C++: Interfaces, Templates, Operator Overload & Exceptions

David Marshall

School of Computer Science & Informatics
Cardiff University

CM2204

All Lecture notes, code listings on CM2204 Web page
C++ allows you to hide the implementation details of a class. By hiding the implementation details

▶ Your program is forced to go through the interface routines your class provides.

▶ when you change the implementation, all you have to do is make whatever changes are necessary to the class’s interface, without having to modify the rest of your program.

### Access Specifier

C++ allows you to assign an access specifier to any of a class’s data members and member functions.
Access Specifier

The access specifier defines which of your program’s functions have access to the specified data member or function. The access specifier must be

- **public** — complete access to the member function or data member, limited only by scope.

- **private** — access to it is limited to member functions of the same class

- **protected** — as private but data member or function can also be accessed by a class derived from the current class or friend.
C++ Interfaces: abstract classes

C++ interfaces implemented using abstract classes

- **not** be confused with **data abstraction** which is a concept of keeping implementation details separate from associated data.

A class is made **abstract** by declaring at least one of its functions as a **pure virtual** function (See previous lecture).

Recap: A pure virtual function is specified by placing "== 0" in its declaration as follows:

```cpp
class Box
{
    public:
        // pure virtual function
        virtual double getVolume() = 0;
    private:
        double length; // Length of a box
        double breadth; // Breadth of a box
        double height; // Height of a box
};
```
Abstract class

- The purpose of an **abstract** class is to **provide** an **appropriate base** class from which **other classes** can **inherit**.
- **Abstract** classes **cannot** be used to instantiate objects and serves only as an **interface**.
  - Attempting to **instantiate** an object of an **abstract** class **causes** a **compilation error**.
- Thus, if a **subclass** of an **abstract** class needs to be **instantiated**,
  - it has to implement each of the **virtual** functions, which means that it supports the interface declared by the **abstract** class.
  - Failure to **override** a **pure virtual** function in a **derived** class, then attempting to **instantiate objects** of that **class**, is a **compilation error**.
- Classes that **can** be used to **instantiate** objects are called **concrete classes**.
Consider the following example where parent class provides an interface to the base class to implement a function called `getArea()`:

```cpp
// Base class
class Shape
{
    public:
        // pure virtual function providing interface framework.
        virtual int getArea() = 0;
        void setWidth(int w)
        {
            width = w;
        }
        void setHeight(int h)
        {
            height = h;
        }
    protected:
        int width;
        int height;
};
```
// Derived classes
class Rectangle: public Shape
{
public:
    int getArea()
    {
        return (width * height);
    }
};
class Triangle: public Shape
{
public:
    int getArea()
    {
        return (width * height)/2;
    }
};

int main(void)
{
    Rectangle Rect;
    Triangle Tri;

    Rect.setWidth(5);
    Rect.setHeight(7);
    // Print the area of the object.
    cout << "Total Rectangle area:" << Rect.getArea() << endl;

    Tri.setWidth(5);
    Tri.setHeight(7);
    // Print the area of the object.
    cout << "Total Triangle area:" << Tri.getArea() << endl;

    return 0;
}
When you design a class, you’re forced to make some decisions about the data types that make up that class.

▶ For example: If your class contains an array, the class declaration specifies the array’s data type.

In the following class declaration, an array of shorts is implemented:

```cpp
class Array
{
private:
    short arraySize;   // Number of array elements
    short *arrayPtr;   // Pointer to the array

public:
    Array(short size); // Allocate an array
                        // of size shorts
    ~Array();          // Delete the array
};
```
In this class,

- the constructor allocates an array of arraySize elements, each element of type short.
- The destructor deletes the array.
- The data member arrayPtr points to the beginning of the array.
- To make the class truly useful, you’d probably want to add a member function that gives access to the elements of the array.
What happens when you decide that an Array of **shorts** is not what you need?

Perhaps you need to implement an array of **longs** or, even better, an array of your own data types.

One approach you can use is to make a **copy** of the Array class (member functions and all) and change it slightly to implement an array of the appropriate type. E.g., An Array class designed to work with an array of **longs**:

```cpp
class LongArray
{
    private:
        short arraySize; // Number of array elements
        long *arrayPtr;  // Pointer to the array

    public:
        LongArray( short size ); // Allocate an
                                  // array of size
                                  // longs

        ~LongArray( void );  // Delete the array
};
```
Not a great idea?

