B16 Object Oriented Programming

Frank Wood

fwood@robots.ox.ac.uk

http://www.robots.ox.ac.uk/~fwood/teaching/

Hilary 2015

This course will introduce object-oriented programming (OOP).

It will introduce C++, including

classes,
methods, function and operator overloading,
constructors,
and program organisation.

We'll discuss data hiding (encapsulation), covering

public and private data,
and accessor methods.

We'll also cover inheritance, along with

polymorphism.

The course will conclude by covering templates, in particular

The Standard Template Library and Design Patterns.

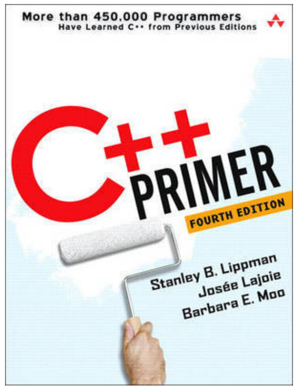
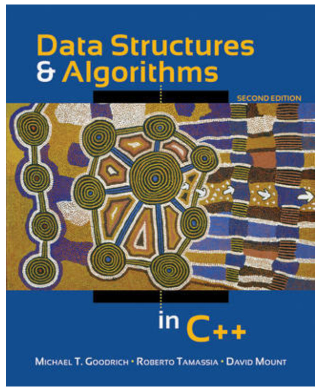
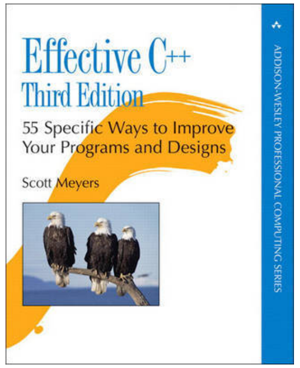
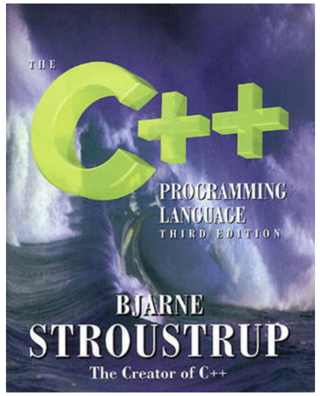
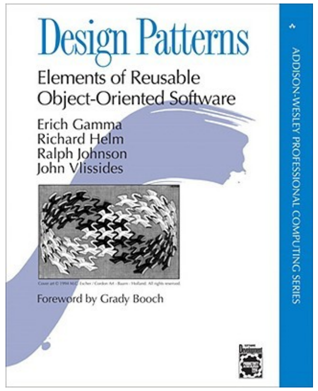
The course will aim to give a good understanding of basic design methods in object-oriented programming, reinforcing principles with examples in C++.

Specifically, by the end of the course students should:

- understand concepts of and advantages of object-oriented design including:
  - Data hiding (encapsulation)
  - Inheritance and polymorphism
  - Templates.
- understand how specific object oriented constructs are implemented using C++.
- Be able to understand C++ programs.
- Be able to write small C++ programs.

There are many useful textbooks.

- http://www.stroustrup.com/4thContents.html
- Lipmann and Lajoie, *C++ Primer,* Addison-Wesley, 2005.
- Goodrich et al., *Data structures and algorithms in C++*, Wiley, 2004
- Stroustrup, *The C++ Programming Language*, Addison-Wesley, 2000
- Meyers, *Effective C++*, Addison-Wesley, 1998
- Gamma et al., *Design Patterns: elements of reusable object-oriented software,* Addison-Wesley, 1995

Topic 1: Programming Paradigms

Top down design means breaking the problem down into components (modules) recursively.

Each module should comprise data and functions that are all related: OOP makes this explicit.

The designer needs to specify how components interact – what their dependencies are, and what the interfaces between them are.

Minimising dependencies, and making interfaces as simple as possible are both desirable to facilitate modularity.

By minimising the ways in which modules can interact, we limit the overall complexity, and hence limit unexpected behaviour, increasing robustness.

Because a particular module interacts with other modules in a carefully defined manner, it becomes easier to test/validate, and can become a reusable component.

A key part of this course will emphasize how C++ provides tools to help the designer/programmer explicitly separate interface and implementation, and so create more modular code.

Consider the general engineering principles of abstraction and modularity.

The idea behind abstraction is to distil the software down to its fundamental parts, and describe these parts precisely, but without cluttering the description with unnecessary details such as exactly how it is implemented.

The abstraction specifies what operations a module is for, without specifying how the operations are performed.

Consider the general engineering principles of abstraction and modularity.

The aim of modularity is to define a set of modules each encapsulates a particular functionality, and which interacts with other modules in well defined ways.

The more complicated the set of possible interactions between modules, the harder it will be to understand.

Humans are only capable of understanding and managing a certain degree of complexity; it is quite easy (but bad practice) to write software that exceeds this capability!

Top-down design achieves abstraction and modularity via four steps.

1. Architectural design: identifying the building blocks.

2. Abstract specification: describe the data/functions and their constraints.

3. Interfaces: define how the modules fit together.

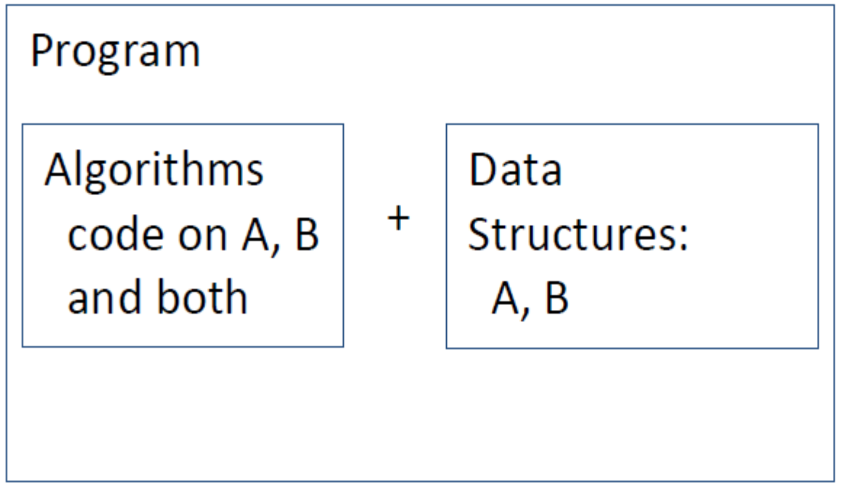
4. Component design: recursively design each block.

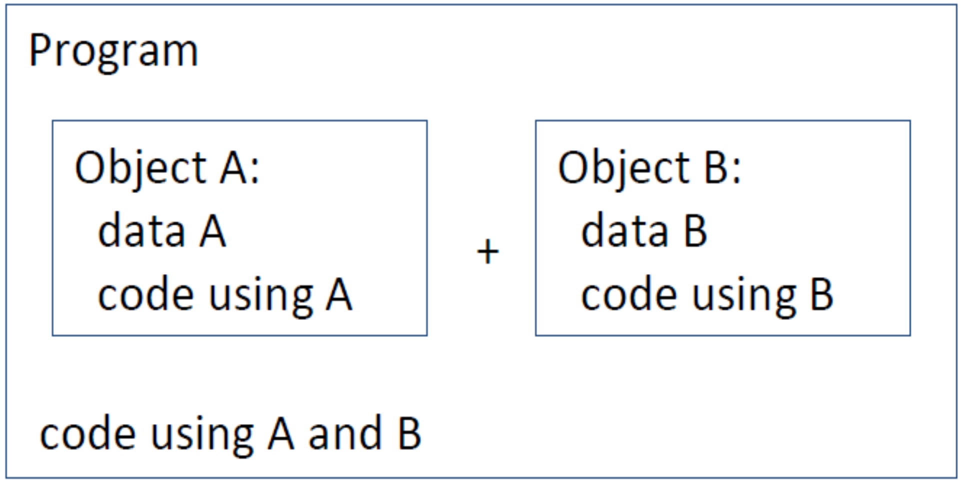
Let's map out programming paradigms, mindful of the fact that these are fuzzily defined.

Paradigm	Description	Examples
Imperative	A defined sequence of commands for a computer to perform, directly changing the state (all stored information) e.g. featuring `goto` statements.	BASIC
Functional	Programs are like mathematical functions: they cannot change the state.	Lisp, Anglican
Structured	Improves upon imperative approach by adding loops, subroutines, block structures. Procedures are still separated from data.	Matlab, C
Object-Oriented	Data, and procedures for acting upon them (methods), are united in objects.	C++, Java

Topic 2: Foundational Concepts in Object-oriented Programming

In structural programming, structures contain only data, and we separately create functions to act on them. Objects contain both data and functions (methods) to operate upon them.



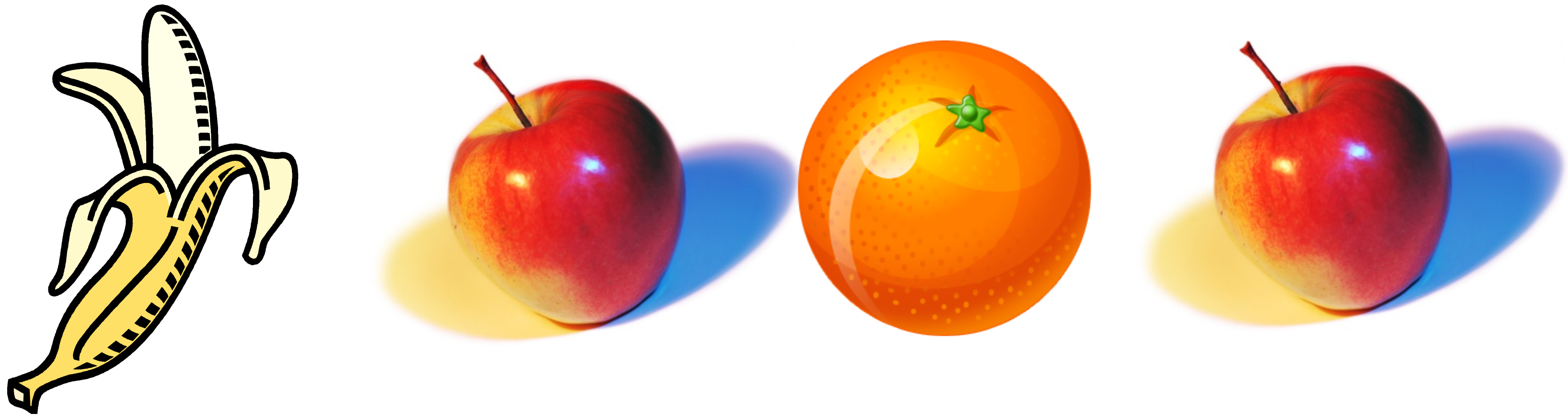


Structural programming

Object-oriented programming

An object is an instance of a class, declared for use.

Fruit



An object interface defines how an object can be interacted with, providing an explicit separation of how an object is used from the implementation details.

Classes are a mechanism for building compound data structures i.e. *user-defined* types.

Like C's struct, C++ provides a compound data structure to encapsulate related data into a single "thing": a class.

C++ goes further by also allowing a class to contain functions on the data (methods).

C++ also allows a means to control access to the components (the data and methods) of the class (private and public keywords). This is important in creating a well-defined interface for an object (i.e. defining the ways in which other objects and code can use and interact with the object).

Matlab also supports classes.

C++ predefines a set of atomic types (primitives) e.g. bool, char, int, float, double. These cannot be broken down further.

