B16 Software Design in C++

Nick Hawes and Ioannis Havoutis*

HT2022

Contents

*These notes started from copying Mike Osbourne's and Frank Wood's B16 lecture slides into LaTeX.

1 Introduction

This first half of B16 will introduce you to software design, with a focus on the **C++** language and **object-oriented programming** (OOP).

We'll start by introducing C++ and the main features that make it different from Python (which you used in A2). We'll them move on to designing software in C++ using the OOP paradigm. As part of the software design aspects 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**, and use this to introduce **The Standard Template Library**. 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 the concept and use of flow, scope, variables, stack and heap.
- Understand how specific object-oriented constructs are implemented using C++.
- Understand concepts of and advantages of object-oriented design including:
  - Data hiding (encapsulation)
  - Inheritance and polymorphism
  - Templates.
- Understand the concepts of modularity and abstraction.
- Be able to understand C++ programs.
- Be able to write small C++ programs.

These notes only cover a very small portion of the C++ language and its correct application. We therefore recommend you supplement them with a textbook. The most authoratative reference (and our recommendation) is The C++ Programming Language (4th Edition) by Bjarne Stroustrup, who designed and implemented the C++ language. You can access this book online using your SSO here: `https://solo.bodleian.ox.ac.uk/permalink/f/89vilt/oxfaleph021512475`.

2 What is C++

C++ is a programming language which first appeared as an extension of the as an extension of the C programming language to support the paradigm of OOP. C++ and C share many properties, and both are widely used in engineering applications. In particular the C family of languages is used in applications where runtime performance is crucial (e.g. video games, database implementations) or where safety is critical (e.g. flight software, Mars rovers). You will learn about the properties of C++ in detail in the rest of these notes. For now, we will informally call out some of the major elements that are noticably different to Python, and also make C++ a good choice of language for engineering software systems. The tutorial sheet asks you to discuss how these design features link to the application requirements above, so you may want to consider this as you read on.

C++ is compiled In A2D you learned about the process of *compilation* which converted code in a high level language first to assembly language (such as MIPS) and then into machine code to be executed on the CPU. C++ is an example of a compiled language. Therefore after writing your code in a source file you must *compile* it to produce an executable. If you wish to use libraries of code from other developers in your executable, you must *link* this code into your executable so that the CPU can run it too. Therefore to create a C++ executable you need a *compiler* and (optionally) a *linker*. The resulting executable is then loaded into memory and run by your operating system. This brings in an important distinction between the process of compiling your source code ("compile time") and running your executable ("run time"). One reason that this distinction is important because different types

of errors can occur in these different phases. This is different from Python, which is an *interpreted* language. In an interpreted language, source code is incrementally converted into machine code as it is executed. Therefore to execute Python source code you need an *interpreter* which is executed first and knows how to interpret your source code as sequences of instructions which it can load into the CPU. As an exercise, it may be interesting to consider what steps you would need to follow to build an interpreter for a programming language.

C++ is statically typed You should already be familiar with the idea that different types of information must be encoded differently within a computer. For example, integers and floating point numbers have different binary encodings, which are both different again from booleans and sequences of characters ("strings"). When you create a variable in C++ you must declare in advance what kind of data it will contain. This is known as *declaring* its *type*. For example,

```
int i;
```

declares that the variable `i` can only be used to store integers. In C++, type information must be fixed at compile time. It is therefore known as a *statically-typed* language. In contrast, Python is *dynamically typed*. In a dynamically-typed language, variables can store whatever type is assigned to them, and this can change on subsequent assignments.

C++ allows for explicit memory management In A2D you learned about how memory accesses can be optimised in order to speed up executable runtime. In A2E we also (briefly) introduced memory usage as one way of measuring the performance of algorithms. However, when we write code in Python, the actual mechanics of managing memory is hidden from us. In C++, memory management is explicit. In particular programmers can explicitly create and delete chunks of memory on the heap, and choose how to that memory is passed between parts of the programme.

Syntax It is also worth stating that reading and writing C++ requires you learn a new language syntax. Hopefully most of the syntax will be familiar from other languages you've used (Python, Matlab), and the rest you should be able to learn through practice. Some of the main things you need to need to be aware of before we get going:

- Blocks of code are indicated using curly brackets/braces: `{ }`.
- Each statement within a block must be terminated by a semicolon: `;`.
- Static typing means that definitions functions must always include type information for the return type (which can include `void` indicating nothing is returned) and parameters.

2.1 A First C++ Programme

To put some of the above into context, here is an example of a programme written in Python.

```
# Define a function with one argument
def say(utterance):
    print(utterance)

# This is the entry point for execution
if __name__ == '__main__':
    # Create a string and store it in the variable
    to_say = "Hello world!"
    # Call the say function
    say(to_say)
```

And here's the same code in C++. The code has been written to deliberately demonstrate the language features described above (but in practice could be greatly simplified).

```cpp
// These statements include libraries for input/output and the string class
#include <iostream>
#include <string>
// This allows us to write string and cout instead if std::string and std::cout
using namespace std;

// Define a function that returns nothing (void).
// Its argument is a pointer to a string in memory
void say(const std::string * utterance) {
  cout << *utterance << endl;
}

// This is the entry point for execution
int main(int argc, char** argv) {

  // Declare a new variale of type string pointer,
  // i.e. something which points to a string in memory
  // You can think if this as storing the memory address of a string
  std::string * to_say;
  // Create a new string object in memory and record its address in the variable
  to_say = new std::string("Hello world!");
  // Call the say function
  say(to_say);
  // Free up the memory we created previously but no longer use
  delete to_say;

  // Return zero to indicate nominal completion
  return 0;
}
```

At this point I strongly advise you to compile and run the above C++ programme for yourself. You could do this using an online compiler (e.g. `https://replit.com/languages/cpp`). This will allow you to test the basics, but is not representative of a realworld coding environment.

To move beyond the online compile, the most basic approach is to copy and paste your code into a text file, e.g. `hello_world.cpp`, then, using the terminal, run a compiler on this file. For example, on OS X or Linux, you could use the command `clang++` or `g++` to compile this file (try the latter if the below does not work) as follows:

```
clang++ hello_world.cpp -o hello_world
```

You can then run the resulting executable (which we requested to be called `hello_world`) as follows:

```
./hello_world
```

On Windows you will need to use a slighly different approach from the commmand line. You can see an example at here.

Most of the work in B16 can be completed with a text editor and a compiler as above. Your general workflow is to write your code in the text editor, then compile it (which checks for some errors), and then run it (which may show others). This approach can become awkward as your projects start to span multiple files. Therefore you should consider using an Integrated Development Environment (IDE) instead. An IDE typically allows you to develop, compile, then run in a single window, as well as highlighting errors in your code as you write it. For B16 we recommend the Visual Studio Code IDE, since this is widely used and runs on all common operating systems. You will learn more this IDE in the B16 lab. To get started with its basic features, you should follow this guide.

2.2 Printing to Screen in C++

In C++ we *print to screen* using statements like:

```
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.

3 Variables and Memory in C++

3.1 Types

Before we go on to OOP, we first need to cover some of the fundamental syntax in C++ for defining variables. This is the part that people new to C++ typically struggle with, but we need to get this right since may of the later syntax for OOP leverage these basics.

As we saw above, a variable in C++ must be declared with a type. We can choose to *instantiate* the variable, i.e. give it a value, at the same time as declaring it, or we can do this later via an *assignment* statement. Here's some example C++ that demonstrates some basic operations with variables[1].

```
// declare i
int i;
// declare and instantiate j
int j = 10;
// assigns values to i, then j, then i again
i = 20;
j = i;
i = j + 1;
// declare and instantiate k
string k = "30";
k = "31";
i = k;
```

Take a moment to think about the above code. Will the compiler happily accept it as correct C++? And if it does, what will the final values of the variables be? You can check your answers by compiling and running the code yourself.

One of the advantages of a compiled, typed language is that the compiler can check our programme for bugs. For example, consider the function `substr` which returns a substring from a string:

```
string s = "I love engineering";
cout << s.substr(2,4) << endl;
```

It only makes sense to call this function when the argument is a string, and we must provide integer values as the arguments (for position to start extracting the substring, and its length). This means that although we could write the following...

```
string s = "I love debugging";
cout << 2.substr(2, 4) << endl;
cout << s.substr("2", "4") << endl;
```

...the code has no well-defined meaning. Luckily at this point the compiler can step in and tell us that we have some problems with our code, and where we should look to fix them. Note that in a compiled language, this checking happens at compile time, so we can find problems *before* we try to execute our code. If this was Python, we wouldn't find those errors until the interpreter tries to execute these lines (and in branching programmes, perhaps they wouldn't even be encountered on every execution).

[1] Note that unlike Python, this code will need to be pasted into a `main` method before you can run it.

3.2 Scope

You can only use a variable once you've declared it, and you must be using it in the correct *scope*. The scope of a variable is limited to the code block it is defined within, and any block nested within it. A block is defined with curly brackets, including function definitions. We typically use the term *local* to refer to a variable that is constained to the current scope. It is also possible to define *global* variables which are typically in scope for an entire source file. When the compiler passes out of a scope, any variable declarations from that scope are lost. The code below shows an example of scope and some common errors. Why not paste it into your editor and test your understanding with it.

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

// k is global to this file
int k = 30;

void blank() {
    // This next line will not compile, since i is local to main
    i = k + 20;
    // This is fine, since it's declaring a new variable called j
    int j = k + 30;
    cout << j << endl;
}

int main(int argc, char** argv) {
    // The next line won't compile since we haven't declared i yet
    i = 10;
    // This is now fine, i is local to main
    int i = 10;
    blank();
    // We can define blocks if we wish
    {
        cout << i << endl;
        i = i + 1;
    }
    {
        // k is global, so can be used here
        int j = k + 20;
    }
    cout << i << endl;
    // The next line will not compile, since j was limited in scope to the block above
    cout << j << endl;
    // This is fine. Local definitions override global ones
    int k = 2000;
    cout << k << endl;
    blank();
    return 0;
}
```

3.3 The Stack

From our adventures in MIPS in A2D we already know roughly what happens when we declare a new local variable. Initially that variable is associated with a register and then the data that is assigned to that variable is loaded into that register. As long as the variable is still in scope then it must still be accessible by the programme. However, the register we initially used for the variable may be required by a subsequent instruction. Hopefully you recall that this is when the value of the register is written to the *stack* for later retrieval. This value will remain on the stack until it is used again, or the stack pointer is reset to a location lower on the stack (or higher in memory address terms).