There are definitely problems with this approach:

▶ You are creating a maintenance nightmare by duplicating the source code of one class to act as the basis for a second class.
▶ Suppose you add a new feature to your Array class.
▶ Are you going to make the same change to the LongArray class?

Templates to the rescue:

C++ templates allow you to parameterise the data types used by a class (also a function).

▶ Instead of embedding a specific type in a class declaration, you provide a template that defines the type used by that class.
C++ Template Definition

Here’s a **templated** version of the **Array** class:

```cpp
template <class T>

class Array
{
private:
    short arraySize;
    T *arrayPtr;
public:
    Array( short size );
    ~Array( void );
};
```

- The keyword **template** tells the compiler that what follows is not your usual, run-of-the-mill **class** declaration.
- Following the keyword template is a pair of angle brackets (**<>**), that surround the **template**’s template argument list.
  - This list consists of a series of **comma-separated arguments**
  - **one** argument is the **minimum**
Using a template definition

Once your class template is declared,

- You can use it to create an object.
- When you declare an object using a class template you have to specify a template argument list along with the class name.

Here’s an example:

```cpp
Array<long> longArray(20);
```

- The compiler uses the single parameter, long, to convert the Array template into an actual class declaration.
- This declaration is known as a template instantiation.
- The instantiation is then used to create the longArray object.
The **template** technique can also be applied to **functions**, E.g.:

```c++
template <class T, class U>
T MyFunc( T param1, U param2 )
{
    T var1;
    U var2;
    .......
}
```
Consider an example program that provides a basic demonstration of class and function templates.

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

const short kNumElements = 10;

// Array

template <class T>
class Array
{
  private:
    short arraySize;
    T *arrayPtr;
    T errorRetVal;

  public:
    Array( short size );
    ~Array();
    T &operator[]( short index );
};
```
A Template Example Cont.

```cpp
// template.cpp cont.

template <class T>
Array<T>::Array( short size )
{
    arraySize = size;
    arrayPtr = new T[ size ];
    errorRetValue = 0;
}

template <class T>
Array<T>::~Array()
{
    delete[] arrayPtr;
}

template <class T>
T &Array<T>::operator[]( short index )
{
    if ( ( index < 0 ) || ( index >= arraySize ) )
    {
        cout << "index out of bounds( " << index << " )\n";
        return( errorRetValue );
    }
    else
        return( arrayPtr[ index ] );
}
```
A Template Example Cont.

```cpp
// Template Example Cont.

// Template Example Cont.

// Template Example Cont.

template <class T>
T Power( T base, T exponent )
{
    T i, product = 1;

    for ( i=1; i<=exponent; i++ )
        product *= base;

    return( product );
}
```
```cpp
int main()
{
    Array<short> myRay(kNumElements);
    Array<long> myLongRay(kNumElements);
    short i, shortBase = 4;
    long longBase = 4L;

    for (i = 0; i <= kNumElements; i++)
        myRay[i] = Power(shortBase, i);

    cout << "−−−−\n";

    for (i = 0; i <= kNumElements; i++)
        cout << "myRay[" << i << "]: " << myRay[i] << "\n";

    for (i = 0; i <= kNumElements; i++)
        myLongRay[i] = Power(longBase, (long)i);

    cout << "−−−−\n";

    for (i = 0; i <= kNumElements; i++)
        cout << "myLongRay[" << i << "]: " << myLongRay[i] << "\n";

