CE221
Programming in C++
Part 9
Type Conversions,
Virtual Functions and Abstract Classes
Type Conversion 1

We have seen that it is possible to convert between types by creating new objects using type conversion constructors. However we often wish to treat a pointer or a reference to an object of one type as a pointer or reference to an object of a subtype.

In Java we might use a loop of the form

```java
for (Person p:myList)
    if (p instanceof Student)
        System.out.println(((Student)p).regNo());
```

to print the registration numbers of all students in a list of persons; we use down-casting to treat a reference to a person as a reference to a student.
Type Conversion 2

In C++ we can use a similar syntax; this is known as \textit{C-style casting}. If we know that all of the members of \texttt{myList} are students (where \texttt{myList} is of type \texttt{List<Person*>}) we could print their registration numbers using

\begin{verbatim}
for (List<Person*>::Iterator it = myList.begin();
     it != myList.end(); it++)
{
    Person *p = *it;
    cout << ((Student*)p->regNo()) << endl;
}
\end{verbatim}

Note that this type of casting only works with pointers and references; also we had to use a list of pointers since a list of objects of type \texttt{Person} cannot hold objects of derived classes.
When C-style casting is used no type-checking whatsoever is performed either by the compiler or at runtime, so if any of the objects in the list in the code on the previous slide was not a student the behaviour of the program would be unpredictable and almost certainly incorrect.

Because of the total lack of type-checking the use of C-style casting in C++ is discouraged; C++ instead provides type-casting operators that do perform some type-checking; these include `static_cast` and `dynamic_cast`. 
The `static_cast` operator can be used to convert between types in situations where the compiler regards it as reasonable to do so. Permitted static casts include down-casting from pointers or references to base class objects to pointers or references to derived class objects, and also conversions between numeric types.

The syntax for a static cast is `static_cast<T>(e)`, where `T` is a type and `e` is an expression.

We could replace the output line of the code fragment on slide 3 with

```cpp
    cout << static_cast<Student*>(p)->regNo() << endl;
```
When `static_cast` is used no runtime checking is performed so, as before, if the pointer `p` on the previous slide did not point to a student the behaviour of the program would be unpredictable and almost certainly incorrect. The compiler will however check that the type conversion is reasonable; an attempt to cast a pointer to a person to a pointer to a date would be rejected.

As in Java when both operands of `/` are of integer type the result of the operator is of integer type. To obtain a real-number average from an integer total and an integer count we could use

```
double average = static_cast<double>(sum)/count;
```

The compiler will generate code to convert between integer and real-number representations.
The `dynamic_cast` operator can be used to treat a pointer or reference to a base class object as a pointer or reference to a derived class object, with run-time type checking being performed.

In the case of pointer conversion the result of the operation will be a null pointer if the cast is invalid; in the case of reference conversion an exception of type `bad_cast` would be thrown.

C++ has no `instanceof` operator, but we can check if a person is a student by examining the outcome of an attempted dynamic cast. Two versions of C++ code to perform the same task as the Java code on slide 2 are presented on the next slide; one uses pointers and the other uses references.
# dynamic_cast

// first version
for (List<Person*>::Iterator it = myList.begin();
    it != myList.end(); it++)
{ Student* s = dynamic_cast<Student*>(*it);
  if (s != 0)
    cout << s->regNo() << endl;
}

// second version
for (List<Person*>::Iterator it = myList.begin();
    it != myList.end(); it++)
{ try
  { cout << dynamic_cast<Student&>(*(*it)).regNo()
    << endl;
  }
  catch (bad_cast b) {}
Virtual Functions 1

Consider a class for the storing information about shapes that are to be displayed on some graphical display. Each shape will have some attributes such as size, screen position and colour. To allow all of the shapes that are to be displayed to be processed uniformly we need to store them in a list or set of objects of the same type; hence we shall need a Shape class. However, some of the properties of shapes are dependent on the individual shapes: the area of a square is the square of its sides, but the area of a circle is $\pi r^2$. Consequently we will need a subclass of the Shape class for each type of shape.
Virtual Functions 2

Here is an outline of a **Shape** class.

```cpp
class Shape
{
    public:
        Shape(int size, int x, int y);
        void changeSize(int newSize);
        void move(int newX, int newY);
        int getSize(), getX(), getY();
    
    protected:
        int size, xpos, ypos;
};
```

We assume that the shapes being used are squares, circles, equilateral triangles and other regular polygons, so that we do not have to consider the size in terms of width and length.
Virtual Functions 3

A `Circle` class can be written as a derived class of `Shape`:
```cpp
class Circle: public Shape
{
    public:
        Circle(int diam, int x, int y):
            Shape(diam, x, y) {}
        float area() const
        {
            int radius = size/2;
            return M_PI * radius * radius;
        }
};
```

Note that the header file `<cmath>` needs to be included in order to use `M_PI`. 
Virtual Functions 4

We can write similar classes \texttt{Square} and \texttt{Triangle}; although the areas of the squares will be integers, all of the \texttt{area} methods should return results of type \texttt{float} so that all shape classes have similar functionality.

