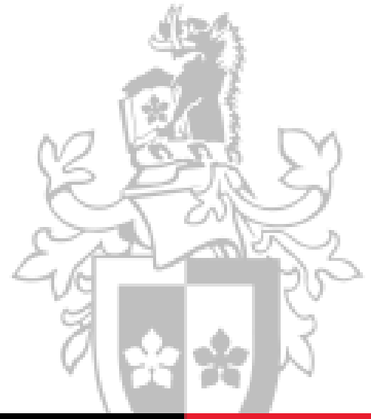
Introduction to C++

by Willem van Straten and Andrew Cain



SWINBURNE UNIVERSITY OF TECHNOLOGY

Object oriented programming involves creating objects that know and do things

Syntax is secondary to good design through application of OO Principles

When you understand the principles, you can understand any OO language

See that syntax is similar by learning a second language

Learning a second (human) language teaches you to appreciate the grammar of your native tongue.

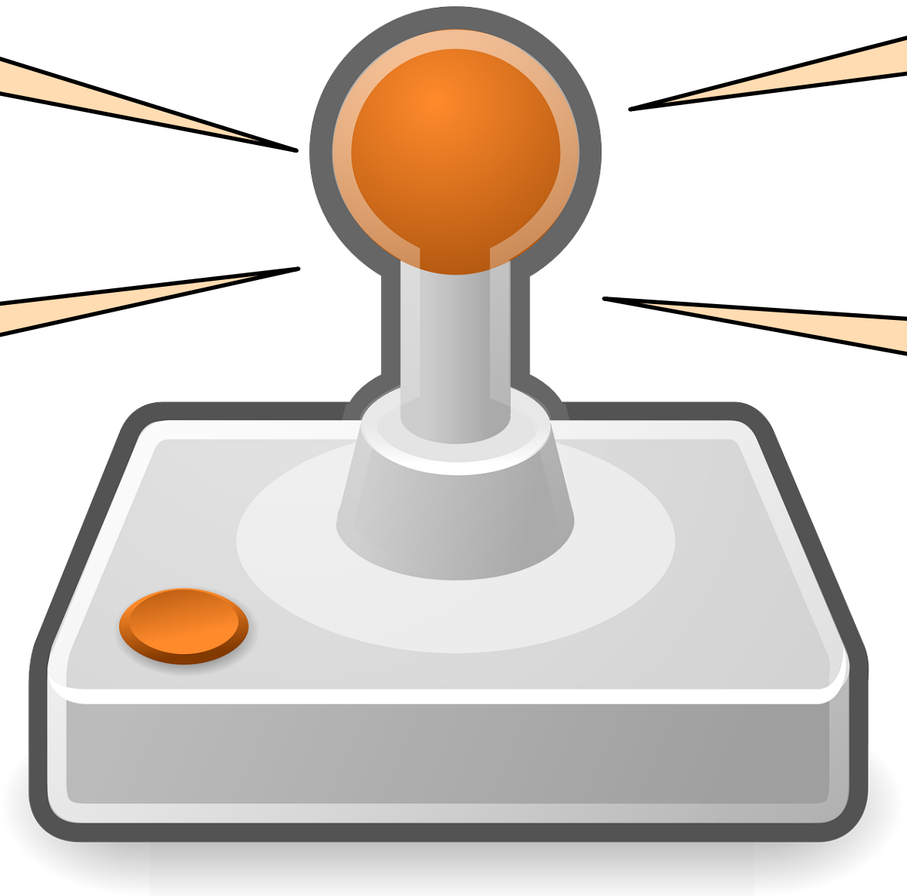
Why C++?