C++ libraries also define non-atomic types, such as strings (comprised of chars).

C++ provides mechanisms so that user-defined types (classes) can behave like the predefined non-atomic types.

C++ uses static typing, as you've seen in B16: Structured Programming.

A class is a user-defined data type which encapsulates related data into a single entity.

It defines how an object of this type will look (and behave).

The data elements that make up the class are known as fields.

```
class Complex {
  public:
    double re, im;
};
```

Class definition

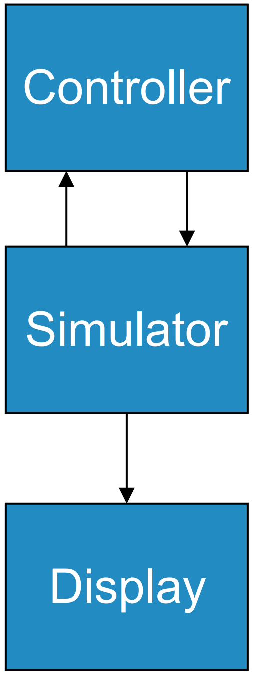
Fields

Don't confuse with creating an instance (i.e. declaring)

```
int i;
Complex z;
```

Create an object (an instance) of this type

♣ Example: Vertical take-off and landing (VTOL) aircraft state.



Let’s represent the current state as, say, a triple of numbers and a bool:
(position, velocity, mass, landed).

```
class State {
  public:
      double pos, vel, mass;
      bool landed;
};

State s;
```

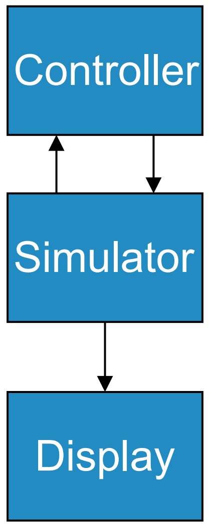
♣ Example: Vertical take-off and landing (VTOL) aircraft state.

We could represent the state as four separate variables.

However this would not capture the conceptual relationship between the variables, and would result in code that has a more complicated interface.

The controller would need to take 4 input variables instead of one! Likewise the simulator output would be four quantities.

This would be harder to understand and less close to our data flow diagram.



```
class State {
  public:
    double pos, vel, mass;
    bool landed;
};

State s;
```

Class members are accessed using the member selection operator (which is a dot).

```
State s;

s.pos = 1.0;
s.vel = -20.0;
s.mass = 1000.0;
s.landed = false;

s.pos = s.pos +
  s.vel*deltat;

Thrust =
  ComputeThrust(s);
```

s is an instance of type (class) State.

Structures in Matlab are very similar: but don’t require initialisation. You can jump straight to:

```
s.pos = 1.0;
s.vel = -20.0;
…
```

Class members are accessed using the member selection operator (which is a dot).

```
State s;

s.pos = 1.0;
s.vel = -20.0;
s.mass = 1000.0;
s.landed = false;

s.pos = s.pos +
  s.vel*deltat;

Thrust =
  ComputeThrust(s);
```

We'll see later that this kind of access to data members is discouraged in C++. Instead the better practice is have data declared private and accessed only through the class' methods.

An aside: in C++, we print to screen as per

```
cout << "Hello world!" << endl;
```

`cout` is a special object that represents the screen/terminal.

`<<` is the output operator. Anything sent to `cout` by the << operator will be printed on the screen.

`endl` (‘end line’) moves the cursor to the next line.

Recall that in C++ a class encapsulates related data and functions that operate on the data.

A class member function is called a method.

Methods (like data fields) are called using the dot operator.

```
class Complex {
   public:
      double re, im;

      double Mag() { return sqrt(re*re + im*im); }
      double Phase() { return atan2(im, re); }

};

Complex z;
cout << "Magnitude=" << z.Mag() << endl;
```

Notice that the fields `re` and `im` are used without the `z.` within the methods. When a method is invoked, it is invoked by a specific object (in this case `z`), so we already know which object's fields are being referred to; the `z.` is implicit.

`re` and `im` do not need `z.`, which is implicitly passed to the function via a special pointer (`this`).

```
class Complex {
   public:
      double re, im;

      double Mag() { return sqrt(re*re + im*im); }
      double Phase() { return atan2(im, re); }

};

Complex z;
cout << "Magnitude=" << z.Mag() << endl;
```

When an object's method is called, a pointer to the calling object is always passed in as a parameter. In the rare event that you need access to the whole object, you can do so via the `this` pointer.

Let’s recap pointers and references.

`int * p = & n;`	Define pointer `p` as the address where integer `n` is stored.
`& n`	The address (in memory) of variable `n`.
`* p`	Dereference pointer `p` (giving the contents of the pointed-to memory).
`int & r = n;`	Define reference `r` as “another name” for integer `n`. It provides a dereferenced pointer to the address of `n` (that can’t be redirected).

A reference gives “another name” for a variable. It provides a dereferenced pointer to the address of the variable (that can’t be redirected).

```
int nasty_long_name = 3;
int & r = nasty_long_name ;
// r is a reference to nasty_long_name

nasty_long_name = 2;
// nasty_long_name is now 2
r = 7;
// nasty_long_name is now 7

cout << nasty_long_name ; // prints 7
nasty_long_name++;
cout << r; // prints 8
```

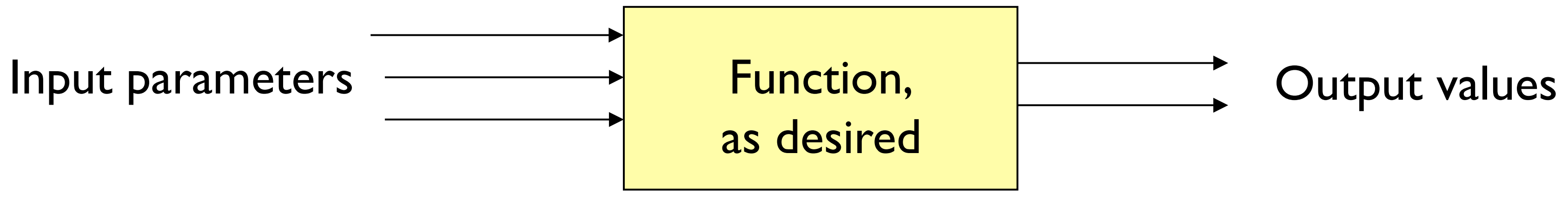
Topic 3: Encapsulation

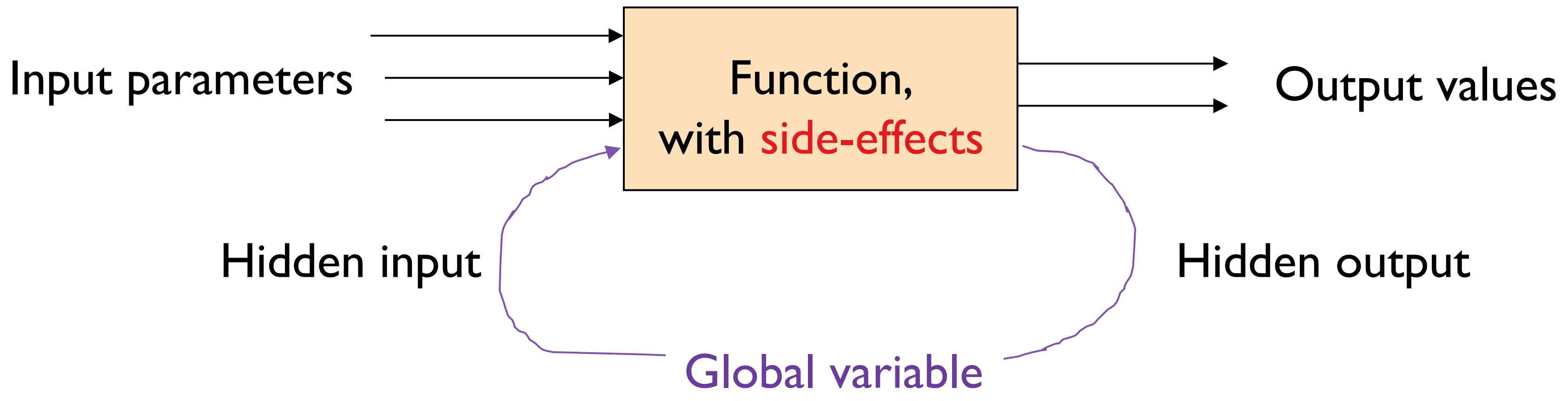
Information hiding / data hiding / encapsulation is the ability to make object data available only on a “need to know” basis.

Encapsulation has made OOP the programming paradigm of choice for large projects.

By specifying the object interfaces clearly in the design phase of a project, teams of programmers can then implement the components with some assurance that the components will all work together.

Recall side-effects, which break the intended function-like semantics of our programs.





Encapsulation means that software components hide the internal details of their implementation.

In procedural programming, we treat a function as a black box with a well-defined interface, and use these functions as building blocks to create programs.

We then need to take care to avoid side-effects.

In object-oriented programming, a class defines a black box data structure, which has:

1. a public interface;
2. private data.

Other software components in the program can only access the class through a well-defined interface, minimising side-effects.

The private data fields (`re` and `im`) of an instance of the Complex class cannot be accessed by other software components.

```
class Complex {
   public:
         double Re() { return re; }
         double Im() { return im; }
         double Mag() { return sqrt(re*re + im*im);}
         double Phase() { return atan2(im, re);
      }
   private:
         double re, im;
};
```

Private members can be accessed only through the public interface: the (read-only) accessor methods `Re()` and `Im()`.

```
Complex z;
cout << "Real part=" << z.Re() << endl;
cout << "Magnitude=" << z.Mag() << endl;
```

This may seem like we are adding an unnecessary layer of complexity, but we do so in order that the interface of the object is as unambiguously defined as possible.

```
class Complex {
   public:
         double Re() { return re; }
         double Im() { return im; }
         double Mag() { return sqrt(re*re + im*im);}
         double Phase() { return atan2(im, re);
      }
   private:
         double re, im;
};

Complex z;
cout << "Real part=" << z.Re() << endl;
cout << "Magnitude=" << z.Mag() << endl;
```

Code that used “z.re” or “z.im” would produce an error.

Here, we have changed the internal representation, but the interface has remained unaltered: other components are unaffected.

```
class Complex {
   public:
         double Re() { return r*cos(theta); }
         double Im() { return r*sin(theta); }
         double Mag() { return r;}
         double Phase() { return theta; }
      }
   private:
         double r, theta;

};
```

Internal implentation now in polar coords

```
Complex z;
cout << "Magnitude=" << z.Mag() << endl;
cout << "Real part=" << z.Re() << endl;
```

Unchanged!!

This is the essence of encapsulation: the interface captures all that is required for other program components to use the class.

```
class Complex {
   public:
         double Re() { return r*cos(theta); }
         double Im() { return r*sin(theta); }
         double Mag() { return r;}
         double Phase() { return theta; }
      }
   private:
         double r, theta;

};
```

Internal implentation now in polar coords

```
Complex z;
cout << "Magnitude=" << z.Mag() << endl;
cout << "Real part=" << z.Re() << endl;
```

Unchanged!!