In B16 we will abstract away from the detail of these operations, and simply consider that local variables use memory which is allocated from the stack. For example, when we do the following we speak of two local variables being allocated on the stack...

```
int i = 10;
float j = i * 1.11f;
```

...although we know that if we translated just those statements directly into MIPS we wouldn't need to use the stack directly. What is important about the stack is that the compiler automatically manages the allocation and freeing of memory by translating our C++ to the correct sequence of assembly instructions.

3.4 Calling Functions

We can also use our MIPS knowledge to understand what happens when we pass arguments into a function. In MIPS we copied the values we wanted the function we were calling to receive into the special registers `$a0` to `$a3`, before jumping the programme counter to the appropriate location. A similar thing was required for the function to return a value to the calling location (this time with `$v0` to `$v1`). The important thing to note here is that the functions pass *values* to each other by *copying*. In higher level languages such as C++ we refer to this process as *pass-by-value*. For example, in the code below, the arguments to function `mult` are passed to it by copying the values 10 and 20 into the argument registers, where they are retrieved by the instruction in that function. The result of the multiplication is then copied to a result register from where it is copied into the register corresponding to `k`.

```
int mult(int a, int b) {
    return a * b;
}

// This is the entry point for execution
int main(int argc, char** argv) {
    int i = 10;
    int j = 20;
    int k = mult(i, j);
    return 0;
}
```

3.5 Memory Allocation

Everything we described above works well when we're talking about variables that store values that occupy a single word in memory, such as integers or standard floating point numbers. But once we start working with data types that occupy more memory such as arrays (the basic C++ name for what would be called a list in Python) or user-defined data types (such as those we will create shortly), storing everything on the stack, and using pass-by-value can have negative consequences. For example, consider trying to load a 1920 x 1080 image into a C++ programme and then passing it between different functions to process it (e.g. to remove motion blur, or correct colour imbalances due to lighting). If we use a byte per pixel, this would require approximately 2MB to be copied twice on every function call (once on input, once on output), slowing down our application. If the image data was always allocated on the stack, then the stack pointer would regularly be moving by large amounts. This would great reduce the ability of cache underpinning the stack to exploit the locality which is usually present in stack accesses.

C++ allows us to avoid these problems by allocating memory on the *heap*, an area of memory which is set aside specifically for storing larger values created during programme execution. This is achieved using the `new` keyword. For example, the following code allocates enough memory on the heap to store a single integer:

```
new int;
```

This does not make much sense by itself, since we also want to put a value into that memory. Therefore C++ provides *pointers* which are used to point to a location in memory. A pointer is effectively a way of tracking the address of a value in memory. Since C++ requires the type of a variable to be known at compile-time, a pointer must be associated with the type of the value it is pointing to. A pointer is declared by writing the type being pointed to, followed by the symbol `*`. For example:

```
// Declare an int pointer called p_i
int * p_i;
// Assign the pointer the memory address of some newly allocated memory
p_i = new int;
// Perform allocation and construction in one line
int * p_j = new int(20);
```

In order to interact with the memory pointed to (or *referenced by*) by a pointer we must *dereference* it. We dereference a pointer using the unary `*` operator. Note this is the same symbol we use when we declare a pointer type, but the context of usage should unambigiously tell you which meaning is intended. When we reference a pointer we gain access to the value sitting in the heap. We can use this to both read from and write to that memory. For example:

```
// Store the value 20 on the heap
int * p_j = new int(20);
// Print out the memory address stored in the pointer
cout << p_j << endl;
// Deference pointer, printing out the memory contents
cout << *p_j << endl;
// Deference pointer, overwriting the memory with 10
*p_j = 10
// What is printed out after this?
*p_j = *p_j * *p_j;
cout << *p_j << endl;
```

If our pointer is to an *object* which provides *member functions* we need to dereference the pointer before calling a function. For example, if we replace the `string` in our substring example above with a pointer to a `string`, we would need to do the following:

```
string * p_s = new string("I love engineering");
string substr_1 = (*p_s).substr(2,4);
```

You can read this as `(*p_s)` being evaluated to produce a normal `string` variable, which then has its `substr` method called. Since calling functions via pointers is required often, C++ provides a shorthand for this: the operator `->`. This operator is applied to a *pointer* and combines the dereference (`*`) and function accessor (`.`) into a single step. You can see this below:

```
string substr_2 =  p_s->substr(2,4);
```

If we have a non-pointer variable, we can take the address of the value using the `&` unary operator. This is exemplified below where we create a pointer that points to the value that is stored in `i` initially. But please note, this simple example contains a dangerous problem. If you think back to what we know about scope, perhaps you can spot it?

```
int i = 10;
int * p_i = &i;
```

3.6 Freeing Memory

Once you create memory, you are in charge of managing. The `new` operator remembers what memory it has assigned and will not reassign it unless you declare that you are no longer using it. To understand why this is important, consider the following (nonsensical) example:

```
int process_pixel(int a, int b) {
    // allocate a new array in memory based on input values
    int * p_array = new int[a*b];
    int c = a * b;
    p_array[0] = c;
    return c;
}

int main(int argc, char** argv) {
    int width = 1980;
    int height = 1080;

    for (int i = 0; i < width; i++)
    {
        for (int j = 0; j < height; j++)
        {
            process_pixel(i, j);
        }
    }

}
```

Each time we call `process_pixel` we allocate an array on the heap. The size of those arrays can each be millions of times the size of a single integer. Depending on the values set for `width` and `height` we may exhaust the heap space available, causing a out-of-memory exception to crash our executable.

To inform the operating system (to which `new` is a simplified interface) that we no longer require a particular chunk of memory, we can use the `delete` operator. This call allows the operating system to free the memory addressed by the pointer so it can be used on a subsequent call to `new`. To include this, we would modify our example as follows:

```
int process_pixel(int a, int b) {
    // allocate a new array in memory based on input values
    int * p_array = new int[a*b];
    int c = a * b;
    p_array[0] = c;
    // free the allocated memory since we no longer need it
    delete p_array;
    return c;
}
```

3.7 Common Pointer Problems

The ability to allocate memory dynamically, and to refer to it using pointers, provides great power to the programmer. It is also the cause of many bugs in programmes. Two common problems are *memory leaks* and *segmentation faults*. A memory leak occurs when memory is allocated, but not freed once it is no longer needed and its pointer goes out of scope. If this happens a chunk of memory is no longer accessible to your programme. In the worst case a memory leak may cause your programme to exhaust the heap and no longer be able to allocate memory (causing it to crash). A segmentation fault occurs when code tries to read or write memory that is invalid, e.g. it has been freed prior to its access. This can happen when multiple pointers exist to an object in memory and one deletes it incorrectly, or if the the pointer is to a local varible (on the stack) which has since gone out of scope.

To reduce the likelihood of such errors it is important to make sure you know which pointer *owns* the memory, and is therefore in charge of deleting it when it is no longer needed. Modern

`int * p = & n;`	Define **pointer** `p` as the address where integer `n` is stored.
`& n`	is the **address** (in memory) of variable `n`.
`int i = * p;`	Dereference pointer p to copy the contents from the memory it points to to i.
`* p`	Dereference pointer `p`.
`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).

Table 1: Recap of syntax for pointers and references.

C++ implementations provide special pointer types which are designed to prevent prpogrammers from making these errors. For example, they automatically manage and delete memory when it is no longer used. You will meet these special pointers in the second half of these notes.

3.8 References

Pointers were inherited by C++ from C. But as C++ added capabilities (e.g. operator overloading) the syntax of using pointers became quite ugly. Therefore C++ also uses a type of variable called a *reference*, indicated with `&`. In short a reference can be used as an *alias* for another variable. You can work with references much as you work with standard variables, but this is done without any additional copying. The key idea is that any operation performed with a reference is also done on the original value. For example, the following code will print `1` which indicates the final comparison returns `true`. This is because the assignment to `k` is really operating on `i`, since `k` is an alias (i.e. another name for) `i`.

```
int i = 10;
int j = 20;
// k is now an alias for i
int & k = i;
k = 20;
cout << (i == j) << endl;
```

As a rule of thumb you should use references instead of pointers where you can, since it is generally harder to make errors using references. A recap of the syntax for pointers and references is provided in Table 1.

