
d.push_back( 3.142 );

// remove from the end of the vector
double x = d.pop_back();
```

  - The std::vector class has methods which add to or remove from the back of an array.
  - Performance is good – it is not uncommon to build up a large array one element at a time.

C++

Polymorphism: (templates)

- e.g. `std::vector`:

```
// insert at 3 from the front
d.insert( d.begin()+3, 2.718 );

// delete from three from the end
d.delete ( d.end()-2 )
```

  - The std::vector class has methods to insert and deletes at arbitrary positions in the list.
  - (There are performance overheads of course – a linked list may be better).

C++

Polymorphism: (templates)

- e.g. `std::vector`:

```
// sort the list
std::sort( d.begin(), d.end() );
```

  - The std::sort algorithm is the most efficient algorithm known.
  - We usually want to sort by key: Make `std::vector<std::pair<keytype,datatype> >` containing the list of keys and data, and apply a std::sort.

C++

Polymorphism: (templates)

- Crystallographic examples from Clipper:
  - A crystallographic map can contain any sort of data:
    ```
    Xmap<float>
    Xmap<int>
    Xmap<Histogram>
    ```
  - Reflection data can be of any one of a range or predefined or user-defined types, e.g.
    ```
    HKL_data<F_sigF<float> >
    HKL_data<F_phi<double> >
    HKL_data<ABCD<float> >
    ```

Part 4: Odds and ends.

C++

Nested classes:

- A class can be defined inside another class. This is useful for:
  - Classes which are only used internally by another class.
  - Classes which are only used in conjunction with another class.
  - Template programming, when you want to use a bundle of classes for a template type, you can put them inside another class.

C++

Nested classes:

- Outer class is treated like a namespace:

```
Class Outer {
   ...
   Class Inner {
     ...
   };
   ...
};

Outer x;
Outer::Inner y;
```

C++

- Do:
  - Use encapsulaation, const etc.
  - Use STL (standard template library) data structures.
  - Use STL algorithms.
- Don’t:
  - Use memory allocation. As soon as you start using memory allocation, your program can develop memory leaks.
    - new/delete
    - malloc/free
    - If there is no usable STL data structure, write one of your own and test it to destruction.
  - Use pointers, except where absolutely necessary.
    - Encapsulate in STL style template classes.