Topic 4: Constructors

Recall our definition of the Complex class.

```
class Complex {
   public:
         double Re() { return re; }
         double Im() { return im; }
         double Mag() { return sqrt(re*re + im*im);}
         double Phase() { return atan2(im, re);
      }
   private:
         double re, im;
};

Complex z;
cout << "Magnitude=" << z.Mag() << endl;
cout << "Real part=" << z.Re() << endl;
```

Code that used "z.re" or "z.im" would produce an error.

As `re` is private, we can no longer say

```
z.re = 1.0;
```

so how can we get values into an object?

Whenever a variable is created (declared),

```
int i;
```

1. at compile time, the code is checked to ensure data is used correctly;
2. at run time, memory space is allocated for the variable, creating an object.

However the above will not initialise the variable. Instead, we need to do e.g.

```
int i = 10;
```

In general, this is the job of the constructor function.

For the predefined types, the constructor is automatically defined and so we never mention it e.g.

```
   int i = 10;
```

In general, we can initialise using a constructor, a member function of the class:

```
   int i(10);
```

A constructor gives the programmer control over what happens when a user-defined type is created.

The constructor is a special function with the same name as the class and no return type. It must be defined in the class definition like any other method.

```
Complex(double x, double y) {
 re = x; im = y;
}
```

The constructor must be called once each time an instance (or object) of the class is created.

An initialisation list is an implicit way of making a constructor using other constructors.

```
class Complex {
 public:
  Complex(double x, double y) : re(x), im(y)
      {}
 private:
  double re, im;
};
```

using constructors of double class

NB: no semicolon

Although a little less readable, this approach is generally preferred as it allows

1. the initialisation of const variables,
2. the initialisation of reference member variables,
3. composition and
4. inheritance (more on all these later).

Let’s add a constructor to the interface for Complex.

```
class Complex {
   public:
         Complex(double x, double y) { re = x; im = y; }
         double Re() { return re; }
         double Im() { return im; }
         double Mag() { return sqrt(re*re + im*im);}
         double Phase() { return atan2(im, re);
      }
   private:
         double re, im;
};

Complex z(10.0, 8.0);
cout << “Magnitude=“ << z.Mag() << endl;
cout << “Real part=“ << z.Re() << endl;
```

When the constructor is called, it receives parameters 10.0 and 8.0. The code in the constructor sets the value of `re` to be the first formal parameter (x) and the value of `im` to be the second formal parameter (y). Hence we have declared (instantiated) an object of type `Complex`, with value 10 + j 8.

Changing to polar coordinates requires changing the workings of methods, including the constructor. The interface is unchanged.

```
class Complex {
   public:
         Complex(double x, double y) {
            r = sqrt(x*x + y*y);
            theta = atan2(y,x);
         }
         double Re() { return r*cos(theta); }
         double Im() { return r*sin(theta); }
         double Mag() { return r;}
         double Phase() { return theta; }
      }
   private:
         double r, theta;

};
```

```
Complex z(10.0,8.0);
cout << "Magnitude=" << z.Mag() << endl;
cout << "Real part=" << z.Re() << endl;
```

The internal implentation is now in polar coords.

The interface is unchanged! Hence all other code using `Complex` need not be altered.

We can define our constructor to supply default values for data.

```
class Complex {
 public:
  Complex(double x = 0.0, double y = 0.0);
            { re = x; im = y; }
 private:
  double re, im;
};
```

Hence our constructor can act as a default constructor in the absence of some or all input parameters.

```
class Complex {
 public:
  Complex(double x = 0.0, double y = 0.0);
            { re = x; im = y; }
 private:
  double re, im;
};
```

```
Complex cDefault; // will call Complex(0, 0)
Complex cSix(6); // will call Complex(6, 0)
Complex cFivePlusThreej(5,3); // will call
Complex(5,3)
```

The copy constructor is a particular constructor that takes as its single argument an instance of the class; typically, it copies its data into the new instance.

```
Complex(Complex& z) {
 re = z.Re(); im = z.Im();
}
```

```
Complex(Complex& z) : re(z.Re()), im(z.Im()) {}
```

The compiler creates a copy constructor by default, but it’s not always what we want.

We need to take extra care when dealing with objects that contain dynamically allocated components.

For example, if an object has a pointer as a field, the memory address will be copied by default, rather than the contents.

Hence any change to the contents will affect both the original and the copy.

In C++, parameters are usually passed by value. This means a copy (requiring a call to the copy constructor) of the parameter value is put on the stack.

```
void foo(int x) {...} // passing by value
```

Alternatively, you can pass by reference, meaning that a reference is placed on the stack.

```
void foo(int & x) {...}
                 // passing by reference
```

Note the copy constructor is passed its parameter z by reference.

Were z to be passed instead by value, a copy of z would have to be placed on the stack. To make that copy, the copy constructor would be called. Which means you'd have to pass another z by value, which would mean another copy would have to be made etc.

```
Complex(Complex& z){
 re = z.Re(); im = z.Im();
}
```

```
Complex(Complex& z) : re(z.Re()), im(z.Im())
{}
```

This must be a reference to `z` (or else infinite recursion results!).

Note that we have now defined two (constructor) functions with the same name. This is called overloading, and it’s possible because the compiler looks at both the name of the function and the types of the arguments when deciding which function to call. More later!

Cartesian

```
class Complex {
  public:
      Complex(double x,
             double y)…
      Complex(Complex& z)…

      double Re()  …
      double Im()  …
      double Mag()  …
      double Phase()  …
  private:
      double re, im;
};
```

Polar

```
class Complex {
   public:
         Complex(double x,
                double y)…
         Complex(Complex& z)…

         double Re()  …
         double Im()  …
         double Mag()  …
         double Phase()  …
   private:
         double r, theta;
};
```

Both

```
Complex z(10.0,8.0);
cout << “Magnitude=“ << z.Mag() << endl;
cout << “Real part=“ << z.Re() << endl;
```

Topic 5: Implementation and Interface

In C++ programs, the header file (.h) plays a much more important role than in C. Every program element that uses the Complex class needs to know its interface, specified in the header.

Complex.h:

```
class Complex {
  public:
      Complex(double x, double y);
      double Re();
      double Im();
      double Mag();
      double Phase();

  private:
      double re, im;
};
```

The implementation of each method in a class appears in the .cpp file.

In order to understand how to use a class, the programmer usually doesn’t need to look at the .cpp file (the implementation); instead, they can simply look at the uncluttered header file defining the interface.

Whenever a programmer wants to use the Complex class, they can simply import the header into the code via the

```
#include “Complex.h”
```

compiler directive, and the compiler knows everything it needs to know about legal uses of the class.

It is the job of the linker to join the implementation details of the class (that lives in Complex.cpp) with the rest of the code.

The implementation of each method in the Complex class appears in the .cpp file.

The class scoping operator "::" is used to inform the compiler that each function is a method belonging to class Complex.

Complex.cpp

```
#include "Complex.h"

Complex::Complex(double x, double y) {
   re = x; im = y;
}
double Complex::Re() { return re; }
double Complex::Im() { return im; }
double Complex::Mag() {
   return sqrt(re*re+im*im);
}
double Complex::Phase() { return atan2(im,re); }
```

C/C++ parameters are passed by value (i.e. a copy of the parameter value is put on the stack).

```
void foo(int x) {...} // passing by value
```

Arrays are an exception: they are passed by reference, meaning that a reference is placed on the stack.

```
void foo(int & x) {...}
                     // passing by reference
```

Arrays can be very big, and hence putting a copy of the whole array onto the stack can take much processing and be wasteful of stack space.

Instead, it is cheap both in memory and processing to put a reference onto the stack, hence the exception. Another exception is found in large classes, for similar reasons.

The problem with passing by reference is that there is nothing to prevent the code in a function changing values of x, causing a side-effect.

```
void FooPassByVal(int x) { // passing by value
     x++;
}

void FooPassByRef(int &x) { // passing by reference.
      x++;
}
```

When `FooPassByRef(y)` is called, its internal `x` becomes a reference to `y`. Hence whatever is done to `x` is also done to `y`!

Below we see an example of a side-effect caused by passing by reference. Maybe that’s what we want, but usually it’s not!

```
void FooPassByVal(int x) { // passing by value
     x++;
}

void FooPassByRef(int &x) { // passing by reference.
     x++;
}

int main() {
     int x = 3;
     FooPassByVal(x);
     cout << x << endl; // prints 3
     FooPassByRef(x);
     cout << x << endl; // prints 4
     return 0;
}
```

As a solution, an object (variable) can be declared as `const`, meaning the compiler will complain if its value is ever changed.

This is the standard way of passing large objects into a function.

```
const int i(44);
i = i+1;  /// compile time error!
```

It is good practice to declare constants explicitly. It is also good practice to declare formal parameters `const` if the function does not change them.

```
int foo(BigClass& x);
```

versus

```
int foo(const BigClass& x);
```

In C++, `const` plays an important part in defining the class interface.

Class member functions can (and sometimes must) be declared as `const.` This means they do not change the value of the calling object, which is enforced by the compiler.

Notice that as far as code in a class member function is concerned, data fields are a bit like global variables: they can be changed in ways that are not reflected by the function prototype. The use of `const` can help to control this.

We might declare a Complex variable z to be const. This would be a sensible thing to do if its value never changes, and will mean that the compiler will ensure its value can't be changed.

```
class Complex {
    public:
            Complex(double x, double y) { re = x; im = y; }
            double Re() { return re; }
            double Im() { return im; }
            double Mag() { return sqrt(re*re + im*im);}
            double Phase() { return atan2(im, re);
        }
    private:
            double re, im;
};
```

```
const Complex z(10.0, 8.0);
```

However, this can often lead to difficult-to-find compile-time errors e.g. the code below.

```
class Complex {
   public:
         Complex(double x, double y) { re = x; im = y; }
         double Re() { return re; }
      }
   private:
         double re, im;
};

const Complex z(10.0, 8.0);
cout << "Real part=" << z.Re() << endl;
```

Error!

The reason is that the compiler doesn't analyse the semantics of each function, so it can't be sure that the accessor function `z.Re()` doesn't change the internal values (re=10.0 and im=8.0) of the object z.

The way to avoid this is to declare the accessor methods as `const` functions.

To fix this ‘problem’, we add `const` after the function name for functions that don’t change the value; these are const methods, telling the programmer and the compiler that these functions do not modify any of the object’s data.

```
class Complex {
   public:
         Complex(double x, double y) { re = x; im = y; }
         double Re() const { return re; }
         double Im() const { return im; }
         double Mag() const { return sqrt(re*re + im*im);}
         double Phase() const { return atan2(im, re);
      }
   private:
         double re, im;
};
```

```
const Complex z(10.0, 8.0);
cout << “Real part=” << z.Re() << endl;
```

Code should now compile

Faced with an error like above, you may be tempted to make all data public (and not use `const`). Don't't! The grief caused by doing things 'properly' will be minor compared to the potential problems created by code that abuses the interface.

```
class Complex {
   public:
          double re, im;
          Complex(double x, double y) { re = x; im = y; }
          double Re() { return re; }
          double Im() { return im; }
          double Mag() { return sqrt(re*re + im*im);}
          double Phase() { return atan2(im, re);
       }
};
```

Topic 6: Functions and Operators

C++ allows several functions to share the same name, but accept different argument types: this is function overloading.

```
void foo(int x);

void foo(int &x, int &y);

void foo(double x, const Complex c);
```

The function name and types of arguments together yield a signature that tells the compiler if a given function call in the code is valid, and which version is being referred to.