3.9 Pass by Reference

In Section 3.4 we saw that calling a function can involve a lot of overhead due to copying values on to the stack. Pointers and references allow us to avoid much of this overhead by passing the memory address of the value, rather than the value itself. In this case the copying is limited to the size of a memory address, regardless of the size of the value. This approach is called *pass-by-reference*, and is achieved by using pointer or reference types as arguments for in functions. For example, returning to our toy image processing code, we can now pass the "image" using a pointer:

```
int width = 1980;
int height = 1080;

int get_pixel(int * image, int column, int row) {
    return image[(row * width) + column];
}

int main(int argc, char** argv) {
    int * image = new int[width*height];

    for (int i = 0; i < width; i++)
    {
```

```
        for (int j = 0; j < height; j++)
        {
            int pixel_value = get_pixel(image, i, j);
        }
    }
}
```

For another example, consider the following code which tests if the selected substring of a string matches another string:

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

using namespace std;

bool test_substring(string & s, int pos, int len, string & test) {
    string substr = s.substr(pos, len);
    return substr == test;
}

int main(int argc, char** argv) {
    string s = "I love engineering";
    string target = "love";

    cout << "input: \"" << s << "\" test: \"" << target << "\"" << endl;
    cout << test_substring(s, 2 , 4, target) <<endl;
    cout << test_substring(s, 7, 12, target) <<endl;
    cout << "input: \"" << s << "\" test: \"" << target << "\"" << endl;
}
```

This should print out the following:

```
input: "I love engineering" test: "love"
1
0
input: "I love engineering" test: "love"
```

3.10 Const

Although pointers and references allow us to speed up the execution of our code, they also introduce a new problem. When we pass a value by reference, the receiving code can *alter* the value and the alteration affects the original copy. Sometimes this is desirable (as in our `i == k` example above), but more often than not we do not want our functions to have *side-effects*, i.e. interact with our wider programme other than through arguments and return values. Consider the following alteration to our `test_substring` function:

```
bool test_substring(string & s, int pos, int len, string & test) {
    string substr = s.substr(pos, len);
    bool result = substr == test;
    // alter the input string
    s[14] = 'k';
    return result;
}
```

This would change the output to be as follows:

```
input: "I love engineering" test: "love"
1
0
```

```
input: "I love engineeking" test: "love"
```

. . . which would probably be considered an underdesirable side effect.

To indicate to the compiler that we do not want the value of our strings to change, we can mark them as `const`:

```
const string s = "I love engineering";
const string target = "love";
```

This means that they should be treated as *constants*. This change will stop our programme from compiling because the `test_substring` function treats its inputs as references, which it is allowed to change. We must therefore update its signature to accept constant references:

```
bool test_substring(const string & s, int pos, int len, const string & test) {
    ...
}
```

You can consider the function signature as defining a contract that the function must follow. The `const` modifiers on the string references state that the contract prevents the function from modifying those values. This will cause another compilation error, this time at the line where we make the modification to the input string. This demonstrates the power of typing and the compiler: typing allows us to define the contract we wish our code to follow; and the compiler will only accept code that actually follows that contract.

3.11 Primitive and User-Defined Types

The C++ specification defines *primitive types* that are directly supported by the compiler. These include integers (`int`, `long`), booleans (`bool`) and floating point numbers (`float`, `double`). All other types and data structures must be built from these primitive types. The rest of this course describes how you can create user-defined types (called *classes*) using the object-oriented programming (OOP) paradigm. We have already seen examples of classes above, e.g. `string`.

4 Programming Paradigms

Before tackling the details of object-oriented programming, we first look at how this paradigm fits into a wider context of top-down programme design. Top-down design means breaking the problem (or programme) down into components (modules) recursively. Each module should contain related data and the functions to operate on that data: OOP is a way of making this relationship 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. The aim of modularity is to define a set of modules where 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.

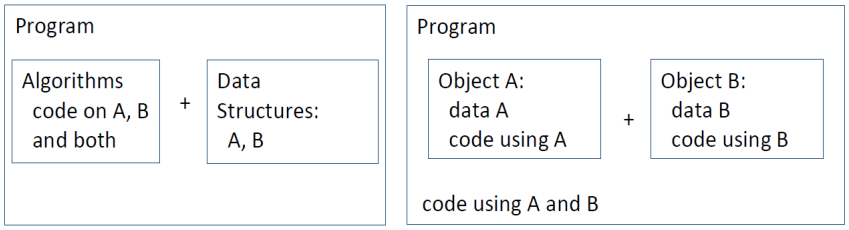

Figure 1: Stuctured (left) compared to Object-Oriented (right) programming paradigms.

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.

OOP is a programming paradigm which supports a particular stragegy for following these steps. The list below provides a brief list of other 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, Python
Object-Oriented	Data, and procedures for acting upon them (methods), are united in objects.	C++, Java, Python

5 Object-Oriented Programming

In structured programming, such as you performed in Python in A2, structures contain only **data**, and we separately create **functions** to act on them. The key insight in OOP is to create **objects** which contain both **data** and the **functions (methods)** to operate upon this data.

An **object** is an *instance* of a **class** that has been instantiated for use. You can imagine a **class** as defining a template, recipe, or blueprint for creating something, and the **object** as an **instance** of the thing created (programmers tend to say *instantiated* or *constructed*). Consider mobile phones for example. The iPhone 13 is the class, and an individual physical phone is the object instantiated from that class. Note that different configurations can be created when instantiating (or constructing) an iPhone 14 object (e.g. different storage space or data plan) and this object can also be changed later (e.g. apps are added). The same is true for objects in a programming language.

An **object interface** defines how an object can be interacted with, providing an explicit separation of how an object is used from its *implementation details*. Returning to the phone example, you know where to touch the screen to activate various operations, how to configure the network etc. This is the *interface*. The *implementation* is how these operations change the internal state of the phone's software and hardware. You (thankfully) don't need knowledge of the latter to do the former.

6 Defining Classes in C++

Classes are a mechanism for building compound data structures i.e. *user-defined* types. A class provides a mechanism for encapsulating one or more related data items, called **fields** or **data members**, into a single structure. This structure can also contain **member functions** which operate on the data (alternatively called **methods**). C++ also allows a means to control access to the the **fields** and **methods** of the class (via the `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).

6.1 Example: Complex numbers

This is perhaps best illustrated with an example. The following defines a class to descrie a complex number.

```
// Class definition
class Complex {
  // Access control specifier
  public:
    // Fields are declared inside the class definition
    double re;
    double im;
};
```

Once we have defined a class, we can use it to declare class of a variable.

```
// New variables of type int and double
int i;
double d;
// Create an instance of the Complex type
Complex z;
```

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

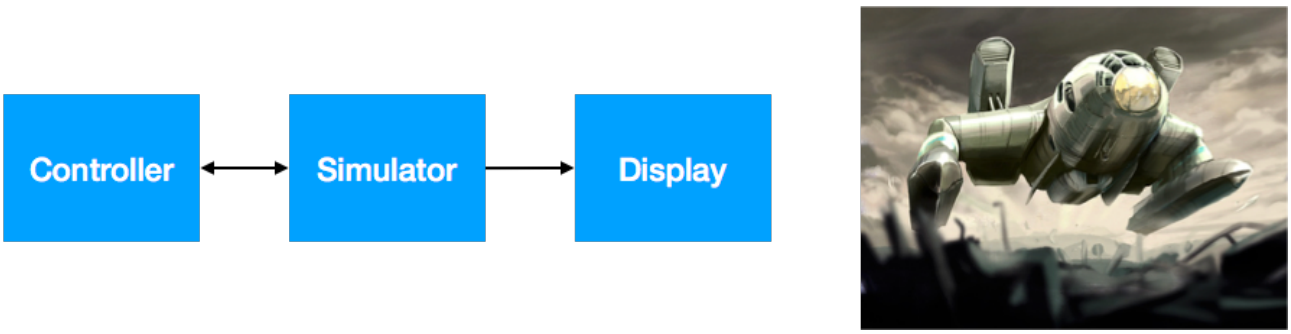

Figure 2: High-level sketch of a programme for controlling a simulator vertical take-off and landing (VTOL) aircraft.

As another example, consider trying to implement the software sketched in Figure 2. We can represent the current state of the aircraft with three numbers plus a bool: `position`, `velocity`, `mass`, `landed`. We could implement the state using *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.

Instead, we could design a `State` class which combines all of the state variables into a single structure in our code:

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

Class members are accessed using the **member selection operator** (a full stop).

```
//s is an instance of type State
State s;

//This assigns values to the data members of s
s.pos = 1.0;
s.vel = -20.0;
s.mass = 1000.0;
s.landed = false;

// This demonstrates reading and writing data members
s.pos = s.pos + s.vel * deltat;
```

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's methods.

6.3 Member Functions

Recall that in C++ a class encapsulates related data and *functions that operate on the data*. Functions within classes are called **member functions** or **methods**. Methods (like data fields) are called using the full stop operator.

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

    double Magnitude() {
      return sqrt(re*re + im*im);
    }

    double Phase() {
      return atan2(im, re);
    }

};

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

Notice that the fields `re` and `im` are used *within* the member functions without the reference to `z`. When a method is invoked, it is invoked on a *specific object* (in this case `z`), so we already know which object's fields are being referred to. Within a member function the calling object instance (the `z` in this case) is available via **a special pointer**: the `this` pointer. Therefore in the rare event that you need access to the whole object from within a member function, you can get access via the `this` pointer. The following demonstrates the explicit use of the `this` pointer:

```
// The Phase method with the implicit this pointer explicitly included
double Phase() {
  return atan2(this->im, this->re);
}
```

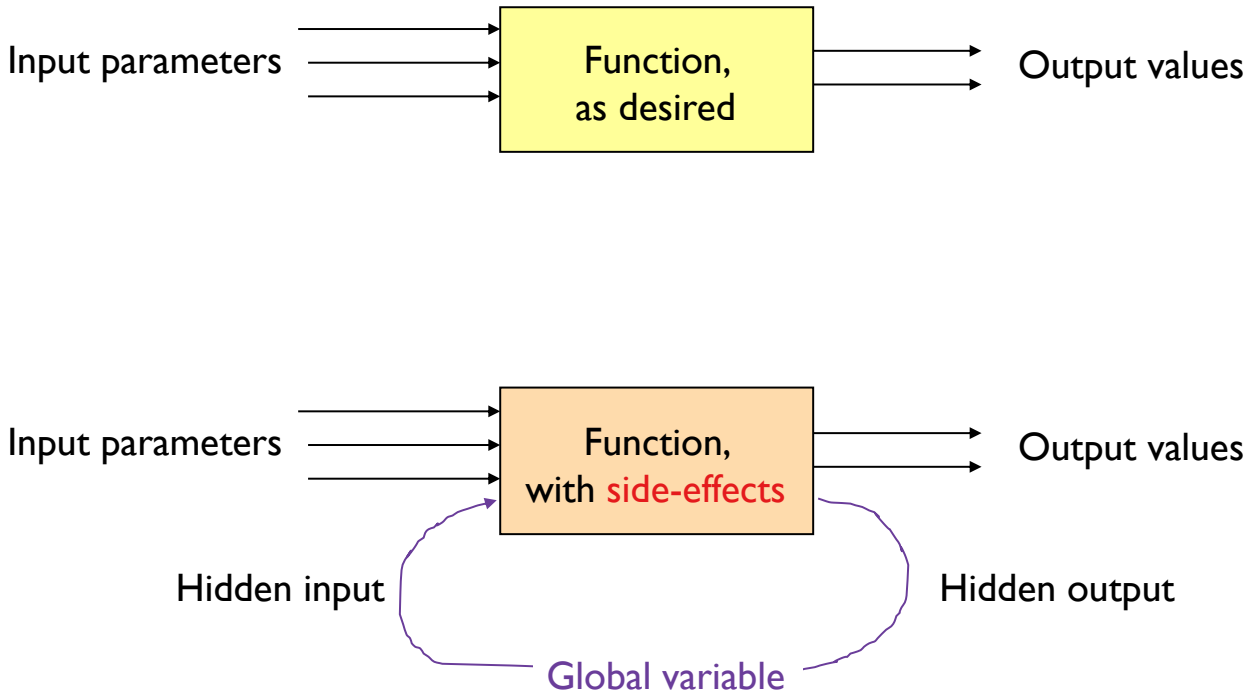

Figure 3: Code with side-effects can can unpredictable behaviour.

7 Encapsulation

Information hiding / data hiding / encapsulation is the ability to make object data available only on a "need to know" basis. The need for encapsulation has made OOP the programming paradigm of choice for large projects. By specifying 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** (Figure 3, 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*. We cam re-rewrite our previous `Complex` class example to use the `private` keyword to protect our class-internal data:

```
class Complex {
  public:
    double Real() { return re; }
    double Imaginary() { return im; }
    double Magnitude() { return sqrt(re*re + im*im);}
    double Phase() { return atan2(im, re);}
  private:
    double re, im;
};
```

The private data fields (`re` and `im`) of an instance of the `Complex` class cannot be accessed by other software components. Private data members can be accessed only through the public interface: the

(*read-only*) **accessor methods** (or **getter** methods) `Real()` and `Imaginary()`. Code that used `z.re` or `z.im` would produce a compilation error.

```
Complex z;
cout << "Real part = " << z.Real() << endl;
cout << "Magnitude = " << z.Magnitude() << 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*. A major advantage of this approach is that we can change the *internal representation* of a concept in our code, but the interface can remain the same. Here we do this for our `Complex` class:

```
class Complex {
  public:
      double Real() { return r*cos(theta); }
      double Imaginary() { return r*sin(theta); }
      double Magnitude() { return r;}
      double Phase() { return theta; }
    }
  private:
    // Internal implentation now in polar coords
    double r, theta;

};
```

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

```
Complex z;
// The code which uses the object instance is unchanged!
cout << "Real part=" << z.Real() << endl;
cout << "Magnitude=" << z.Magnitude() << endl;
```

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

8 Constructors

In our definition of the `Complex` class above, the member variables `re` and `im` are private. This means assigning to them (e.g. `z.re = 1.0;`) would produce a compilation error. So, *how can we get values into an object?* Consider what happens when a variable is 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 though, 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. The constructor must be called once each time an instance (or object) of the class is created. Here is a constructor for our `Complex` class:

```
// A constructor for our Complex class
Complex(double r, double i) {
    re = r;
```

```
    im = i;
}
```

Rather than assigned values to member variables in the body of the contructor, we can use an **initialisation list** before the body. An initialisation list is an *implicit* way of defining a constructor using other constructors:

```
// re(x) and im(x) use the constructors of the double class
// note no semi colon at the end of the line
Complex(double x, double y) : re(x), im(y) {
    //... nothing else to do here
}
```

Although using an initialisation list is a little less readable, this approach is generally preferred as it allows the initialisation of const variables, the initialisation of reference member variables, composition and, inheritance (more on all these later). Some other things to consider when writing constructors:

- members' constructors are called before constructor body
- members' constructors called in the order in which the members are declared in the class rather than in the order of the initialiser list
- destructors are called in opposite order after own destructor
- implicitly initialised members of built-in types are left uninitialized (!!!)
- "the rules are not as clean as we would like." ..."However, for local variables and free-store objects, the default initialisation is done only for members of class type, and members of built=in type are left uninitialised" [Stroustrup, 2014, pg. 490]

Let's add the constructor to the definition of the for `Complex` class.

```
class Complex {
    public:
            Complex(double r, double i) : re(r), im(i) { }
            double Real() { return re; }
            double Imaginary() { return im; }
            double Magnitude() { 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.Magnitude() << endl;
cout << "Real part=" << z.Real() << 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 parameter (`r`) and the value of `im` to be the second parameter (`i`). Hence we have declared (instantiated) an object of type `Complex`, with value 10+j8. Changing to polar coordinates requires changing the workings of methods, *including the constructor*. However, the interface is *unchanged*.

```
// Complex class with polar coordinate implementation
class Complex {
    public:
            Complex(double r, double i) {
                //
                r = sqrt(r*r + i*i);
                theta = atan2(i,r);
            }
            double Real() { return r*cos(theta); }
```

```
            double Imaginary() { return r*sin(theta); }
            double Magnitude() { return r;}
            double Phase() { return theta; }
        }
    private:
            double r, theta;

};

// The interface is unchanged!
// Hence all other code using Complex need not be altered.
Complex z(10.0,8.0);
cout << "Magnitude=" << z.Magnitude() << endl;
cout << "Real part=" << z.Real() << endl;
```

We can define our constructor to supply default values for data. Hence our constructor can act as a *default constructor* in the *absence of some or all input parameters*.

```
class Complex {
 public:
  Complex(double r = 0.0, double i = 0.0) {
    re = r;
    im = i;
  }
 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(const Complex & z) : re(z.Real()), im(z.Imaginary()) {}
```

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* (if that doesn't make sense to you, now is a good time to stop and revise pointers, objects and constructors again!).

9 Const in Classes

As we saw in Section 3.10 it is good practice to declare constants explicitly. It is also good practice to declare parameters `const` if the function does not change them. 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*. For the value of the object not change requires that the value of its data fields should not be changed.

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

However, this can often lead to difficult-to-find *compile-time errors*, e.g. the code below will not compile.

```
const Complex z(10.0, 8.0);
// Compile time error below
cout << "Magnitude=" << z.Magnitude() << endl;
```

The reason for this is that the compiler doesn't analyse the semantics of each function, so it can't be sure that the accessor function `z.Magnitude()` doesn't change the internal values (`re` and `im`) of the object `z`. The way to avoid this is to *declare the accessor methods* as `const` functions. To do this we add `const` after the function name for functions that should not change the value of the data fields. 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 r, double i) : re(r), im(i) { }
            double Real() const { return re; }
            double Imaginary() const { return im; }
            double Magnitude() 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);
// This will now compile
cout << "Magnitude=" << z.Magnitude() << endl;
cout << "Real part=" << z.Real() << endl;
```

Faced with an error like above, you may be tempted to *make all data public* (and not use `const`). **Don't!** The grief caused by doing things 'properly' will be minor compared to the potential problems created by code that *abuses the interface*. In general, mark your variables and functions as `const` as early as possible in your development. This will force you to write correct code from the start. This is typically easier than having to re-engineer your code to include `const` at a later stage.

10 Implementation and Interface

So far our examples have written all our C++ code in one place. However, in larger C++ programs, it is necessary to separate your code into *header files (.h or .hpp)* and *implementation files (.c or .cpp)*. Every program element that uses the Complex class needs to know its **interface**. This should be specified in the header.

```
// Complex.h:

class Complex {
  public:
      Complex(double x, double y);
      double Real() const;
      double Imaginary() const;
      double Mag() const;
      double Phase() const;

  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 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 r, double i) {
  re = r; im = i;
}

double Complex::Real() const {
  return re;
}

double Complex::Imaginary() const {
  return im;
}

double Complex::Magnitude() const {
  return sqrt(re*re+im*im);
}

double Complex::Phase() const {
  return atan2(im,re);
}
```

Arithmetic	`+ - * / %`
Relational	`== != < > <= >=`
Boolean	`&& || !`
Assignment	`=`
I/O streaming	`<< >>`

Table 2: C++ operators

11 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: $e^{a+bi} = e^a e^{ib} = e^a(cosb + isinb)$.

```
#include <cmath>

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

When should we use a *member function* and when should we use a *non-member function* (as in `exp` above)? This is ultimately a matter of style and personal preference. Programming is full of choices like these where people differ (sometimes passionately) about the way to write code. Whatever you prefer, it is important that you are consistent in your approach so that your code becomes easier to understand.

As you know, C++ (and Matlab etc.) provide **operators**, which are functions, except with disallowed names and/or different calling syntax. Some of the C++ operators are shown Table 2. 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.Real() + z2.Real(), z1.Imaginary() + z2.Imaginary());
  return zout;
}
```

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

```
Complex 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 to achieve the addition code desired above:

```
Complex operator+(const Complex & z1, const Complex & z2) {
  Complex zout(z1.Real() + z2.Real(), z1.Imaginary() + z2.Imaginary());
  return zout;
}
```

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;
// assignment operator
z3 = z1 + z2;

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

Below is the definition of the assignment operator `=` for the Complex class. The definition of `operator=` is one of the few common uses for the `this` pointer. Here it is dereferenced and returned (via a reference).

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

`z2 = z1` is shorthand for `z2.operator=(z1)`. Since assignment is changing the contents of an object `operator=` must be a *member function* so it has access to the (private) internals of an object.

The left hand side of `z2 = z1` is implicitly passed in to the function as the calling object (the `this` pointer). 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* when memory is accessed incorrectly). Let's create our own safe array class by *overloading the array index operator* `[]`. Note this is an example for an educational purpose. In reality you would use a container such as a `vector` from the *Standard Template Library*.