Available on every platform

High Performance Computing (including games)

Digital Signal Processing

Large Scale Simulations



Graphics Processing

Real-time systems

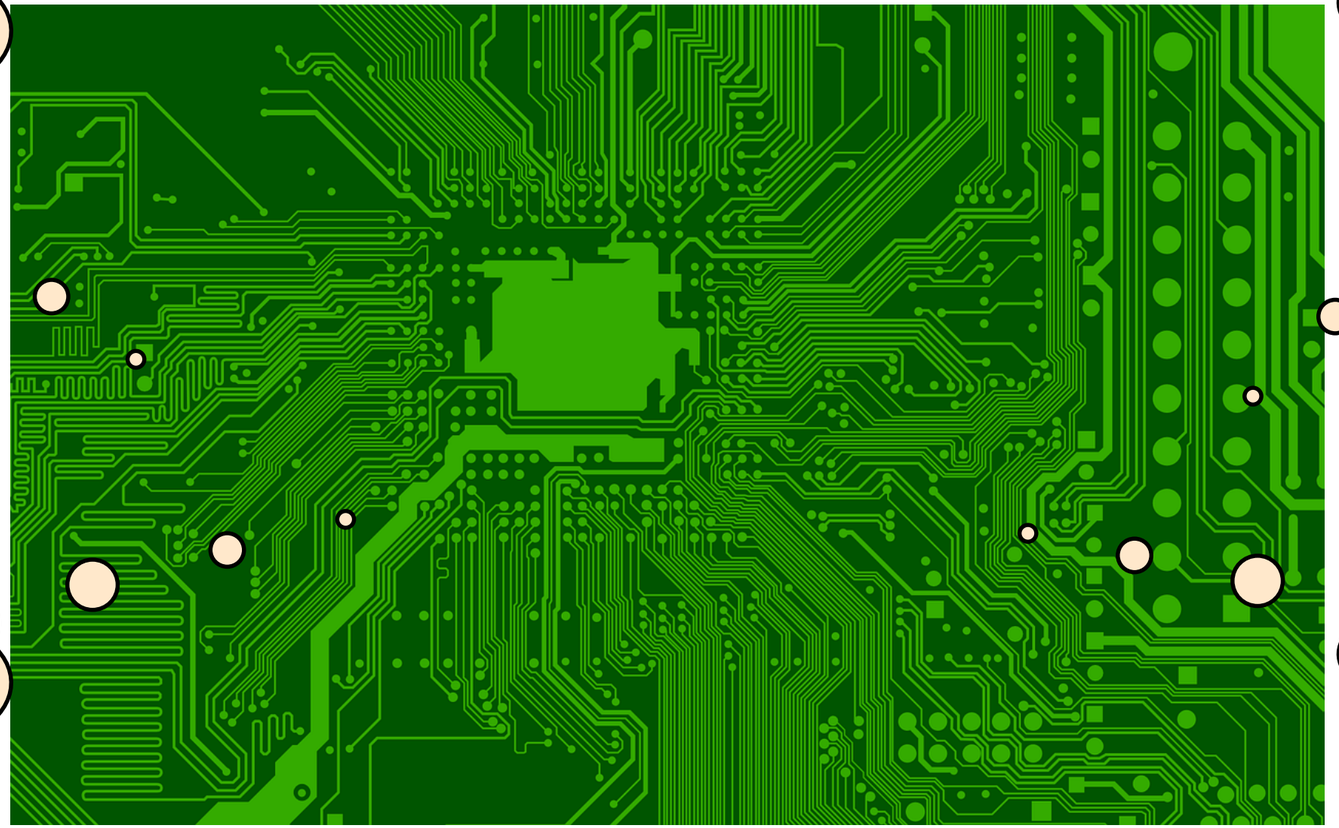
Energy Efficient Computing (including mobile apps)

- C++ programming languages are more energy efficient than others because they simply run faster
- A program that can be executed faster reduces the load on a CPU, which in turn demands less energy from the massive data centers that pull electricity from the grid.