As an example, we can define (the already defined) `exp` for the `Complex` class.
NB: exp(x + i y) = exp(x) exp(i y)

```
#include <cmath>

Complex exp(const Complex z)
{
   double r = exp(z.Re());
   Complex zout(r*cos(z.Im()),
                r*sin(z.Im()));
   return zout;
}
```

When should we use a member function and when should we use a non-member function?

That is, when should we define the non-member `exp(z)` and when the member `z.exp()`?

Consider carefully how it will be used. Does it modify the instance? If so, a non-member function is preferable. Which presents the most natural interface?

For this example, I would go for `exp(z)`: instance modification seems more natural.

Note that non-member functions are a slight abandonment of the OOP paradigm.

As you know, C++ (and Matlab etc.) provide operators (functions, except with disallowed names and/or different calling syntax).

Arithmetic	`+  -  *  /  %`

Relational	`==  !=`
	 `<  >  <=  >=`

Boolean	`&&  ||  !`
Assignment	`=`
I/O streaming	`<<  >>`

C++ aims for user-defined types to mimic, as far as possible, the predefined types.

Suppose we want to add two Complex variables together. We could create a function:

```
Complex Add(const Complex z1, const Complex z2) {
   Complex zout(z1.Re()+z2.Re(),
                z1.Im()+z2.Im());
   return zout;
}
```

However, it would be much cleaner to use the + operator to write:

```
Complex z3;
z3 = z1+z2;
```

We can achieve this by operator overloading.

In C++, the infix notation, e.g. **`a+b`**, is defined as a shorthand for a function expressed using prefix notation e.g. **`operator+(a,b)`**.

Since an operator is just a function, we can overload it:

```
Complex operator+(const Complex z1,
                  const Complex z2) {
  Complex zout(z1.Re()+z2.Re(),
               z1.Im()+z2.Im());
  return zout;
}
```

Hence we can write

```
Complex z3;
z3 = z1 + z2;
```

The assignment operator = is used to copy the values from one object to another already defined object.

Note that if the object is not already defined, the copy constructor is called instead.

```
Complex z1, z2, z3;
z3 = z1 + z2; // assignment operator

// the copy constructor is called for
both of the following
Complex z4(z3);
Complex z5 = z1 + z2;
```

The assignment operator = is used to copy the values from one object to another already defined object.

```
Complex z1, z2, z3;
z3 = z1 + z2;
```

This is the definition of the assignment operator = for the Complex class.

```
Complex& Complex::operator=(const Complex& z1)
{
    re = z1.Re();
    im = z1.Im();
    return *this;
}
```

The definition of the operator = is one of the few common uses for the `this` pointer. Here it is dereferenced and returned.

`z2 = z1` is shorthand for `z2.operator=(z1)`.

`operator=` must be a member function.

The left hand side (here `z2`) is implicitly passed in to this function.

The return value of `z2.operator=(z1)` is a reference to `z2`. That way we can concatenate assignments as in `z3=z2=z1` which assigns the value of `z1` to `z2` and to `z3`, as

`(z3 = (z2 = z1))` which is

```
z3.operator=(z2.operator=(z1));
```

C++ does no array bounds checking (leading to possible segmentation faults). Let's create our own safe array class by overloading the array index operator `[]`.

```
class SafeFloatArray {
   public:
      …
      float & operator[](int i) {
         if ( i<0 || i>=10 ) {
            cerr << "Oops!" << endl;
            exit(1);
         }
         return a[i];
      }

   private:
      float a[10];
};
```

Defines a 10-long array of floats

Returns a reference to a float.

We can use this array as:

```
SafeFloatArray s;

s[0] = 10.0;
s[5] = s[0]+22.4;
s[15] = 12.0;
```

This compiles but at runtime the last call generates the error message "`Oops!`" and the program exits.

Topic 7: A Complete Complex Example

Let’s bring it all together by creating program to calculate the frequency response of the transfer function H(jw) = 1/(1+jw).

We’ll need three files, two modules:

1. Complex.h and Complex.cpp
2. Filter.cpp

The first two files define the Complex interface (.h) and the method implementations (.cpp), respectively.

```
% g++ -c Complex.cpp
% g++ -c Filter.cpp
% g++ -o Filter Complex.o Filter.o -lm
```

Compile source to object files (`.o`)

Link object files together with maths library (-`lm`) to create executable

This is Complex.h and defines the interface. Any program that wants to use the Complex class should include this file: #include “Complex.h”.

```
// Complex.h
// Define Complex class and function prototypes
//

class Complex {
 public:
  Complex(const double x=0.0, const double y=0.0);
  double Re() const;
  double Im() const;
  double Mag() const;
  double Phase() const;
  Complex &operator=(const Complex z);

 private:
  double _re, _im;
};

// Complex maths
Complex operator+(const Complex z1, const Complex z2);
Complex operator-(const Complex z1, const Complex z2);
Complex operator*(const Complex z1, const double r);
Complex operator*(const double r, const Complex z1);
Complex operator*(const Complex z1, const Complex z2);
Complex operator/(const Complex z1, const Complex z2);
```

Note that default values are supplied only in the header.

The underscore is just used to denote a private field (it’s just an arbitrary part of a name).

This is `Complex.cpp`, containing the implementation of the various methods in the class interface `Complex.h`.

```
#include <cmath>
#include <iostream>
#include "Complex.h"

// First implement the member functions
// Constructors
Complex::Complex(const double x, const double y) : _re(x), _im(y) {}
Complex::Complex(const Complex& z) : _re(z.Re()), _im(z.Im()) {}

double Complex::Re() const { return _re; }
double Complex::Im() const { return _im; }
double Complex::Mag() const { return sqrt(_re*_re + _im*_im); }
double Complex::Phase() const { return atan2(_im, _re); }

// Assignment
Complex& Complex::operator=(const Complex z)
{
  _re = z.Re();
  _im = z.Im();
  return *this;
}
```

We include maths (so we can use e.g. `sqrt`) and input/output (so we can use e.g. `cout`) libraries.

A library is just a pre-packaged bundle of useful code.

```
// Complex .cpp continued

// Now implement the non-member arithmetic functions
// Complex addition
Complex operator+(const Complex z1, const Complex z2)
{
  Complex zout(z1.Re()+z2.Re(), z1.Im()+z2.Im());
  return zout;
}
// Complex subtraction
Complex operator-(const Complex z1, const Complex z2)
{
  Complex zout(z1.Re()-z2.Re(),
               z1.Im()-z2.Im());
  return zout;
}

// scalar multiplication of Complex
Complex operator*(const Complex z1, const double r)
{
  Complex zout(r*z1.Re(), r*z1.Im());
  return zout;
}

Complex operator*(const double r, const Complex z1)
{
  Complex zout(r*z1.Re(), r*z1.Im());
  return zout;
}
```

Overloading to be able to put the scalar in either input.

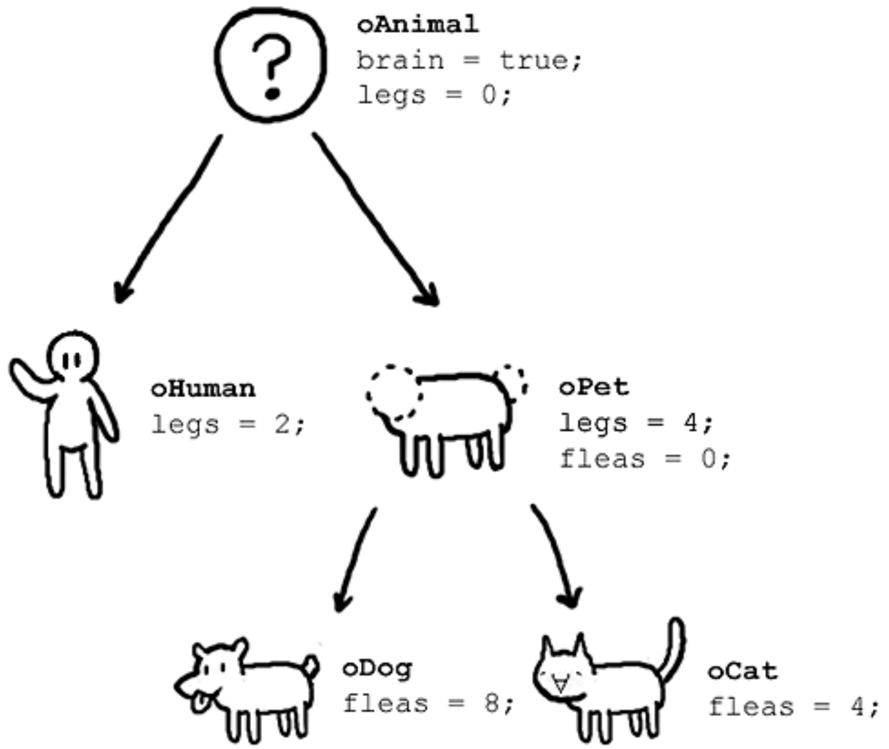
```
// Complex multiplication
Complex operator*(const Complex z1, const Complex z2)
{
  Complex zout(z1.Re()*z2.Re() - z1.Im()*z2.Im(),
               z1.Re()*z2.Im() + z1.Im()*z2.Re());
  return zout;
}

// Complex division
Complex operator/(const Complex z1, const Complex z2)
{
  double denom(z2.Mag()*z2.Mag());
  Complex zout((z1.Re()*z2.Re() + z1.Im()*z2.Im())/denom,
               (z1.Re()*z2.Im() - z1.Im()*z2.Re())/denom);
  return zout;
}


// end of file Complex.cpp
```

`main.cpp` contains code that uses the Complex class.

```
#include <iostream>
#include "Complex.h"

using namespace std;

Complex H(double w)
{
  const Complex numerator(1.0);
  const Complex denominator(1.0, 0.1*w);
  Complex z(numerator/denominator);
  return z;
}


int main(int argc, char *argv[])
{
  double w=0.0;
  const double stepsize=0.01;
  Complex z;

  for (double w=0.0; w<100.0; w+=stepsize) {
    z = H(w);
    cout << w << " " << z.Mag() << " " << z.Phase() << endl;
  }
}
```

Topic 8: Inheritance and Composition

Inheritance is the ability to create class hierarchies, such that the inheriting (or child, sub-, derived) class is an instance of the ancestor (or base, super-, parent) class.



A class in C++ inherits from another class using the : operator in the class definition.

Hierarchical relationships often arise between classes. Object-oriented design supports this through inheritance.