```
class SafeFloatArray {
  public:
    ...

    // returns a reference to a float
    float & operator[](int i) {
      if ( i<0 || i>=10 ) {
        cerr << "Index out of bounds!" << endl;
        exit(1);
      }
      return a[i];
    }

  private:
    // Defines a 10-long array of floats
    float a[10];
```

```
};
```

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 "Index out of bounds!" and the program exits.

12 A Complete Complex Example

Let's bring the previous sections together by creating program to calculate the frequency response of the transfer function $H(j\omega) = 1/(1 + j\omega)$. We'll need three files, corresponding to two modules:

1. Complex.h and Complex.cpp
2. main.cpp

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

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:
  // Note that default values are supplied only in the header.
  Complex(const double r=0.0, const double i=0.0);
  Complex(const Complex & z);
  double Real() const;
  double Imaginary() const;
  double Magnitude() const;
  double Phase() const;
  Complex & operator=(const Complex & z);

 private:
  // The underscore is just used to denote a private field
  // (it's just an arbitrary part of a name).
  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);
```

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

```
// 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.

#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.Real()), _im(z.Imaginary()) {}

double Complex::Real() const { return _re; }
double Complex::Imaginary() const { return _im; }
```

```
double Complex::Magnitude() const { return sqrt(_re*_re + _im*_im); }
double Complex::Phase() const { return atan2(_im, _re); }

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

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

// Complex subtraction
Complex operator-(const Complex & z1, const Complex & z2)
{
  Complex zout(z1.Real() - z2.Real(),
               z1.Imaginary() - z2.Imaginary());
  return zout;
}



// scalar multiplication of Complex
// Overloading to be able to put the scalar in either input.

Complex operator*(const Complex & z1, const double r)
{
  Complex zout(r * z1.Real(), r * z1.Imaginary());
  return zout;
}

Complex operator*(const double r, const Complex & z1)
{
  Complex zout(r * z1.Real(), r * z1.Imaginary());
  return zout;
}

// Complex multiplication
Complex operator*(const Complex & z1, const Complex & z2)
{
  Complex zout(z1.Real() * z2.Real() - z1.Imaginary() * z2.Imaginary(),
               z1.Real() * z2.Imaginary() + z1.Imaginary() * z2.Real());
  return zout;
}

// Complex division
Complex operator/(const Complex & z1, const Complex & z2)
{
  double denom(z2.Magnitude() * z2.Magnitude());
  Complex zout((z1.Real() * z2.Real()
                    + z1.Imaginary() * z2.Imaginary()) / denom,
               (z1.Real() * z2.Imaginary()
                    - z1.Imaginary() * z2.Real()) / denom);
```

```
  return zout;
}
```

`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, 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.Magnitude() << " " << z.Phase() << endl;
  }
}
```

You can complile then link as follows:

```
g++ -c Complex.cpp
g++ -c main.cpp
g++ Complex.o main.o -o main
```

You can then execute the `main` executable.

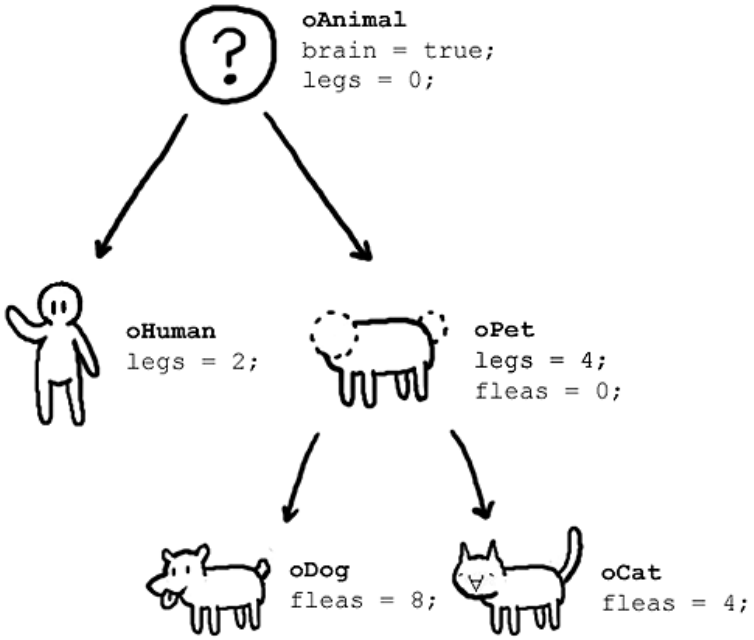

Figure 4: An (interestingly rendered) motivating example of a class hierarchy.

13 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 (see Figure 4).

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*. A 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.

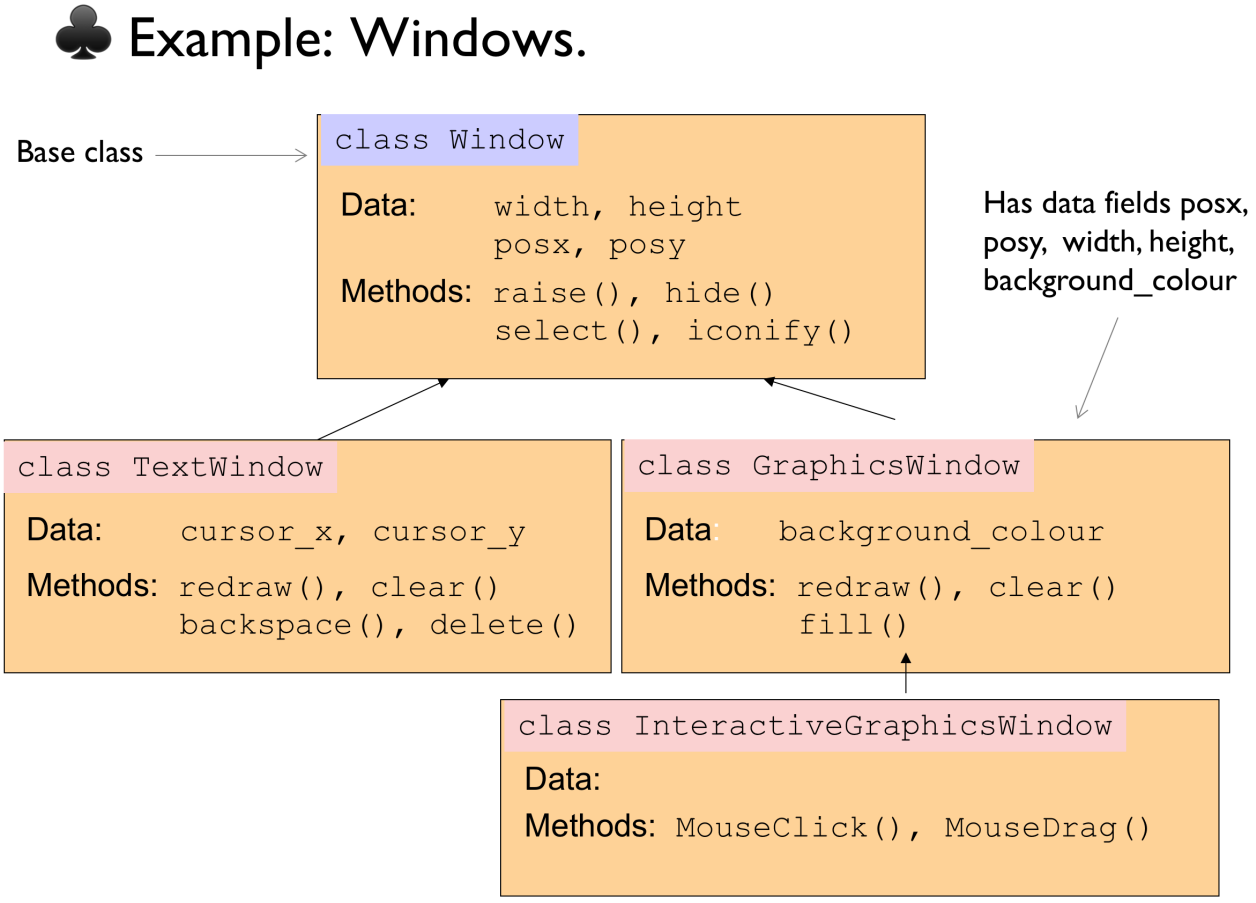

Figure 5: An example class hierarchy from a GUI windowing toolkit.

The keyword used before the typename of the ancestor (or parent, super-, base) class in the class definition controls the *access* which the inheriting has to members of the ancestor. The options (`public`, `private` or `protected`) are summarized in the following listing:

```
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};


class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected, therefore external access is removed but field is still inheritable
    // y is protected
    // z is not accessible from C
};

class D : private A
{
    // x is private, therefore external access is removed and field no longer inheritable
    // y is private, therefore field no longer inheritable
    // z is not accessible from D
};
```

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.

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*.

Figure 6: Polymorphism allows different classes (this this case different animals) to respond to the same method ("*Now Speak!*") with different (hidden) implementations.

14 Polymorphism

Polymorphism (Greek for "many forms") 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*. This is illustrated in Figure 7. 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*.

To understand the implementation of polymorphism in C++ we'll use the example in Figure 8. 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.

