Introduction to C++

by Willem van Straten and Andrew Cain

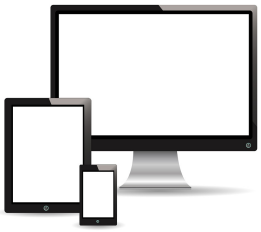
Object oriented programming involves creating objects that know and do things

Syntax is secondary to good design through application of OO Principles

See that syntax is similar by learning a second language

Why C++?

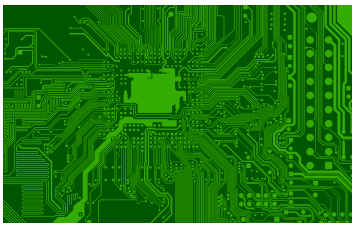
Available on every platform



High Performance Computing
(including games)

Energy Efficient Computing
(including mobile apps)

Embedded Systems
(including device drivers)



Flexibility

With great power comes great responsibility

- François-Marie Arouet

Resource Management

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

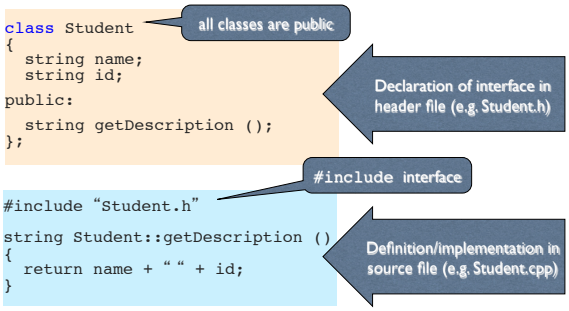
Flexibility = Decision = Debate

(too) many ways to achieve
the same objective

See the principles of OOP
expressed in C++

Encapsulation 1.

complete separation between interface
and implementation



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.

Inheritance

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

implicit override

Polymorphism - base class

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

destructor must be virtual

pure virtual (abstract) method

See `polymorphic_dtor.cpp`

Polymorphism - subtype

```
void work_with_shapes (Shape* arg)
{
  arg->draw ();
}
[...]
Rectangle* rect = new Rectangle;
work_with_shapes( rect );
```

draws a Rectangle

Abstraction

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;
};
```

Use multiple inheritance ***only*** when implementing an interface

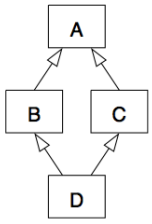
```
class Player : public GameObject,
               public IHaveInventory
{
public:
  // implements IHaveInventory::find
  Item* find (string& name);
};
```

Even Better: use the Adapter pattern

```
class PlayerHasInventory : public IHaveInventory
{
  Player* adaptee;
public:
  PlayerHasInventory (Player* to_adapt);
  Item* find (string& name)
  { return adaptee->find(name); }
};
```

inline method definition

Multiple Inheritance: Diamond Problem



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!

I/O Streams

C#

```
while ((input = sr.ReadLine()) != null)
{
  char[] delim = { ' ', '\t' };
  string[] words = input.Split(delim);

  mass = Convert.ToDouble(words[0]);
  abundance = Convert.ToDouble(words[1]);
}
```

C++

```
while (input)
{
  input >> mass >> abundance;
}
```

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;
```

a is a reference to a Student

&b returns the memory address of Student b

c is a pointer to a Student

*c is the Student object to which c points

References enable automatic storage

```
// count occurrences of character in string
unsigned count (const std::string& text, char c);
[...]
std::string program = "Hockey Night in Canada";
unsigned a_count = count (program, 'a');
unsigned o_count = count ("Toronto Maple Leafs",'o');
```

a temporary std::string object is automatically constructed and destroyed on return

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 and stream operators

```
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

Implicit Methods

```
void function ()
{
  MyClass A;
  MyClass B (A);
  A = B;
}
```

Default Constructor

Copy Constructor

Assignment Operator

Destructor (x 2)

```
class Student
{
   friend istream& operator >> (istream&, Student&);
   friend ostream& operator << (ostream&, const Student&);

public:
   // accessors and modifiers
   const string& getName () const;
   void setName (const string&);

private:
   // attributes
   string name;
   string id;
};
```

extraction operator

insertion operator

no properties

```
const string& Student::getName () const
{
  return name;
}

void Student::setName (const string& _name)
{
  name = _name;
}
```

return value cannot be modified by the caller

method will not modify state of object

passing by reference avoids copy constructor

```
istream& operator >> (istream& in, Student& student)
{
  in >> student.name >> student.id;
  return in;
}

ostream& operator << (ostream& out, const Student& st)
{
  out << st.name << " " << st.id << endl;
  return out;
}
```

stream operators always return reference to stream argument

The Standard C++ Library and Standard Template Library

Everything in `std` namespace

```
#include <iostream>
[...]
std::cerr << "Hello, World!" << std::endl;
```

OR

```
#include <iostream>
using namespace std;
[...]
cerr << "Hello, World!" << endl;
```

Everything in `std` namespace

```
#include <string>
[...]
std::string name = "Anthony Hewish";
```

OR

```
#include <iostream>
using namespace std;
[...]
string name = "Jocelyn Bell";
```

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
iterators
implementation-independent access to container elements
algorithms
common computational tasks performed on containers
functors
customize algorithm and container behaviour

Review the differences

C++ has no interfaces,
no properties, and
no explicit overrides

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

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

Enjoy programming in C++!

This Week's Tasks

Pass Task 15: Plantary Rover UML Class Diagram

Pass Task 16: Planetary Rover Code

Pass Task 17: Case Study – Iteration 3