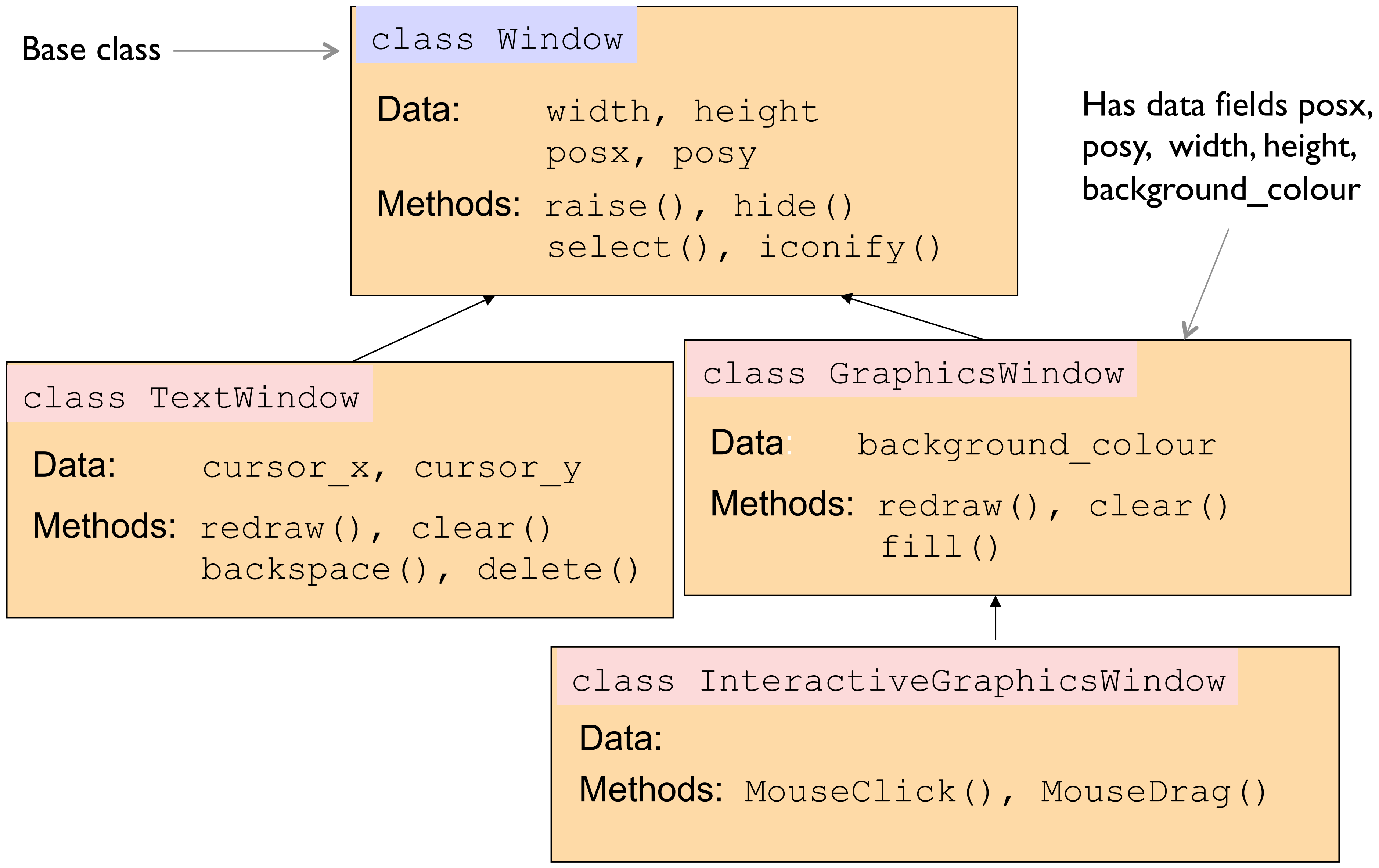
An derived class is one that has the functionality of its parent class but with some extra data or methods.

In C++

```
    class A : public B {
    …
    };
```

The code above reads “class A inherits from class B”, or “class A is derived from class B”. Inheritance encodes an “is a” relationship.

♣ Example: Windows.



In C++, you can use the `protected` access specifier to limit access to member functions of derived classes, and those of the same class.

Previously, we used `public` for universal access and `private` to limit access to member functions from the same class.

`protected` is a relaxed version of `private` in which derived classes are also granted access.

```
class Window{
protected:
    double width, height;
    };
```

```
class TextWindow : public Window {
public:
    double GetTextWidth() const {return 0.9 * width;}
    ...
    };
```

Inheritance allows code re-use without the dangers of copy-and-pasting.

Copy-and-pasting results in having to maintain multiple copies of the same code: improvements or bug-fixes have to be applied to each copy separately. It's also easy to forget that some copies even exist!

Inheritance allows you to re-use code without these dangers: any changes to the parent class are automatically propagated to all derived classes.

Each derived class need only be defined by its difference from the parent class, rather than having to redefine everything: this leads to shorter, neater, code.

Inheritance is an “is a” relationship.

Every instance of a derived class is also an instance of the parent class e.g.

```
class Vehicle;
class Car : public Vehicle { … };
```

Every instance of a `Car` is also an instance of a `Vehicle.`

A `Car` object has all the properties of a `Vehicle.`

Composition is an “has a” relationship.

`Wheels`, `Light` and `Seat` are all (small, simple) classes in their own right.

```
class Vehicle {
private:
   Wheels w;
   Light frontLight;
   Light backLight;
   Seat s;
};
```

Composition allows for abstraction and modularity.

Each class can be kept simple and dedicated to a single task, making it easier to write and debug.

Using composition, some classes that may not be very useful on their own may nonetheless be re-usable within many other classes e.g. `Wheels`. This encourages code re-use.

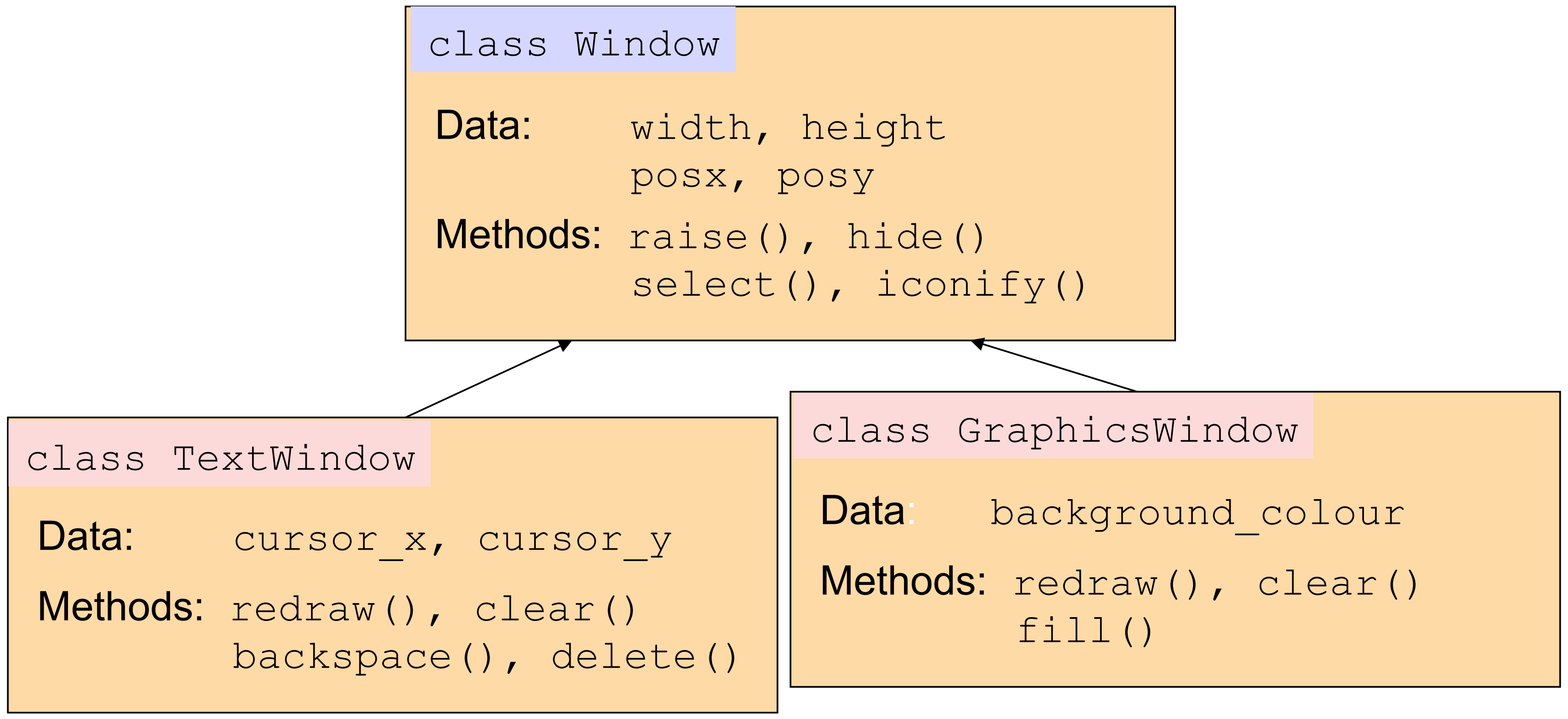
Topic 9: Polymorphism

Polymorphism is the ability of objects in the same class hierarchy to respond in tailored ways to the same events. It allows us to hide alternative implementations behind a common interface.

"Now Speak!"
"Quack"
"Woof"
"Meow"

Polymorphism is Greek for "many forms".

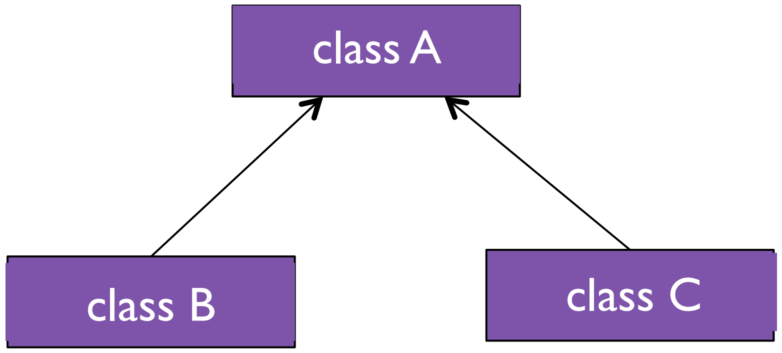
Both `TextWindow` and `GraphicsWindow` have a method `redraw()`, but redrawing a `TextWindow` is different from redrawing a `GraphicsWindow`. Polymorphism permits the two different window types to share the interface `redraw()` , but allows them to implement it in different ways.



Let A be a base class; B and C derive from A.

Every instance of a B object is also an A object.

Every instance of a C object is also an A object.



The call **`y.func()`** will invoke the **`func`** method belonging to class **B**, because `y` is an instance of class `B`. Likewise **`z.func()`** will call class C's **`func()`**.

```
#include <iostream>
using namespace std;

class A {
  public:
    void func() {
        cout << "A\n";
    }
};

class B : public A {
  public:
    void func() {cout<<"B\n"; }
};

class C: public A {
  public:
    void func() {cout << "C\n"; }
};
```

```
void callfunc(A param)
{
    param.func();
}

int main(int argc, char*
argv[])
{
    A x;
    B y;
    C z;

    x.func();
    y.func();
    z.func();

    callfunc(x);
    callfunc(y);
    callfunc(z);

    return 0;
}
```

callfunc(x) passes an object of type A into callfunc. The effect is to call the **func()** method of class **A**.

```
#include <iostream>
using namespace std;

class A {
  public:
    void func() {
        cout << “A\n”;
    }
};


class B : public A {
  public:
    void func() {cout<<"B\n"; }
};


class C: public A {
  public:
    void func() {cout << "C\n"; }
};
```