Embedded Systems (including device drivers)

Microcontroller

Embedded PCs



RTOSes

Toasters

Flexibility

C++ is a multi-paradigm language:

- *Object-oriented programming*
- *Procedural programming*
- *Generic programming*

Legacy

C++ can utilize code written in C and Fortran. They have powerful C++ libraries: e.g. Standard Template Library (STL), QT GUI library, Eigen Matrix Library for linear algebra

With great power comes great responsibility

- François-Marie Arouet

Resource Management

e.g. any object created with `new`
must also be destroyed with `delete`

There is no Garbage Collector

Flexibility = Decision = Debate

(too) many ways to achieve the same objective

Different design teams may choose different paradigms, styles, conventions, policies, etc.

Let’s have a look at
the **principles of OOP**
expressed in **C++**

Encapsulation 1.

complete separation between interface and implementation

```
class Student
{
   string name;
   string id;
public:
   string getDescription ();
};
```

all classes are public

Declaration of interface in header file (e.g. Student.h)

`#include` interface

```
#include “Student.h”

string Student::getDescription ()
{
   return name + “ “ + id;
}
```

Definition/implementation in source file (e.g. Student.cpp)

Encapsulation 2.

public, protected, and private keywords control access to members

```
class Student
{
private:
   std::string name;
   std::string id;

public:
   string getDescription ();
   void getName ();
};
```

Everything declared after `private:` is private.

Everything declared after `public:` is public.

Encapsulation

Recap on Definition:

“Encapsulation is the process of combining data and function into a single unit called class.”

“Encapsulation is the mechanism that binds code and data together and keeps them safe and away from outside inference and misuse.”

Box.h (Header file)

```
#include<iostream>
using namespace std;
class Box{
   private:
      int length;
      int breadth;
      int height;
   public:
      //constructor
      Box(int l, int b, int h);
      //setter
      void setLength(int l);
      //getter
      int getLength();
      //calculate volume
      int volume();

};
```

Box.cpp (Implementation)

```
#include "Box.h"
// constructor (implementation)
Box::Box(int l, int b, int h){
   length = l;
   breadth = b;
   height = h;
}
//setter (implementation)
void Box::setLength(int l){
   length = l;
}
//getter (implementation)
int Box::getLength(){
   return length;
}
//calculate volume (implementation)
int Box::volume(){
   return (length*breadth*height);
}
```

Main.cpp

```
#include<iostream>
using namespace std;
int main(){
   //creating a Box object
   Box  *b1 = new Box(2,3,4);
   //calculating and display the volume of Box b1
   cout<<"\nThe volume of Box b1 is " << b1->volume();
   //change the length of Box b1
   b1->setLength(10);
   //calculating and display the volume of Box b1
   cout<<"\nThe new volume of Box b1 is " << b1->volume();
   return 0;
}
```

Inheritance

```
class Rectangle : public Shape
{
public:
  void draw ();
};
```

implicit `override`

Note that inheritance is always public.

Protected and private use of a base class do not represent an inheritance relationship

```
class Parent{
   public:
        int a;
   protected:
        int b;
   private:
        int c;
};

class ChildX : public Parent{
   // a is public
   // b is protected
   // c is not accessible from
      ChildX
};
```

```
class ChildY : protected Parent{
   // a is protected
   // b is protected
   // c is not accessible from
      ChildY
};

class ChildZ : private Parent{
   // a is private
   // b is private
   // c is not accessible from
      ChildZ
};
```

Inheritance

Recap on definition:

“Inheritance is a technique used in OOP that one object acquires the properties of another object without redefining in order to create well-defined class.”

“When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class.”

Shape.cpp

```
//Base class
class Shape{
   protected:
      int width;
      int height;
   public:
      void setWidth(int w){
         width=w;
      }
      void setHeight(int h){
        height=h;
      }
};
```

Rectangle.cpp

```
//Derived class
class Rectangle:public Shape{
   public:
      int getArea(){
          return (width*height);
      }
};
```

Main.cpp

```
//Main Program
int main(){
     Rectangle Rect;
     Rect.setWidth(5);
     Rect.setHeight(7);
     //Print the area of Rect
     cout<<"Total area:"<<Rect.getArea();
     return 0;
}
```

Polymorphism

Poly = many
Morph = form

Polymorphism - base class

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

destructor must be virtual

pure virtual (abstract) method

The destructor of a base class that is used in a polymorphic design must be declared as virtual.