```
#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();
```

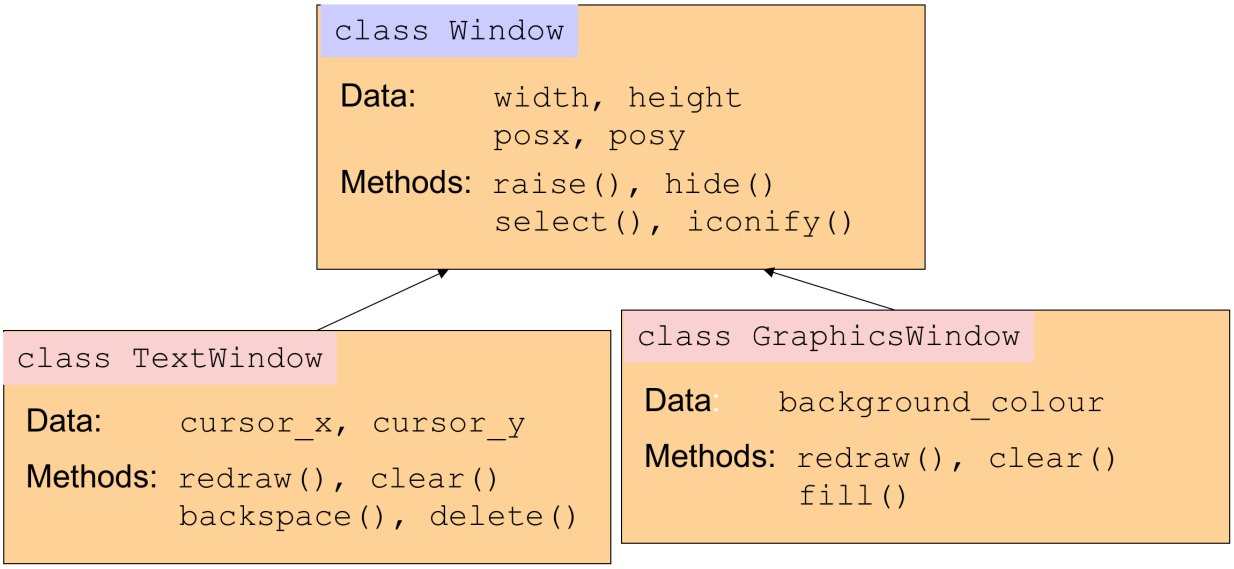

Figure 7: Polymorphism allows TextWindow and GraphicsWindow to redraw() themselves but in different ways.

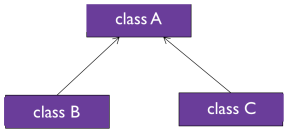

Figure 8: A *class diagram* showing a simple class hierarchy.

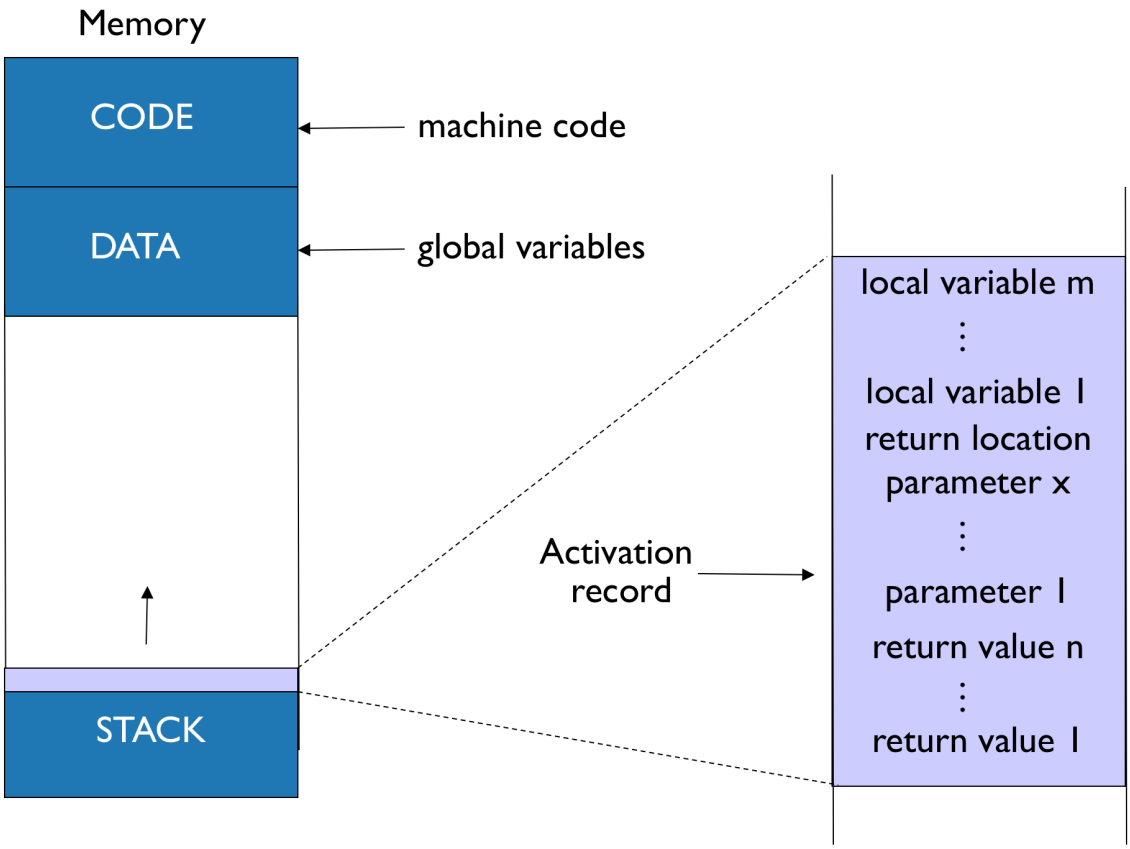

Figure 9: A visualisation of how the activation record is used when calling a function.

```
}

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()` 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()`. `callfunc(x)` passes an object of type `A` into `callfunc`. The effect is to call the `func()` method of class `A`. `callfunc(y)` passes an object of type `B` into the function. Because every `B` is also an `A` this is legal, 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()`.

Recall, as shown in Figure 9, that when a *function is called*, parameters, return location and other stuff are put onto the stack in the *activation record*. With this in mind 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**. Below we re-write the example. Now `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:
          // Now a virtual function
        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"; }
};
```

```
// param is now passed by reference
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()`. `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. In the case of `callfunc(y)`, `callfunc` still takes a reference to an object of class `A`. A reference is just a memory address. The call is legal 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`. This is as opposed to the non-virtual case where the parameter passed by value was constrained to act as an `A` regardless of the method called.

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 reference to a derived class or a pointer to a derived class performs run-time type identification on the object that invoked the call, and will call the appropriate version. In our example, if the object is of type A, then it will call A's func(), if the object is of type B, then call B's func(), etc.

In the code we have written thus far it is up to the designer to decide whether or not to override a method in a derived class. However, this can be enforced by the use of **pure virtual methods**. If class `A` defines `func()` as

```
// a pure virtual, or abstract, method
virtual void func() = 0;
```

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` thus 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`. 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()`.

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

int main(int argc, char* argv[])
{
        A x;
}
```

14.1 Example: A Vector Graphics 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 {
        ...

        // This forces all derived classes to implement Draw().
        virtual void Draw() = 0;
};
class Line : public Drawable { ... };
```

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.

14.2 Example: 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];
};
```

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.

15 Templates

```
class SafeFloatArray {
  public:
    ...

    // returns a reference to a float
    float & operator[](int i) {
      if ( i<0 || i>=10 ) {
        cerr << "Index out of bounds!" << endl;
        exit(1);
      }
      return a[i];
    }

  private:
    // Defines a 10-long array of floats
    float a[10];
};
```

Recall our array-bounds-checking class above: unfortunately, it was *limited to storing floats*. How might we create a class that can store any type we choose? **Templates** provide a way to parametrise 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 `T`). This is a *class template*, a template that is used to generate classes.

```
template <class T>
class SafeFloatArray {
  public:
    ...

    // returns a reference to a T
    T & operator[](int i) {
      if ( i<0 || i>=10 ) {
        cerr << "Index out of bounds!" << endl;
        exit(1);
      }
      return a[i];
    }

  private:
    // Defines a 10-long array of floats
```

```
    T 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 `T` is replaced with `int` everywhere in the definition. This is a *template class*, a class that is produced by a template. Of course, you could do this by simply copy-and-pasting the code and doing a manual replacement of `T`. As usual, however, **copy-and-pasting code is a very bad idea**! We can also use templates to include additional data in a class which is fixed at compile time. For example, below we create a new version of our safe array where the size (`S`) is fixed at compile time.

```
template <class T, int S>
class SafeFloatArray {
  public:
    // returns a reference to a T
    T & operator[](int i) {
      if ( i<0 || i>=S ) {
        cerr << "Index out of bounds!" << endl;
        exit(1);
      }
      return a[i];
    }

  private:
    // Defines a 10-long array of floats
    T a[S];
};

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

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.

16 The Standard Template Library

The **Standard Template Library** (STL) 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*.

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 i^{th} 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;
// The user inputs the size via standard input
std::cin >> size;
z.resize(size);

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

Now let's create a two dimensional array.