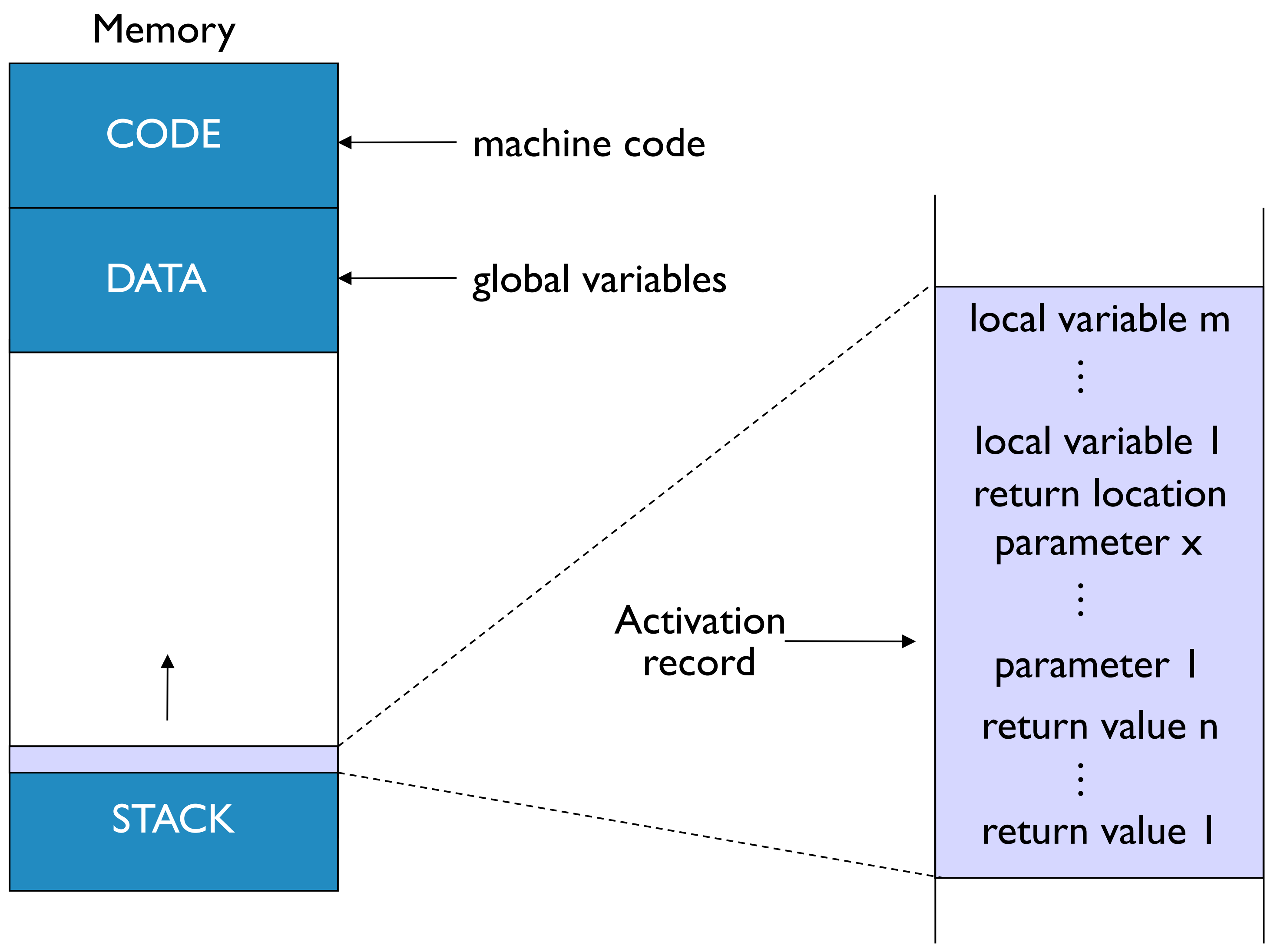
```
void callfunc(A param)
{
    param.func();
}

int main(int argc, char*
argv[])
{
    A x;
    B y;
    C z;

    x.func();
    y.func();
    z.func();

    callfunc(x);
    callfunc(y);
    callfunc(z);

    return 0;
}
```

`callfunc(y)` passes an object of type `B` into the function. Because every `B` is also an `A` this is acceptable, but only the bit of y that is an `A` is put onto the stack. The "`B` bits" are left behind, so as far as `callfunc` is concerned, it's received an object of type `A`, and `param.func()` will call **A's `func()`.**

```
#include <iostream>
using namespace std;

class A {
  public:
    void func() {
        cout << "A\n";
    }
};


class B : public A {
  public:
    void func() {cout<<"B\n"; }
};


class C: public A {
  public:
    void func() {cout << "C\n"; }
};
```

```
void callfunc(A param)
{
    param.func();
}

int main(int argc, char*
argv[])
{
    A x;
    B y;
    C z;

    x.func();
    y.func();
    z.func();

    callfunc(x);
    callfunc(y);
    callfunc(z);

    return 0;
}
```

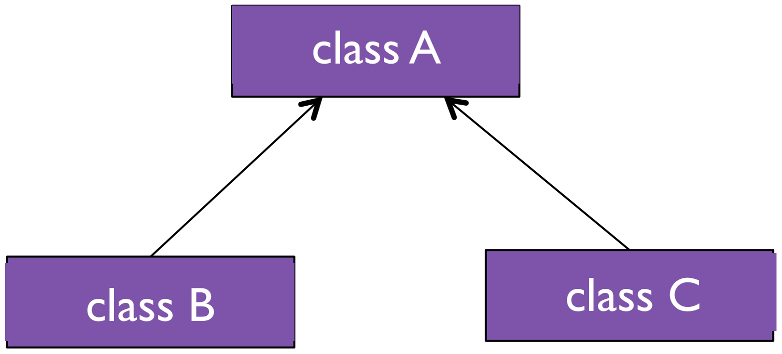
Recall that when a function is called, parameters, return location and other stuff are put onto the stack in the activation record.



Let's re-examine the previous example.

```
void callfunc(A param)
{
   param.func();
}
```

```
B y;
callfunc(y)
```

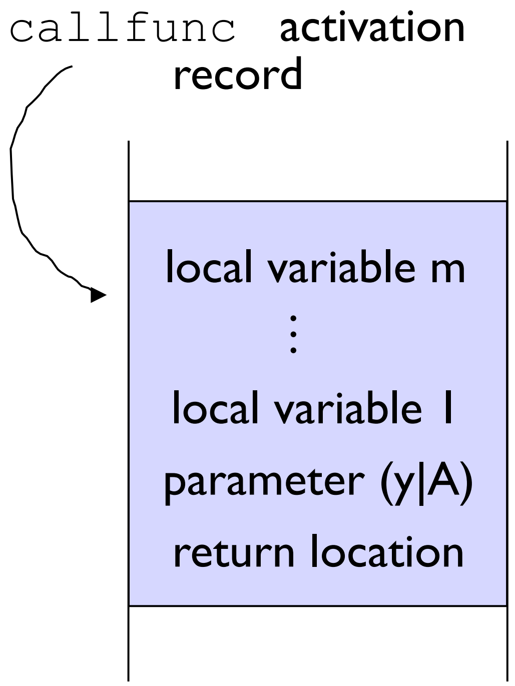


`callfunc` takes a value parameter of class `A`.

The call above is legitimate but only the bit of `y` that is an `A` will be copied onto the stack.

Once "inside" the function, the parameter can only behave as an `A`.

Note that here there is no return value because `callfunc` returns void.



So far, no polymorphism! `callfunc` treats derived classes `B` and `C` just as if they were the parent class `A`.

In C++, run-time polymorphism is invoked by the programmer via virtual functions.

```
class Window {
   …
   virtual void redraw();
};
```

`func()` in the base class `A` has been designated as a virtual function, and the parameter to `callfunc()` is now passed by reference.

```
#include <iostream>
using namespace std;

class A {
  public:
    virtual void func() {
        cout << “A\n”;
    }
};


class B : public A {
  public:
    void func() {cout<<"B\n"; }
};


class C: public A {
  public:
    void func() {cout << "C\n"; }
};
```

```
void callfunc(A& param)
{
    param.func();
}

int main(int argc, char*
argv[])
{
    A x;
    B y;
    C z;

    x.func();
    y.func();
    z.func();

    callfunc(x);
    callfunc(y);
    callfunc(z);

    return 0;
}
```

The call y.func() still invokes the func method belonging to class B because y is an instance of class B. Likewise z.func() will call class C's func().

```
#include <iostream>
using namespace std;

class A {
  public:
    virtual void func() {
        cout << “A\n”;
    }
};

class B : public A {
  public:
    void func() {cout<<"B\n"; }
};

class C: public A {
  public:
    void func() {cout << "C\n"; }
};
```

```
void callfunc(A& param)
{
    param.func();
}

int main(int argc, char*
argv[])
{
    A x;
    B y;
    C z;

    x.func();
    y.func();
    z.func();

    callfunc(x);
    callfunc(y);
    callfunc(z);

    return 0;
}
```

`callfunc(x)` passes an object of type "reference to `A`" into `callfunc`. Since `x` is of type `A`, `param.func()` calls the **`func()`** of class **`A`**.

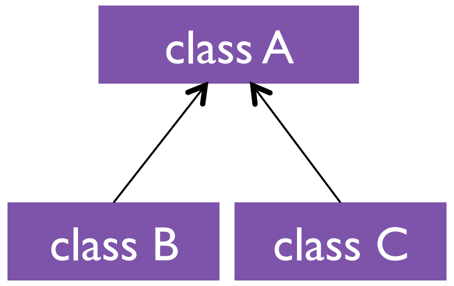
```
#include <iostream>
using namespace std;

class A {
  public:
    virtual void func() {
        cout << "A\n";
    }
};

class B : public A {
  public:
    void func() {cout<<"B\n"; }
};

class C: public A {
  public:
    void func() {cout << "C\n"; }
};
```

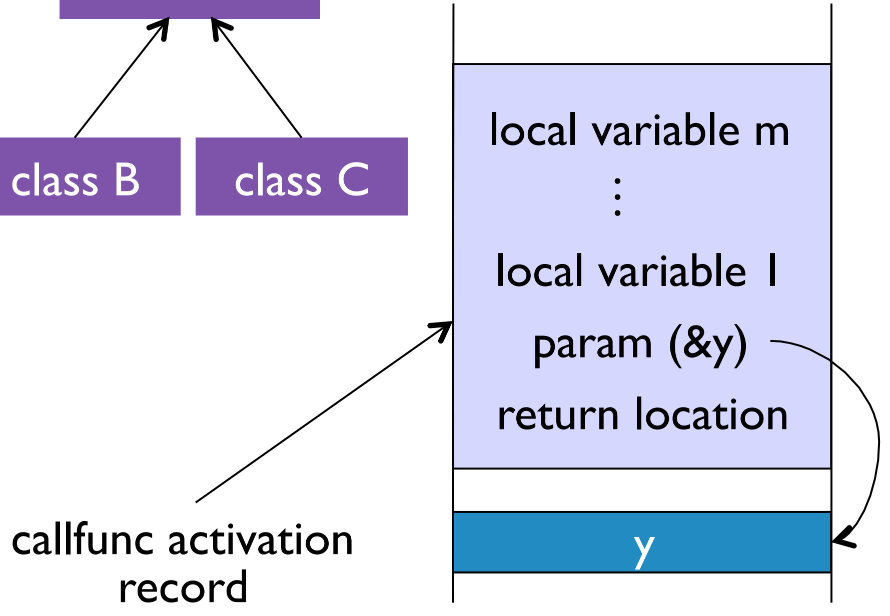
```
void callfunc(A& param)
{
    param.func();
}

int main(int argc, char*
argv[])
{
    A x;
    B y;
    C z;

    x.func();
    y.func();
    z.func();

    callfunc(x);
    callfunc(y);
    callfunc(z);

    return 0;
}
```

```
class A {
  public:
    virtual void func() {
        cout << "A\n";
    }
};
```

```
void callfunc(A& param)
{
    param.func();
}
```

```
B y;
callfunc(y)
```





`callfunc` takes a reference to an object of class `A`.
A reference is just a memory address.
The call above is legit because objects of type `B` are also of type `A`.
Dereferencing the parameter `&y` leads to `y`.
`y` can identify itself as being of class `B`, and so can behave like a `B`.
As `A.func()` was declared to be virtual, the system will therefore invoke a function call to the `func()` method belonging to class `B`.