A pure virtual method has no implementation.

```
class Polygon{
   protected:
      int width, height;
   public:
      void set_values(int a, int b){
         width = a;
         height = b;
      }
      virtual int area() = 0;
};


class Rectangle : public Polygon{
   public:
      int area(){
         return (width*height);
      }
};
```

```
class Triangle : public Polygon{
   public:
      int area(){
         return (width*height/2);
      }
};


int main(){
   Rectangle rect;
   Triangle trg;
   Polygon * poly1 = &rect;
   Polygon * poly2 = &trg;
   poly1->set_values(4,5);
   poly2->set_values(4,5);
   cout<<poly1->area()<<"\n";
   cout<<poly2->area()<<"\n";
    return 0;
}
```

Abstraction

Abstraction is an OOP design principle that is expressed in the concepts and syntax of encapsulation and inheritance.

Abstraction

- The concept of abstraction relates to the idea of hiding data that is not needed for presentation.
- It is to give a clear separation between properties of data type and the associated implementation details.
- This separation is achieved in order that the properties of the abstract data type are visible to the user interface and the implementation details are hidden.
- Thus, abstraction forms the basic platform for the creation of user-defined data types called objects.

Implement the interface concept using pure virtual classes

A pure virtual class has ***only*** pure virtual methods

```
class IHaveInventory
{
public:
  virtual Item* find (string& name) = 0;
};
```

Understand new syntax: pointers, references, and streams

Explicit pointer syntax: choice between static and dynamic

```
void function ()
{
   Student A;
   string nameA = A.getName();

   Student* B = new Student;
   string nameB = B->getName();
   delete B;
}
```

static: object is automatically deleted when it goes out of scope

dynamic: memory allocated on heap must be explicitly deleted

References

```
void update_text (string& text)
{
   text = “hi!”;
}

[...]

string my_string;

update_text (my_string);

cout << my_string << endl;
```

reference to string object

Why not use a pointer?

`my_string` is modified

prints `hi!`

Value, Reference, and Pointer

```
int a = 4;
```

a is an integer
equal to the value **4**

```
int& b = a;
```

b is a reference to an integer
that refers to the variable **a**

```
int* c = &a;
```