```
int width = 5, height = 5;
std::vector< std::vector<int> > x;

x.resize(height);
for (int i = 0; i < height; i++) {
```

```
        x[i].resize(width);
}
x[2][3] = 10;
// etc.
```

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.

Other important containers included in the STL are lists and forward-lists, queues and stacks, sets and maps. A number of templated functions are also implemented as part of the STL, and are used for common operations on containers, for example, sorting or finding elements. Algorithms are not implemented as member functions of the various containers, but as free template functions that can operate on many containers.

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. For example, 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 is a pointer to an element of a container
        // this code will print the elements of v.
        cout << *it << endl;
}
```

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

17 Smart Pointers

Pointers are a strong feature of C++ that can make code efficient and lean. The difficulty with base pointers is that managing them can be a hard task and can lead to errors that are hard to spot. There are two main issues that often happen, *wild pointers* and *memory leaks*.

A *wild pointer* is one that points to an object that has already been deleted from memory. Dereferencing a wild pointer leads to unpredictable behaviour, depending on the state of the memory it is pointing to. For example, if the memory has been allocated to a different process, dereferencing a wild pointer will cause a segmentation fault. In contrast, if the program is able to write to the memory address it will corrupt unrelated data. These are subtle bugs that can be extremely difficult to find and can become critical, also in the context of security, when part of a process with system privileges.

```
#include "SafeFloatArray.h"

SafeFloatArray<float,20>* a = new SafeFloatArray<float,20>();
SafeFloatArray<float,20>* b = a;

delete(b);
a->ComputeMean(); //Runtime error, a is a wild pointer since b was deleted!
```

A *memory leak* occurs when dynamically allocated memory is not released back to the memory heap when it is no longer needed. As every dynamically allocated object must be followed by a deallocation, keeping track of this, especially in larger projects where pointers can span different scopes, becomes hard or sometimes even unclear as to who is responsible for the deallocation. Depending on the size of the allocated memory this can case the entire heap to run out of memory and the process to crash. For example:

```
#include "SafeFloatArray.h"

void memoryleak() {
        SafeFloatArray<float,20>* a = new SafeFloatArray<float,20>();
}

//infinite loop
while (1){
        memoryleak();
}
```

Smart pointers were introduced in C++11 as a way to automate memory management related to pointers and the issues outlined above. Smart pointers automatically deallocate the memory they point to when they go out of scope. These are template classes that are part of STL and wrap the raw (or base) pointers while maintaining the same syntax, overloading the * and -> operators.

STL provides tree types of smart pointers and are and defined in the <memory> header.

- `std::unique_ptr`, to represent unique ownership of a dynamically allocated resource. No other pointer can point to this object and the object is deleted when smart pointer goes out of scope. No copy constructor or copy assignment but can be moved from one `std::unique_ptr` to another.
- `std::shared_ptr`, to represent shared ownership of a dynamically allocated resource. Allows multiple references and keeps an internal *reference counter*. Once all references (use count) go out of scope (use count goes to zero) the object pointed to is deleted.
- `std::weak_ptr`, is special case of `std::shared_ptr` that holds a non-owning reference, meaning that it does not increase the reference count. This is used to point to a resource that is managed by another `std::shared_ptr`. `std::weak_ptr` models temporary ownership: when an object needs to be accessed only if it exists, and it may be deleted at any time by someone else.

18 Namespaces

Up to now in most of our examples we have used the directive `using namespace std;` in the `#inlcude` section of our example programs. A namespace is a mechanism for naming groups of definitions. This allows us to organise and arrange elements of our program (variables, functions, classes) into different logical scopes.

In addition, the use of namespaces allows the compiler to differentiate between variables, functions and classes that have the same name. This is essential for preventing name conflicts in larger projects. A namespace can be defined in several parts and is made up of the sum of its separately defined parts. The separate parts can be spread over multiple files and namespaces can be nested.

In our examples we have been using `cout` for output that is an object defined within the `std` namespace, for 'standard', in the `<iostream>` header file. Including the `using namespace std;` directive allows us to use `cout` (and all other elements within the namespace) instead of `std::cout` within that scope.

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

int number = 0;

namespace ns1 {
        int number = 1;
}
namespace ns2 {
        int number = 2;
        namespace ns3 {
        int number = 3;
        int other_number = 4;
        }
}

cout << number << endl;
cout << ns1::number << endl;
cout << ns2::number << endl;
cout << ns2::ns3::number << endl;

using namespace ns2::ns3;
cout << number << endl; //Error, reference to number is ambiguous!
cout << other_number << endl;
```

19 Error Handling

C++ provides an integrated way to handle errors with the use of *exceptions*. An exception is usually a point during a program's execution where an error has been encountered. Typically these errors can be due to wrong input, programming mistakes, or unforeseeable circumstances. For example, a network socket is not available, expected input was not received or there is not enough memory for allocation. In such cases an exception is *thrown* to signal the error to the calling function or method.

Exception handling is organized in `try` and `catch` blocks. The portion of the code that can throw an exception is enclosed in the `try` block and the exception handling is done inside the `catch` block (or blocks) that follow immediately after the `try` block. `Catch` blocks can be chained to handle different types of exceptions and a catch-all block (`catch (...)`) can be used to catch all types.

```
try {
        ... // code that can throw an exception here
}
// catch an exception of type int
catch (int e) {
        cerr << "An int exception occurred: " << e << endl;
        ... // handle int exception
}
catch (std::bad_alloc& ba) {
        cerr << "A bad allocation exception occurred: " << ba.what() << endl;
        ... // handle a bad memory allocation exception
}
catch (...) {
        cerr << "An exception occurred." << endl;
        ... // Catch any type that is not previously handled.
}
```

Throwing an exception causes the function or method to exit immediately with no return value, while all objects created inside the enclosing `try` block are destructed before control is transferred to the `catch` block.

The C++ Standard library provided base classes for declaring object to be thrown as exceptions. You can use these in your programming or derive classes to create your custom exceptions. The base exception classes are under `std::exception` defined in the `<exception>` header. All exceptions thrown by components of the C++ standard library are derived from this base exception class.

`bad_alloc`	thrown by new on allocation failure
`bad_cast`	thrown by `dynamic_cast` when it fails in a dynamic cast
`bad_exception`	thrown by certain dynamic exception specifiers
`bad_typeid`	thrown by `typeid`
`bad_function_call`	thrown by empty function objects
`bad_weak_ptr`	thrown by `shared_ptr` when passed a bad `weak_ptr`
`logic_error`	error related to the internal logic of the program
`runtime_error`	error detected during runtime

The use of exceptions separates the error handling code from the normal code and improves readability and maintainability. Allows objects to handle the exceptions they choose and can choose to pass handling to the caller as well. Note, exceptions can be nested so objects can also partially handle these and allow the caller to handle part of the exception.

20 Good Practice

In this section we cover a number of important topics on setting up and managing an efficient code base, with a view towards more complex programming tasks and projects.

20.1 Code structure

In previous sections we discussed the importance of separating the implementation of classes and their interfaces, with header files (.hpp) and implementation files (.cpp). In this section we will discuss the parts of a C++ program and the structure that C++ programs commonly follow. Note that this a general C++ structure and not a rule, some parts might not be relevant to your particular programming task.

C++ programs in general have the following structure:

- Documentation
  - It is good practice to start your source files with a block comment that gives an overview of the code to follow and describes the purpose of the program. Authorship and license related information would typically belong here as well. This is an optional section but can dramatically boost with clarity and reusability.
- Link Section
  - Here is where preprocessor directives (`# include (...)`) usually live. These are read by the preprocessor at compile time and insert the content of a user-defined or system header file into the current program.
  - Namespace definitions would typically follow, often to avoid prepending common declarations throughout the code that follows, eg. `using namespace std;`.
- Definition Section
  - Here we declare constants with a given value, eg. `#define M_PI 3.14159 /* pi */` or `#define title "Program 3"`. This creates a macro, which is an identifier and a token string, and the compiler will replace with the token string each occurrence of the identifier in the source code that follows.
  - Another useful declaration that typically lives here is `typedef`. This allows us to specify custom and compound types and use simpler names in our code, improving clarity and readability. For example, `typedef float F;`, `typedef unsigned long ulong;`, or, `typedef std::map<string, int> MarkByName;`.
- Global Declaration Section
  - Here we declare variables that have global (full program) scope. There is rarely a need to use global variables. Global variables make our programs harder to understand and to maintain.
- Function Declaration Section
  - This part of our program contains the classes and functions that we will use in our main function. In most but the simplest of programs you would import classes and functions using header files.
- Main Function, `main()`
  - Here is where the program execution starts. All statements in the body of the main function are executed and once this is done the program terminates.

Below is an example where the typical program sections are outlined:

```
// Documentation Section
/*  This is a C++ program that converts a decimal number to its binary equivalent.
        Asks the user for input and prints out the resulting binary output.
*/

// Linking Section
#include<iostream>
using namespace std;

// Definition Section
typedef int I;

// Global Declaration Section
int DecimalToBinary(int);

// Function Section
int DecimalToBinary(int decimalNum) {
    int binaryNum = 0, mul = 1, rem;
    while (decimalNum > 0) {
        rem = decimalNum%2;
        binaryNum = binaryNum + (rem*mul);
        mul = mul * 10;
        decimalNum = decimalNum / 2;
    }
    return binaryNum;
}