    return 0;
}
```
The Output is as follows:

index out of bounds(10)

--

myRay[0]: 1  
myRay[1]: 4  
myRay[2]: 16 
myRay[3]: 64  
myRay[4]: 256 
myRay[5]: 1024 
myRay[6]: 4096 
myRay[7]: 16384 
myRay[8]: 0  
myRay[9]: 0

--

myLongRay[0]: 1 
myLongRay[1]: 4 
myLongRay[2]: 16 
myLongRay[3]: 64  
myLongRay[4]: 256  
myLongRay[5]: 1024 
myLongRay[6]: 4096 
myLongRay[7]: 16384 
myLongRay[8]: 65536 
myLongRay[9]: 262144
main points

- **Array** features three data members, all of them `private`.
  - `arraySize` is the number of elements in the array;
  - `arrayPtr` points to the beginning of the array;
  - `errorRetValue` is `identical` in type to one of the array elements and comes into play when you try to exceed the bounds of the array.

- **Array()** constructor allocates memory for the array (**more shortly**)

- The **destructor** deletes the allocated memory, and `operator[]()` is used to implement bounds checking

- The constructor uses its parameter, `size`, to initialize `arraySize`. Then, an array of size elements of type `T` is allocated. Finally, `errorRetValue` is initialized to zero.

- The destructor uses `delete` to delete the memory allocated for the array. (**more shortly**)
operator[]() is called whenever an Array element is accessed via the [] operator.

- operator[]() first checks to see whether the index is out of bounds.
  - If it is, an error message is printed and the pseudo-element, errorRetValue, is returned
  - If the index is in bounds, the appropriate element of the array is returned.

- Power() is the templated function:
  Power() raises the parameter base to the exponent power, and the final result is returned.

  - Declared using the template keyword and a single template type, T.
  - Power() takes two parameters of type T and returns a value of type T
    (Note: the type of the two parameters must match exactly).
main() starts by defining a short version of Array and a long version of Array

- Note: you could have declared a class named EraserHead and used Array to create an array of EraserHeads.

A loop then fills the short array with consecutive powers of 4:

- When i is equal to kNumElements, the array runs out-of-bounds, causing an error message to be printed on the console.

Next, a separator line is sent to the console and another loop prints the value of each element in the short array:

By the time we get to Power( 4, 8 ) we’ve reached the limits of a signed short.
C++ allows you to designate a class or a single member function as a `friend` to a specific class.

- This allows only certain functions to access data.

Classes can grant access to their `private` member data and functions to their `friends`

- The class still maintains `control` over which classes and functions have access
- Friends of a class are treated as class members for access purposes, although they are `not` members
- Declare your friends within your class body and use the keyword `friend`
Example Friend Definition

```cpp
//− Payroll

class Payroll
{
    // Data members...
    private:

    // Member functions...
    public:

        Payroll();
    ~Payroll();
    void PrintCheck( Employee *payee );

};
```
```cpp
//— Employee
class Employee {
    friend class Payroll;

    // Data members...
    private:
        char employeeName[kMaxNameSize];
        long employeeID;
        float employeeSalary;

    // Member functions...
    public:
        Employee(char *name, long id, float salary);
        ~Employee();
        void PrintEmployee();
};
```
Three Types of Friends

There are three ways to designate a friend.

▶ You can designate an entire class as a friend to a second class.

▶ You can also designate a specific class function as a friend to a class.

```cpp
class Employee
{
  friend void Payroll::PrintCheck(
    Employee *payee);
};
```

▶ You can also designate a nonmember function as a friend.
  ▶ For example, you could designate main() as a friend to the Employee class:

```cpp
class Employee
{
  friend int main();
};
```
C++ Overloading

**Function overloading**

- Change meaning of function according to the types of the parameters
  - Seen some examples already

**Operator overloading**

- Change meaning of operator according to the types of the parameters
- Perfectly sensible to do if operators are declared to behave sensible
  (e.g. generalisations of +, *, etc. from maths very well known, e.g. matrices)
- You can redefine them strangely, but should not
  (e.g. equality to mean inequality or no relation at all)
C++ Overloading

Non-member and member operator overloading possible

- Access rights like any other function
  (friend, private, protected, public)
- As non-member parameters are operands from left to right
- As member left-hand side is the object operator is acting upon
- Parameter types can differ from each other
  (only use this if it really makes sense)
Operator Overloading: Restrictions

- You cannot change an operator’s precedence
- You cannot create new operators
- You cannot provide default parameter values
- You cannot change number of parameters
- You cannot override some operators: :: sizeof ?: .
- You must overload +, +=, ==, !, etc. separately
- If overloaded, these can only be member functions: = , [] ->
- Postfix and prefix ++ and -- are different (postfix has an unused int parameter)
Any method whose name follows the form:

```
operator <C++ operator>
```

is said to **overload** the **specified operator**.

- When you overload an operator, you’re asking the compiler to call your function **instead** of interpreting the operator as it normally would.
Operator Overload Example: Adding two complex numbers

Consider the following code fragment:

```cpp
Complex a(1.2,1.3);  // complex numbers class, 2
Complex b(2.1,3);    // parameters: real + imaginary parts
Complex c = a+b;      // Need addition operator overloaded
```

The addition **without** having **overloaded operator** `+` could look like this:

```cpp
Complex c = a.Add(b);
```

**However**

- This piece of code **not** that readable and natural –we’re **dealing with numbers**
- **Note:** Programmers often **abuse** this technique, when the concept **not related** to the natural use of `+` and `-` to add and remove elements from a data structure, for example.
  In this case **operator overloading** is a **bad idea**, creating confusion.

**Note:** C++ does provide a `<complex>` class — But the above illustrates a suitable use for maths type operator overload
Adding two complex numbers together: A Complex class

- In order to allow more naturally expressed operations like:
  
  Complex c = a+b,

- we **overload** the ”+” operator.