The virtual function has allowed run-time polymorphism.

The run-time system is able to identify that the parameter `y` is in fact of type `B`. This is known as run-time type identification (RTTI).

A virtual function called from an object that is either a

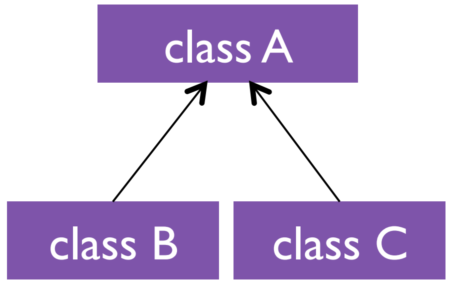
1. reference to a derived class or a
2. pointer to a derived class

performs run-time type identification on the object that invoked the call, and will call the appropriate version.

If the object is of type `A`, then call `A`'s `func()`.

If the object is of type `B`, then call `B`'s `func()`.

If the object is of type `C`, then call `C`'s `func()`.



If class `A` defines `func()` as

```
virtual void func() = 0;
```

then `A` has no implementation of `func()`.

In such a case, class `A` is called an abstract base class.

It is not possible to create an instance of class `A`, only instances of derived classes, `B` and `C`. Class `A` defines an interface to which all derived classes must conform.

We use this idea in designing program components: we specify an interface, then have a guarantee of the compatibility of all derived objects.

This code will generate a compile time error because we have tried to instantiate `A`.

```
#include <iostream>
using namespace std;

class A {
  public:
    virtual void func() = 0;
};

class B : public A {
  public:
    void func() {cout<<"B\n"; }
};

class C: public A {
  public:
    void func() { cout << "C
\n"; }
};
```

```
void callfunc(A param)
{
    param.func();
}

int main(int argc, char*
argv[])
{
    A x;
    B y;
    C z;

    x.func();
    y.func();
    z.func();

    callfunc(x);
    callfunc(y);
    callfunc(z);

    return 0;
}
```

Because `A` has a pure virtual function, denoted with the "`= 0`", it is abstract. There is no function `A.func()`. This forces any object that derives from `A` to implement `func()`.

```
#include <iostream>
using namespace std;

class A {
  public:
    virtual void func() = 0;
};

class B : public A {
  public:
    void func() {cout<<"B\n"; }
};

class C: public A {
  public:
    void func() { cout << "C
\n"; }};
```

```
void callfunc(A param)
{
    param.func();
}

int main(int argc, char* argv[])
{

    B y;
    C z;


    y.func();
    z.func();


    callfunc(y);
    callfunc(z);

    return 0;
}
```

♣ Consider a vector graphics drawing package.

The base class "`Drawable`" defines a graphics object that knows how to draw itself on the screen. Below it in the class hierarchy may be classes that define lines, curves, points etc.

```
class Drawable {
    …
    virtual void Draw() = 0;
};
class Line : public Drawable { … };
```

This forces all derived classes to implement `Draw()`.

The program will keep a list of objects that have been created and upon redraw, will display them one by one. This is implemented using a loop.

```
for (int i=0; i<N; i++) {
obj[i]->Draw();
}
```

In C++. an array can only store objects of the same type; here we use it to store pointers to (i.e. the memory address of) each object. The `->` operator does the same thing as the dot operator, but dereferences the pointer (looks up the object via the memory address) first.

♣ Example: consider a spreadsheet.

Recall that an abstract base class cannot be instantiated itself; it is used to define the interface.

We define a spreadsheet as comprising an array of references to cells; each object of type `Cell` must be able to return its own value via an `Evaluate()` method.

```
class Cell {
    virtual double Evaluate() = 0;
};

class Spreadsheet {
    private:
        Cell& c[100][100];
};
```

♣ Example: consider a spreadsheet.

By specifying the interface to the abstract base class `Cell`, we can implement `Spreadsheet` independently of the various types of `Cell` e.g. Boolean-valued expressions, dates, integers.

This allows us to decide after it's all implemented that we'd like a new type of `Cell` (i.e. a new class that inherits from `Cell`) e.g. the product of two other Cells. So long as it has a proper `Evaluate()` method, `Spreadsheet` can use it seamlessly.

```
class Cell {
    virtual double Evaluate() = 0;
};

class Spreadsheet {
    private:
        Cell& c[100][100];
};
```

♣ Example: recall Euler's method to solve the ODE $\frac{dx}{dt} = f(x,t)$. Let's implement a generic Euler method, which prints *x(t)* for *t* = 0, 1, 2, ...

```
class Euler {
  public:
    Euler(Func &f);
    void Simulate(double x, double step, double time);
  private:
    Func fn;
};
```

```
Euler::Euler(Func &f) : fn(f) {};

void Euler::Simulate(double x, double step, double time)
{
   for (int t=0; t<time; t+=step) {
      x = x + step*fn.dx_by_dt(x,t);
      cout << x << endl;
   }
   return; }
```

We now instantiate an `Euler` object (here called `e`) in which the object `y` becomes the private member `fn` of `e`.

In order for this to work, `y` (which is class `XdotPlusX`) must be an instance of `Func`. That is, `XdotPlusX` must inherit from `Func`.

```
class Euler {
  public:
    Euler(Func &f);
    void Simulate(double x, double step, double time);
  private:
    Func fn;
};
```

```
XdotPlusX y;
Euler e(y);
e.Simulate();
```

Let’s implement an abstract base class `Func`.

`Func` specifies that any object that is to be used in `Euler` must implement a `dx_by_dt` function.

Our first `dx_by_dt` function comes from the equation $\dot{x} + x = 0$, hence `dx_by_dt` returns $-x$.

```
class Func {
  public:
    virtual double dx_by_dt(double x, double t) = 0;
};

class XdotPlusX : public Func {
  public:
    double dx_by_dt(double x, double t) {
      return -x;
    }
};
```

For the non-linear equation $\dot{x} + x^2 = 0$ we can define a new class that implements `dx_by_dt = -x*x.`

```
class Func {
  public:
    virtual double dx_by_dt(double x, double t) = 0;
};

class XdotPlusX2 : public Func {
  public:
    double dx_by_dt(double x, double t) {
      return -x*x;
     }
};
```

Topic 10: Templates

Recall our array-bounds-checking class: unfortunately, it was limited to storing floats.

```
    class SafeFloatArray {
       public:
           float & operator[](int i) {
               if (i<0 || i>=10) {
                   std::cerr << "Access out of bounds\n";
                   std::exit(1);
               } else {
                   return a[i];
               }
           }
       private:
           float a[10];
    };
```

How might we create a class that can store any type we choose?

Templates provide a way to parameterize a class definition with one or more types. This is an example of compile-time polymorphism.

To implement a template in C++, you prefix the class definition with **`template <class XX>`** where `XX` is a parameter to the class definition (below we have used `varType`).

```
template <class varType>
class SafeFloatArray {
   public:
        varType & operator[](int i) {
           if (i<0 || i>=10) {
               cerr << "Access out of bounds\n";
               return varType(0);
           } else {
               return a[i];
           }
        }
   private:
       varType a[10];
};

SafeFloatArray<int> x;
SafeFloatArray<Complex> z;
```

At compile time the compiler encounters `SafeFloatArray<int> x` and creates a class definition (if it doesn't already exist) in which **`varType`** is replaced with **`int`** everywhere in the definition.

```
template <class varType>
class SafeFloatArray {
   public:
        varType & operator[](int i) {
           if (i<0 || i>=10) {
               cerr << "Access out of bounds\n";
               return varType(0);
           } else {
               return a[i];
           }
        }
   private:
        varType a[10];
};

SafeFloatArray<int> x;
SafeFloatArray<Complex> z;
```

Of course, you could do this by simply copy-and-pasting the code and doing a manual replacement of `varType`. As usual, however, copy-and-pasting code is a very bad idea!

```
template <class varType>
class SafeFloatArray {
   public:
        varType & operator[](int i) {
           if (i<0 || i>=10) {
               cerr << "Access out of bounds\n";
               return varType(0);
           } else {
               return a[i];
           }
        }
   private:
       varType a[10];
};

SafeFloatArray<int> x;
SafeFloatArray<Complex> z;
```

Here we create a template class that is a container for a templated number of elements of a templated type.

```
template <class Type, int Size>
class SafeFloatArray {
   public:
       Type& operator[](int i) {
           if (i<0 || i>=Size) {
               std::cerr << “Access out of bounds\n”;
               std::exit(1);
           } else {
               return a[i];
           }
       }
   private:
       Type a[Size];
};

SafeFloatArray<int,10> x;
SafeFloatArray<Complex,40> z;
```

Templates aid the use of similar design solutions to different problems.

The standardisation of design solutions encourages code re-use, increasing code reliability and shortening development time.

An array is special case of a container type, a way of storing a collection of possibly ordered elements e.g. list (ordered but not indexed), stack (a first-in-last-out structure), vector (extendible array), double-ended list, etc.

Templates in C++ offer a way of providing libraries to implement these standard containers.

The Standard Template Library is a suite of code that implements (among many other things) classes of container types.

These classes come with standard ways of
accessing,
inserting,
adding and
deleting elements.

They also provide a set of standard algorithms that
operate on these classes such as
searching and
iterating over all elements.

The Standard Template Library (STL) permits code re-use, as desired.

Different applications will have to represent different data, and will therefore require bespoke classes.

However, it is common for the organisation of multiple instances of bespoke classes to be done in standard ways (such as containing multiple bespoke objects in an array).

Templates allow generic, templated, code, that can be specialised to our bespoke classes at compile-time.

Consider the STL vector.

`std::vector<Type>` is an extendible array.

It can increase its size as the program needs it to.

It can be accessed like an ordinary array (eg `v[2]`).

It can report its current size

```
v.size().
```

You can add an item to the end without needing to know how big it is

```
v.push_back(x).
```

```
#include<vector>

int main() {
    std::vector<int> v;
    for (int i=0; i<20; i++) v.push_back(i);

    for (int i=0; i<v.size(); i++)
      std::cout << v[i] << std::endl;
}
```

If we were coding an extendible array class ourselves we would use dynamic memory allocation (i..e on the heap) and we would want to have:

1. An overloaded `operator[]` to access the ith element;
2. a `size()` method to report current length;
3. a `resize()` function to allocate more space;
4. a method to add an element to the end that will automatically allocate a new chunk of memory if we go beyond the end of the memory that has already been allocated.

We don't need to implement this because someone else has done it for us!

This is only possible through the use of templates, because the person who coded the vector class couldn't possibly anticipate all the classes that anyone might want to store.

Let’s create a new STL vector of a size specified at run-time.

```
std::vector<Complex> z;

int size;
std::cin >> size;
z.resize(size);

z[5] = Complex(2.0,3.0);
```

User-inputted size

Now let's create a two dimensional array at run-time.

```
int width, height;
std::vector< std::vector<int> > x;

x.resize(height);
for (int i=0; i<height; i++)
   x[i].resize(width);

x[2][3] = 10;
…
```

The vector class implements a number of methods for accessing and operating on the elements.