c is a pointer to an integer
equal to the address of **a**

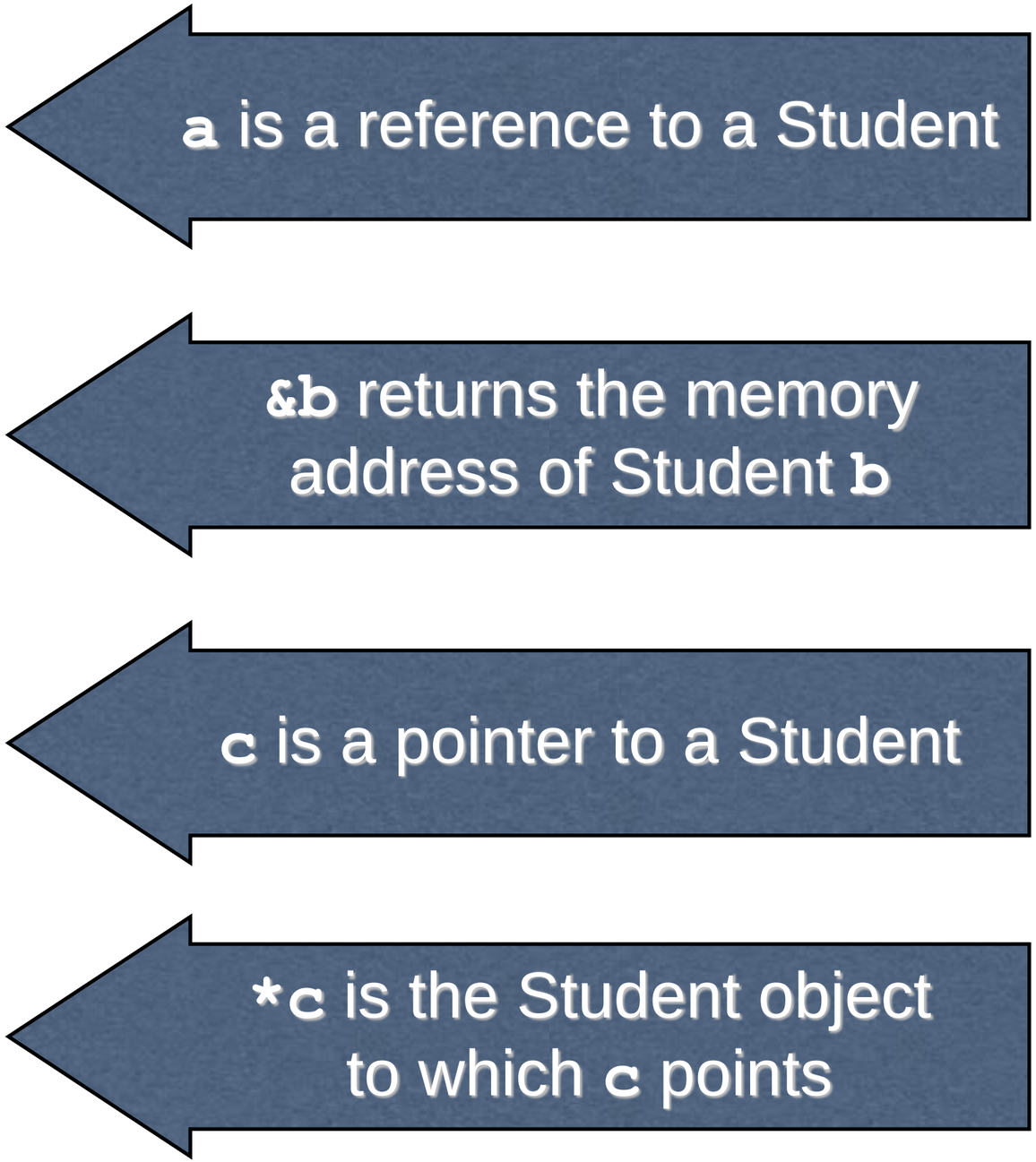
```
int d = *c;
```

d is an integer equal to
the value to which **c** points

```
Student a (“Rusell Alan Hulse”);

Student& b = a;

Student* c = &a;

Student* d = new Student(a);
```

What is the fundamental difference between the last two lines of code?

```
Student& a;
Student b;
&b;

Student* c;

*c;
```

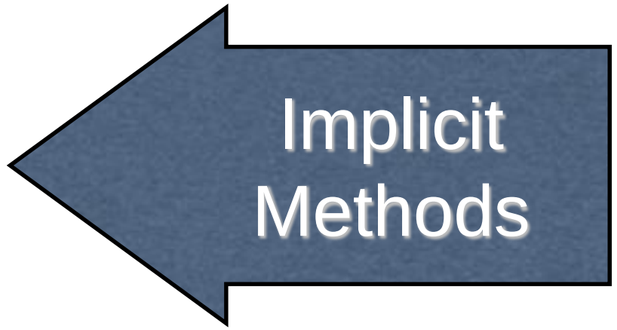


References	Pointers
`Type& ref = var;`	`Type* ptr = new Type;`
No need to dereference	Must be dereferenced
Must be declared with value	Can be initialized to NULL
Not suitable for arrays	Single object or array of objects
Always refers to the same object	Can be redirected to new object
Implicit temporary objects ok	No implicit temporary objects

To smoothly transit from C#
use pointers to objects in C++

Understand class syntax: implicit methods

```
class MyClass
{
stream operators as friends (optional)

public:
    default constructor
    copy constructor
    destructor
    assignment operator
    other constructors
    public methods
    field access methods

private:
    attributes (fields)
    private methods
};
```