```cpp
class Complex {
public:
  Complex(double re, double im)
    : real(re), imag(im) {}
  Complex operator+(const Complex& other);
  Complex operator=(const Complex& other);
private:
  double real;
  double imag;
};
Complex Complex::operator+(const Complex& other) {
  double result_real = real + other.real;
  double result_imaginary = imag + other.imag;
  return Complex(result_real, result_imaginary);
}
```

David Marshall
C++: Interfaces, Templates, Operator Overload & Exceptions
Exceptions

Exceptions are thrown to report exceptional circumstances

- Usually to report an error
  - You can `throw` any type of object, fundamental type or pointer
- It is good practice to throw objects which are sub-classes of `exception` class from the standard library
  - You add handler code to `catch` the exception
- The stack is unwound, one function at a time, until a `catch` which matches the type is found
- The `try` block: For a `throw` to `leave` a function, set up a special block within the function where you `try` to solve your actual programming problem:

```cpp
try {
    // Code that may generate exceptions
}
```
Simple Exception Example in C++

```cpp
try {
    foo ();
}
catch (int &i) {
    std::cout << "int was thrown by foo ()" << std::endl;
}
catch ( ... ) {
    std::cout << "Any other exception was thrown"
             << std::endl;
}

void foo() { throw int(42) }
```
The catch Clause

- **catch** clause will match an exception of the specified type
- **catch** clauses are checked in the order in which they are encountered
  - Order of catch clauses matters!
- **Pointers** and **objects** are different
- **Exceptions** are thrown by **value**
  - catch by **reference** or by **value** would work
  - catch by **reference** avoids the copy
- **char**, **int**, etc. are different
- **catch(...)** will **match any exception**
- Sub-class objects are base class objects
  - **catch** will match sub-class objects
- Use **throw** without arguments in catch clause to **rethrow exception**
**Standard Exceptions**

**exception** The base class for all the exceptions thrown by the C++ standard library. You can ask `what()` and get a result that can be displayed as a character representation.

**logic_error** Derived from exception:

**runtime_error** Derived from exception:

**iostream** Exception class `ios::failure` is also derived from exception, but it has no further subclasses.
Exception classes derived from `logic_error`

- **domain_error**: Reports violations of a precondition.
- **invalid_argument**: Indicates an invalid argument to the function it's thrown from.
- **length_error**: Indicates an attempt to produce an object whose length is greater than or equal to `NPOS` (the largest representable value of `type size_t`).
- **out_of_range**: Reports an out-of-range argument.
- **bad_cast**: Thrown for executing an invalid `dynamic_cast` expression in run-time type identification.
- **bad_typeid**: Reports a **null** pointer `p` in an expression `typeid(*p)`. 
Exception classes derived from runtime_error

- **range_error**: Reports violation of a postcondition.
- **overflow_error**: Reports an arithmetic overflow.
- **bad_alloc**: Reports a failure to allocate memory.

Header Files

- `<stdexcept>` — runtime, logic, overflow error definitions.
- `<new>` — also defines overflow_error

Catching a general exception

```cpp
catch (exception &e) {
    cerr << e.what();
}
```
Memory Allocation Example: Running Out of Memory Exception

What happens when the operator \texttt{new} cannot find a contiguous block of storage large enough to hold the desired object?

- It \textbf{throws} a \texttt{bad_alloc} exception

```cpp
#include <iostream>    // \texttt{cerr}
#include <stdexcept>   // \texttt{bad_alloc}
using namespace std;

int main () {
    try
    {
        int* myarray= new int[10000000000000000];
    }
    catch (bad_alloc& ba)
    {
        cerr << "bad_alloc caught: " << ba.what() << '\n';
    }
    return 0;
}
```

After this lecture & the following lab, you should:

▶ Understand the purpose of Interfaces, Abstract & Templates
▶ Understand the purpose of Friends in defining Classes.
▶ Write simple classes that use Interfaces & Abstract Classes
▶ Write simple classes & functions that use Templates
▶ Be able to overload functions in subclasses
▶ Understand C++ exceptions and catch exceptions in C++ code.