To calculate the total area of all of the shapes in a collection \texttt{c} of pointers to objects of type \texttt{Shape} (assuming that all of the objects belong to derived classes that have \texttt{area} methods) we would wish to be able to write a loop of the form

```cpp
float totalArea = 0.0;
for (it = c.begin(); it != c.end(); it++)
    totalArea += (*it)->area();
```
Virtual Functions 5

The code on the previous slide will not compile since the `Shape` class does not have an `area` method so the compiler will not accept `(*it)->area()`. We could write a version that tries out several different dynamic casts but this would be cumbersome and we would have to know the names of all of the subclasses so the code would have to be modified if new subclasses were created.

In Java we can overcome the problem by giving the `Shape` class an `area` method (that could simply return 0.0); due to dynamic binding the appropriate subclass method would get invoked method for each object in the collection. This would not work in C++ since static binding results in the method from the `Shape` class being invoked for each object.
Virtual Functions 6

To get dynamic binding in C++ we have to declare a member function in a base class to be a *virtual function*. This is done by preceding its name with the keyword `virtual`. Hence we should add to the public part of the `Shape` class the function definition:

```cpp
class Shape {
public:
    virtual float area() const { return 0.0; }
};
```

The function in the derived class should not be declared as virtual unless we expect to further extend this class with other subclasses that will need different versions of the function, so we should not change the declaration of the `area` function in the `Circle` class.
Virtual Functions 7

When a function declared in a base class as virtual is applied to an object of a derived class accessed using a pointer or reference to the base class, the derived class version of that function overrides the inherited version and will be invoked. (If the programmer has not provided an overriding version the base class version will be used, as in Java.)

Note that the dynamic behaviour of virtual functions is only achieved when pointers or references are used since, for example, a variable of type `Shape` cannot hold a `Circle` object. Also note that if a derived class has a function with the same name as a virtual function of the base class, but has different argument types, the derived-class version will not override the virtual function.
Consider the following code.

```cpp
Circle c(12, 4, 8);
Shape s1 = c;
Shape &s2 = c;
Shape *p = &c;
cout << s1.area() << endl;
cout << s2.area() << ',' << p->area() << endl;
```

In the first assignment only the inherited part of `c` is copied into `s1` so `s1` is not a circle, and the base class `area` function will be invoked in the first output statement.

The variable `s2` refers to a circle and the pointer `p` points to a circle, so since `area` is a virtual function the derived class version will be invoked twice in the second output statement.
Abstract Classes 1

An *abstract class* is one that is used purely as a base class; no instances of it are allowed that are not instances of derived classes.

In Java a class is explicitly made abstract by using the keyword `abstract`. In C++ a different technique is used: a class is abstract if it has a *pure virtual function*. This is a function that has no implementation in the base class and is declared using the syntax

```cpp
virtual float area() const = 0;
```

As in Java all concrete subclasses of an abstract class must provide versions of the function to override the pure virtual version.
Here is an abstract version of the Shape class.

```cpp
class Shape
{
public:
    Shape(int size, int x, int y);
    void changeSize(int newSize);
    void move(int newX, int newY);
    int getSize(), getX(), getY();
    virtual float area() const = 0;

protected:
    int size, xpos, ypos;
};
```
Abstract Classes 3

Since no instances of an abstract class that are not instances of derived classes can be created it is not possible to have variables whose type is the abstract class; we must use references and/or pointers, so a declaration such as `Shape s;` would not be allowed. The following would, however, be permissible:

Shape &s = Circle(6, 10, 10)
Shape *p = new Square(5, 20, 20);

A declaration such as `Shape s[10];` is also not allowed and the type of objects in an STL collection cannot be an abstract class.
Abstract Classes 4

Pointers to abstract classes may be used as template arguments for the STL containers so it is possible to declare collections of **Shape** objects via pointers:

```cpp
Vector<Shape*> v;
   v.push_back(new Circle(10, 20, 20));
   v.push_back(new Triangle(5, 30, 30));
-------
for (int i = 0; i < v.size(); i++)
   cout << v[i]->area() << endl;
```
Abstract Classes and Output 1

Suppose we want to use `cout << x`, where `x` is a reference variable of type `Shape&`. Since `x` must refer to an object of a class derived from `Shape`, we would probably want the output to depend on the class of this object. However, since `operator<<` cannot be written as a member function of `Shape` we cannot make it virtual. Instead we need to write an `operator<<` function that calls a virtual function to perform the actual output:

```cpp
ostream &operator<<(ostream &o, const Shape &s) {
    s.put(o);
    return o;
}
```
Abstract Classes and Output 2

In the class **Shape** the function **put** would be declared as a pure virtual function:

```cpp
virtual void put(ostream&) const = 0;
```

Each subclass would contain a version of **put** that outputs the contents of that class to the stream. For example in the **Square** class we might use something like

```cpp
void put(ostream &o) const
{
    o << "Square of size " << size << " at (" << x << "," << y << ");
}
```