Implicit Methods

```
void function ()
{
   MyClass A;

   MyClass B (A);

   A = B;

}
```

Default Constructor

Copy Constructor

Assignment Operator

Destructor (x 2)

The Standard C++ Library and Standard Template Library

Everything in `std` namespace

```
#include <iostream>

[...]

std::cout << “Hello, World!” << std::endl;
```

OR

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

[...]

cout << “Hello, World!” << endl;
```

Everything in `std` namespace

```
#include <string>

[...]

std::string name = “Anthony Hewish”;
```

OR

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

[...]

string name = “Jocelyn Bell”;
```

C++ Basic I/O

C++ uses a convenient abstraction called streams to perform input and output operations in sequential media such as the screen, the keyboard or a file.

Standard output (cout)

Examples:

```
    cout << "Hello World"; // prints Hello World on screen
    cout <<120;           //prints number 120 on screen
    cout << x;            // prints the value of x on screen
```

Standard input (cin)

Examples:

```
    int age;
    cin >> age;
```

Standard Template Library (STL)

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

[...]

// container class defined in STL
vector<string> names;

names.push_back(“Arno Allan Penzias”);
names.push_back(“Robert Woodrow Wilson”);

// vector class supports array notation
cout << names[0] << “ and “ << names[1];
```

Standard Template Library Concepts

Containers

- store elements of the template argument type

Algorithms

- common computational tasks performed on containers
- Allow initialization, sorting, searching, and transforming of the contents of containers

Iterators

- independent access to container elements
- To step through the elements of collections of objects

C++ Standard Template (Vector Container)

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
    //create vector to store int
    vector<int> vec;

    //display the original size of vec
    cout<<"Vector size="<<vec.size()<<endl;

    //push 5 values into the vector
    for(int i=0; i<5; i++){
         vec.push_back(i);
    }

    //display extended size of vec
    cout<<"Extended vector size="<<vec.size()<<endl;

    //access 5 values from the vector
    for(int i=0; i<5; i++){
         cout<<"Value of vec["<<i<<"]="<<vec[i]<<endl;
    }
    //cont.. NEXT SLIDE
```

```
    // use iterator to access the values
    vector<int>::iterator v = vec.begin();
    while(v != vec.end()){
          cout<<"Value of v = "<<*v<<endl;
          v++;
    }

    return 0;
}
```

```
vector size = 0
extended vector size = 5
value of vec [0] = 0
value of vec [1] = 1
value of vec [2] = 2
value of vec [3] = 3
value of vec [4] = 4
value of v = 0
value of v = 1
value of v = 2
value of v = 3
value of v = 4
```

Let’s review the differences

C++ is a low level and indeed platform neutral programming language

C# is a high level language that is component oriented

C++ allows any type to be passed by value, reference or pointer

C# does not have the concept of function pointers

In C++ the memory that is allocated in the heap dynamically has to be explicitly deleted

In C#, memory management is automatically handled by garbage collector

In C++, the end of the class definition has a closing brace followed by a semicolon

In C#, the end of the class definition has a closing brace alone.

C++ does not contain *for each* statement (only having for, while and do…while)

In C#, has another flow control statement called for each

C++ uses stream operators to simplify file I/O

C++ templates
are more flexible and powerful than
C# generics

Review the similarities

C++ supports encapsulation, inheritance, polymorphism (and abstraction)

C++ and C# share syntax based on C

Any Questions?

Enjoy programming in C++!

This Week's Tasks

^ Supplementary Exercise - C++ Classes and Inheritance

^ Supplementary Exercise - Planetary Rover UML Class Diagram

Supplementary Exercise - Planetary Rover Code

** Credit Task 2 - The Electoral System (Implementation)

^ Useful exercises as part of preparation for Pass Task 13 in Week 10
*** Compulsory for Credit and Higher Grade*