`vector::front`	Returns reference to first element of vector.
`vector::back`	Returns reference to last element of vector.
`vector::size`	Returns number of elements in the vector.
`vector::empty`	Returns true if vector has no elements.
`vector::capacity`	Returns current capacity (allocated memory) of vector.
`vector::insert`	Inserts elements into a vector (single & range), shifts later elements up. O(n) time.
`vector::push_back`	Appends (inserts) an element to the end of a vector, allocating memory for it if necessary. O(1) time.
`vector::erase`	Deletes elements from a vector (single & range), shifts later elements down. O(n) time.
`vector::pop_back`	Erases the last element of the vector, O(1) time. Does not usually reduce the memory overhead of the vector. O(1) time.
`vector::clear`	Erases all of the elements. (This does not reduce capacity).
`vector::resize`	Changes the vector size. O(n) time.

We often want to iterate over a collection of data, as

```
    for (int i=0; i<v.size(); i++).
```

Not all container types support indexing: a linked list has order, but only relative order.

An iterator is a class that supports the standard programming pattern of iterating over a container type.

```
std::vector<int> v;
std::vector<int>::iterator it;
for (it=v.begin(); it!=v.end(); it++)
```

An iterator encapsulates the internal structure of how the iteration occurs: it can be used without knowing the underlying representation of the container type itself.

Iterators grant flexibility: you can change the underlying container type without changing the iteration.

```
std::vector<int> v;
std::vector<int>::iterator it;
for (it=v.begin(); it!=v.end(); it++)
{
  cout << *it << endl;
}
```

An iterator is a pointer to an element of a container: this code will print the elements of `v`.

Iterators are implemented differently for each class, but the user need not know the implementation.

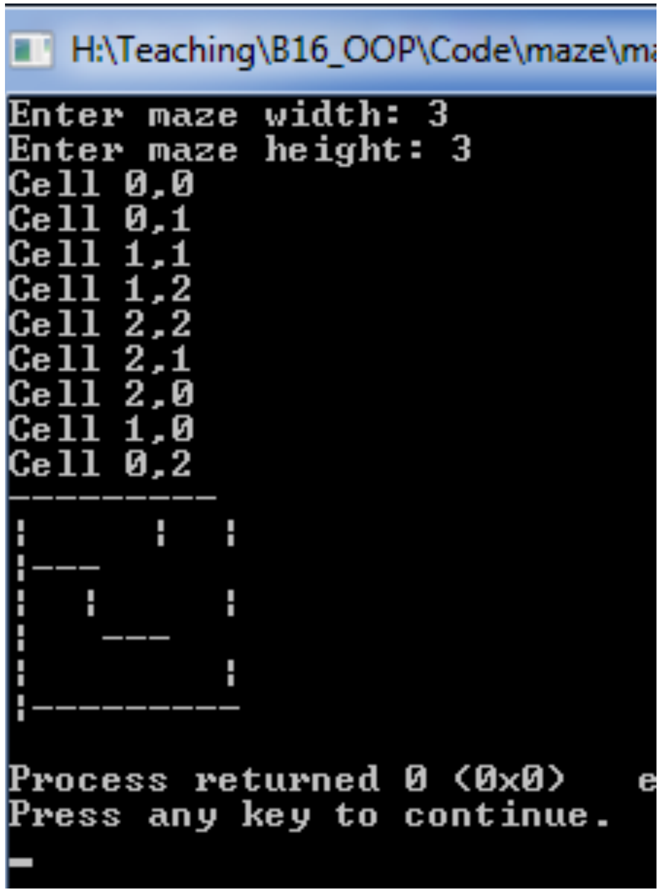
Topic 11: Maze Example

Let’s design a program to compute a maze.

We’ll input a size and have the maze printed out at the end.

The algorithm will run as follows.

1) Mark all cells as unvisited.
2) Mark all walls as standing.
3) Choose the upper left cell to be the current cell.
4) While the current cell has unvisited neighbours, choose a neighbour at random.
5) Break wall between chosen neighbour and current cell.
6) Make the chosen neighbour the current cell.
7) Return to 4).



```
Enter maze width: 3
Enter maze height: 3
Cell 0,0
Cell 0,1
Cell 1,1
Cell 1,2
Cell 2,2
Cell 2,1
Cell 2,0
Cell 1,0
Cell 0,2
---------
¦     ¦  ¦
¦---
¦  ¦     ¦
¦   ---
¦        ¦
¦---------

Process returned 0 (0x0)   e
Press any key to continue.
_
```

Now let's design the classes and class interfaces.

We need the following classes:

1. Maze class, containing
   1. a Compute method (to recursively generate the full maze),
   2. a Print method (to print to screen),
   3. a two dimensional array of Cells, defining the maze.

2. Cell class, containing
   1. Accessor methods,
   2. Break wall methods, to remove a wall of a cell,
   3. Wall flags (which walls does this cell have?),
   4. Visited flag (have we previously considered this cell?).

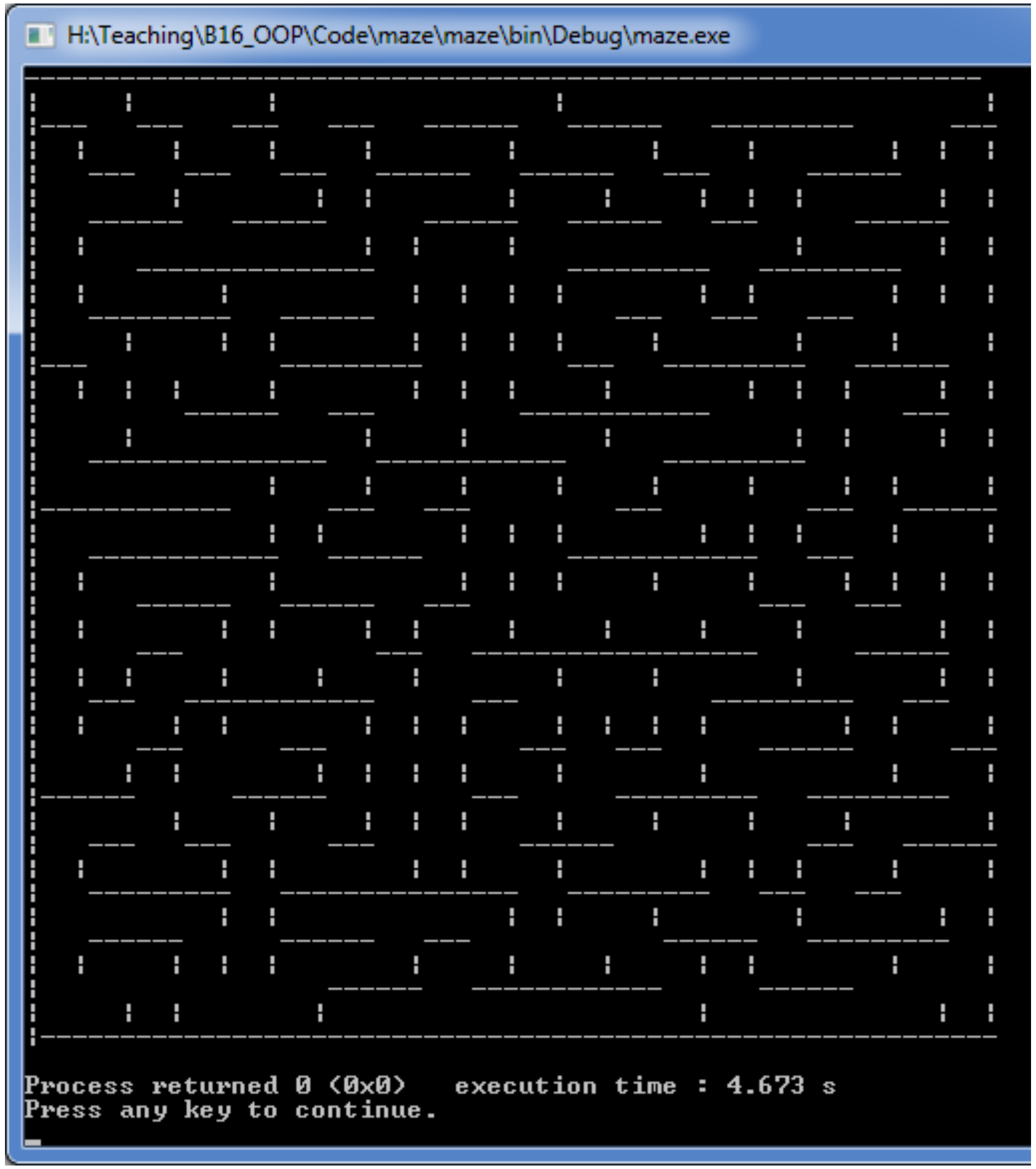
Let’s define the `Cell` class interface (`cell.h`).

```
class Cell {
    public:
     Cell();

     bool Visited();
     void MarkVisited();
     bool BottomWall();
     bool RightWall();
     void BreakBottom();
     void BreakRight();

   private:
     bool bottomwall;
     bool rightwall;
     bool visited;
};
```

Let’s define the `Maze` class interface (`maze.h`).

```
class Maze {
   public:

    Maze(int width, int height);
      void Compute(int x, int y);
      void Print();

   private:

      int Rand(int n);
      int H, W;
      std::vector< std::vector<Cell> > cells;
};
```

Here’s an example of a twenty by twenty maze.



```
Process returned 0 (0x0)   execution time : 4.673 s
Press any key to continue.
```

Topic 12: Conclusion

OOP is not the best approach to all problems.

Joe Armstrong said “the problem with object-oriented languages is they’ve got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.”

Steve Yegge said “OOP puts the nouns first and foremost. ... Why should one kind of concept take precedence over another? It's not as if OOP has suddenly made verbs less important in the way we actually think. ... As my friend Jacob Gabrielson once put it, advocating Object-Oriented Programming is like advocating Pants-Oriented Clothing.”

A study by Potok, Vouk and Rindos (1999) showed no difference in productivity between OOP and procedural approaches.

OOP is the best approach to only some problems.

Jonathon Rees wrote “(OOP) accounts poorly for symmetric interaction, such as chemical reactions and gravity.”

Paul Graham wrote “at big companies, software tends to be written by large (and frequently changing) teams of mediocre programmers. OOP imposes a discipline on these programmers that prevents any one of them from doing too much damage. The price is that the resulting code is bloated with protocols and full of duplication. This is not too high a price for big companies, because their software is probably going to be bloated and full of duplication anyway.”

He also wrote “Object-oriented abstractions map neatly onto the domains of certain specific kinds of programs, like simulations and CAD systems.”

Let’s recap.

Objects, instances of classes, unite data, and functions for acting upon them (methods). Classes are initialised by a constructor member function.

Encapsulation is expressed by classes splitting public data and functions from private data and functions, minimising side-effects.

The interface (expressed in a header file) clearly defines how a program can interact with an object of a given class. The `const` keyword can be used to define variables that cannot be changed, creating a mathematical-function-like interface.

Functions can be overloaded: the function name together with the types of its arguments define which version is being referred to. Likewise, operators (functions with weird names and calling syntax) can be overloaded.

Let’s recap.

Inheritance expresses an “is a” relationship, granting the ability to create class hierarchies. Composition expresses an “has a” relationship.

Run-time polymorphism is the ability of objects in a class hierarchy to respond in tailored ways to the same events, and is achieved through the use of virtual functions in an abstract base class.

Templates provide a way to create a generic, parameterised, class definition. This is an example of compile-time polymorphism.

The End