// Main Function
int main() {
    I decimalNum, binaryNum;
    cout << "Enter the Decimal Number: ";
    cin >> decimalNum;
    binaryNum = DecimalToBinary(decimalNum);
    cout << "\nEquivalent Binary Value: " << binaryNum << endl;
    return 0;
}
```

Often in object-oriented programs you would use some kind of managing class that maintains all other objects that interact within the program. For example, in a navigation program context, a `Navigator` object might be responsible for maintaining `TopologicalMap` objects that in turn interact with `TopologicalNode` and `TopologicalEdge` objects. Often, the managing class will have to maintain some collection of objects, such as a list of nodes or edges, and often an STL container would be used for this purpose. In many cases, `main()` creates a single instance of this managing class, handles the interaction with the user (for example, user input and options within a menu), and calls methods from the manager class object that perform the appropriate actions based on user input.

20.2 Documentation

Code documentation is an important part of writing good programs. It helps describe how your program works and helps you structure your coding as you develop your solution. Good code documentation will save time both to others on your team and the future you as sooner or later most programs will need maintenance or debugging. Nonetheless, there is no single solution when it comes to code documentation as different projects will have different needs. For example, programming as part of larger software team will often require verbose and rigid documentation written alongside the code, while for small educational projects most documentation can be part of the source itself.

A program may be written only once, but is read many times; we have seen before how comments and block comments can be used to describe parts of programs. Good documentation practice begins with picking good, descriptive variable and function and class names. Next, comments can be used to explain the logic of the code and the interaction between different modules. Anything that makes a program more readable and understandable saves lots of time, even in the short run.

There are many styles of code documentation and a number of tools one can use to automate the generation of documentation from source code. As an example, Doxygen (link) is one of the most widely used tools to generate documentation from annotated C++ sources. In most cases, source documentation or comments live in comments (`//`) or block comments (`/**... */`) and appear before the main program, before each function, and before or next to lines of code within a function.

Here is an example from the Doxygen manual:

```
//!  A test class.
/*!
  A more elaborate class description.
*/

class QTstyle_Test
{
  public:

    //! An enum.
    /*! More detailed enum description. */
    enum TEnum {
                 TVal1, /*!< Enum value TVal1. */
                 TVal2, /*!< Enum value TVal2. */
                 TVal3  /*!< Enum value TVal3. */
               }
         //! Enum pointer.
         /*! Details. */
         *enumPtr,
         //! Enum variable.
         /*! Details. */
         enumVar;

    //! A constructor.
    /*!
      A more elaborate description of the constructor.
    */
    QTstyle_Test();

    //! A destructor.
    /*!
      A more elaborate description of the destructor.
    */
   ~QTstyle_Test();

    //! A normal member taking two arguments and returning an integer value.
    /*!
```

```
        \param a an integer argument.
        \param s a constant character pointer.
        \return The test results
        \sa QTstyle_Test(), ~QTstyle_Test(), testMeToo() and publicVar()
      */
      int testMe(int a,const char *s);

      //! A pure virtual member.
      /*!
        \sa testMe()
        \param c1 the first argument.
        \param c2 the second argument.
      */
      virtual void testMeToo(char c1,char c2) = 0;

      //! A public variable.
      /*!
        Details.
      */
      int publicVar;

      //! A function variable.
      /*!
        Details.
      */
      int (*handler)(int a,int b);
};
```

This example generates this corresponding HTML documentation using Doxygen. There are a number of different possible documentation styles, the key –as with coding– is consistency and clarity. Good code documentation makes sharing, reusing and maintaining code easier.

20.3 Project Structure

When working on a larger programming project having all source files in a singe folder with no structure quickly becomes hard to manage. There is no single canonical C++ project structure but project teams often choose and follow a particular layout style. In most projects you would need to have at least 3 folders to organize your source code: `src/`, `include/` and `build/` that are described below.

A popular convention for laying out source code in a filesystem is the Pitchfork Layout (PFL). According to PFL the root folder of your project must contain the following files and folders:

- A `README` plaintext file with a description of the contents of the folder and subfolders.
- A `LICENCE` plaintext file if the project will be redistributed.
- `src/` – Main compilable source location.
- `include/` – Location of public header files.
- `lib/` – Location for potential project submodules.
- `build/` – A folder for storing ephemeral build results. This folder should be ignored when using source control that we discuss in the following sections.
- `docs/` – A folder for project documentation.

Not all of the above folders are required for all projects and there are a few more folder specifications at the PLF reference website here: link. As before, consistency and clarity are key so adopting a good codebase structure that suits the needs of your project, simplifies code management and saves you and your team time and effort.

20.4 Build System

In previous examples we have complied and linked programs using terminal commands, one compilation unit at a time. This quickly becomes impractical when dealing with projects with multiple source files. The problem of handling the compilation of a project can be addressed with the use of a *build system*.

There are many build systems available to choose from and each provide a host of different strengths and characteristics. In most cases the build system you choose to use will have a configuration file where the different parts needed for compilation and linking are defined along with a set of parameters pertinent to the build process. Popular build systems are *make* (created in 1977!), *CMake* (cross-platform make), *MSBuild*, *autotools*, *ninja*, *catkin*.

CMake is one of the most popular tools for C++. It uses the plaintext file `CMakeLists.txt` to organize all aspects of building your project. For a simple project, a three-line `CMakeLists.txt` file is all that is required. For example,

```
cmake_minimum_required(VERSION 3.10)

# set the project name
project(HelloWorld)

# add the executable
add_executable(HelloWorld hello_world.cxx)
```

can be used to compile a simple *HelloWorld* example, using the commands `cmake` and `make` in the same folder, though as previously discussed, you would want to use a `build/` folder for this. Next, you can add different folders that contain your source files, static or shared libraries, custom commands and compiler arguments, paths for installation and system introspection. You can read through the CMake tutorial for examples of the different features CMake provides, here. An example of a simple multi-folder `CMakeFile.txt` is below.

```
cmake_minimum_required(VERSION 3.1)
project(directory_HelloWorld)

#Headers folder, such as helloworld.h
include_directories(include)

#Add the sources using the set:
set(SOURCES src/helloworld.cpp src/other_source_file.cpp)

add_executable(HelloWorld \${SOURCES})
```

20.5 Coding Style

We have underlined how important consistency and clarity are when it comes to writing excellent code. The problem is that there is no single coding standard for C++. Programmers often develop their own style and adapt the style of the team they are working with or the project they are working on.

There exist a few popular coding styles that provide a guidelines for your coding practice, for example, the C++ Foundation provides core guadlines here. Another popular C++ style guide is the one developed by Google and publicly accessible here. One thing to note is that you can use a code linter (a static code analysis tool) to warn you about or correct you code stylistic error. In the case of the Google C++ Style Guide, the linter –`cpplint`– provided is a python script that would examine your code files and generate a list of points to consider or edit.

20.6 Version Control

Version control, or source control, is the process through which you keep track of changes made to the source code of your project. This allows us to record the progress of the code development and to effortlessly collaborate on team projects, merging contributions while, for example, keeping a running

version separate. In addition, for each contributor of the project a different working copy is -in most cases- maintained, providing redundancy in case of hardware failure or similar trouble. Overall, all changes to the codebase are tracked and who, what, why and when is recorded.

There are a few version control systems available, for example, Git, CVS, SVN, Mercurial, and others, while Git is the most popular and widely used system. Version control works for any body of text-based files, so this is not a C++ specific tool but something to definitely use for all your Python and Matlab work as well.

Git keeps a complete copy of the repository, including the record of all changes, in every working directory. The basic steps in using Git are to *modify* files in your working directory, to *stage* the files (adding snapshots of them to your staging area), and to *commit* the files in the staging area, storing that snapshot permanently to your Git directory.

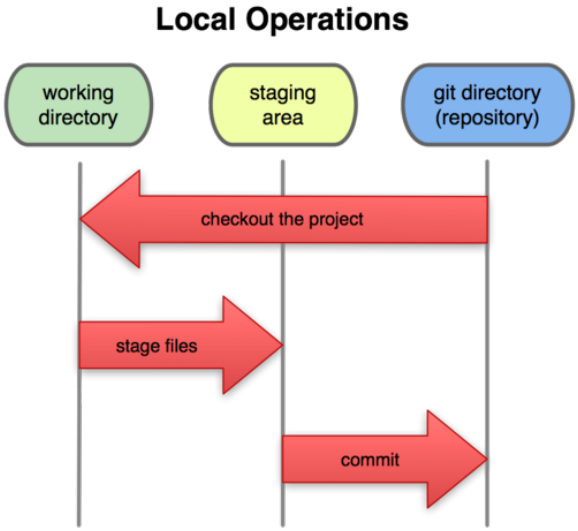

Figure 10: The three basic states of files in your Git repository.

Below is a summary of a minimal Git workflow:

- clone remote repository
- add you changes to the staging area
- commit those changes to the git directory
- push your changes to the remote repository
- pull remote changes to your local working directory

21 Summary

The following paragraph recaps the key concepts from Object Oriented Programming (OOP) that you need to know and be comfortable working with (for writing and understanding small programs). *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. *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, parametrised, class definition. This is an example of *compile-time polymorphism*.

Object Oriented Programming (OOP) is an important, mainstream approach to programming and program design. It is essential to know about OOP if you are going to work on any large programming tasks with existing libraries or with large teams, since OOP will be used in a large proportion of these cases. However, despite its popularity OOP is not the best approach to all problems. And in many languages and code-bases developers mix OOP with a procedural style. Also *functional programming* is becoming increasingly important in highly parallel or high reliability applications, due to the ability to more easily prove properties of functional programs.

The *Standard Template Library* (STL) is a collection of templated classes and functions that implement functionality commonly used in programming tasks. The STL provides implementations of *containers* (sequence and and associative), *iterators* and *algorithms*. *Smart pointers* provide a way to automate memory management related to pointers, helping to avoid problems as memory leaks and instances of wild pointers.

Namespaces are used to logically group and organise elements of a program. Good *documentation* allows us to more easily share and reuse code. Much of the task of documenting your code can be automated using comments that are part of your code source. A *build system* automates the the process of compiling and linking your code to an executable (or a library). *Version control* is the process of managing and keeping track of changes made to a project. Consistency and clarity are key to writing good code, and there are guidelines for the structure of your code, the structure of your project and your coding style.

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 "Object-oriented abstractions map neatly onto the domains of certain specific kinds of programs, like simulations and CAD systems." But he also 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."