CE221

Programming in C++

Part 3

Control Structures, Operators, Classes and Objects
Control Structures 1

The control structures in C++ are essentially the same as in Java.

Selection structures: `if` (with optional `else`) and `switch`

Repetition structures: `while`, `do` and `for`

Exception handling: `try/catch`

However C++ does not have anything analogous to Java's enhanced for loop (e.g. `for (int x:a)`) and the declaration of user-defined exception classes differs from Java. (Exception-handling in C++ will be covered later.)
Control Structures 2

The `break` and `continue` statements can be used to break out of control structures. Their use tends to make the control structure difficult to follow so it should be avoided where possible, although the use of `break` is of course essential in most `switch` statements.

The `break` statement causes the program to jump out of the nearest enclosing loop or `switch` statement and start executing the code that immediately follows it; the `continue` statement (which is used only in loops) causes the program to jump to the end of the current iteration and proceed by checking the condition for continuing in the loop, having in the case of a `for` loop first evaluated the "next-step" expression.
break and continue – an Example

while (testfunction(x, y))
{
    statement1;
    if (x<=-1) break;
    statement2;
    if (y<=-1) continue;
    statement3;
}
Default Arguments To Functions

A function can have arguments with default values, e.g.

```c
int volume(int length = 1, int width = 1,
           int height = 1) { ..... }
```

This function can be called with 0, 1, 2 or 3 arguments, the default values being used for all arguments that are not supplied, e.g. `volume(3,4)` would be treated as `volume(3,4,1).

If some but not all of the parameters of a function have default values they must appear at the end of the parameter list, e.g.

```c
void printArea(float length, float width,
               int decimalPlaces = 2) { ..... }
```
Function Overloading

It is permissible to write more than one function with the same name as long as they have different numbers of arguments or different argument types. The compiler will determine which version to use for each call by looking at the arguments.

The return types of the overloaded functions may be the same or may be different, e.g.

```c
int square(int x);
float square(float x);
float area(float height, float width);
float area(float radius);
```

If overloaded functions have default arguments the types of the non-default arguments must differ.
More about Functions

Functions in C++ may not be nested, i.e. it is not possible to declare a function inside the body of another function.

If a function is a member of a class it must be applied to an object of that class, unless it is defined as `static`.

Short simple functions are sometimes declared as `inline`, e.g.

```cpp
inline int cube(int i)
{
    return i*i*i;
}
```

This causes a copy of the code for the body to be inserted into the code generated by the compiler instead of a function call, resulting in improved performance at the expense of larger code files.
Operators and Precedence

C++ has many operators. It has all of the operators of Java apart from `instanceof`. In addition there are the operators `->`, `::` and `delete` and the unary versions of `*` and `&` that we have already encountered, and there are `.`, `-`, `*`, `-=` , several cast operators and `sizeof`. The operator precedence table is shown on the following slides, with the highest precedence first, so for example `*p.q` means `*(p.q)` . The associativity is used to determine the meaning of an expression if adjacent operators have the same precedence so `a-b+c` means `(a-b)+c` since the operators associate from left to right, but `x = y += z` means `x = (y += z)` since the operators associate from right to left.
### Operator Precedence Table 1

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>:: (binary scope resolution)</td>
<td>left to right</td>
</tr>
<tr>
<td>:: (unary scope resolution)</td>
<td>left to right</td>
</tr>
<tr>
<td>() (function call)</td>
<td>left to right</td>
</tr>
<tr>
<td>[] (array subscript)</td>
<td>left to right</td>
</tr>
<tr>
<td>. (member selection via object)</td>
<td></td>
</tr>
<tr>
<td>-&gt; (member selection via pointer)</td>
<td></td>
</tr>
<tr>
<td>++ (unary postfix increment)</td>
<td></td>
</tr>
<tr>
<td>-- (unary postfix decrement)</td>
<td></td>
</tr>
<tr>
<td><code>typeid</code> (runtime type information)</td>
<td></td>
</tr>
<tr>
<td><code>dynamic_cast&lt;type&gt;</code> (runtime type-checked cast)</td>
<td></td>
</tr>
<tr>
<td><code>static_cast&lt;type&gt;</code> (compile-time type-checked cast)</td>
<td></td>
</tr>
<tr>
<td><code>reinterpret_cast&lt;type&gt;</code> (cast for nonstandard conversions)</td>
<td></td>
</tr>
<tr>
<td><code>const_cast&lt;type&gt;</code> (cast away const-ness)</td>
<td></td>
</tr>
</tbody>
</table>
## Operator Precedence Table 2

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>++</code> (unary prefix increment)</td>
<td>right to left</td>
</tr>
<tr>
<td><code>--</code> (unary prefix decrement)</td>
<td></td>
</tr>
<tr>
<td><code>+</code> (unary plus)</td>
<td></td>
</tr>
<tr>
<td><code>-</code> (unary minus)</td>
<td></td>
</tr>
<tr>
<td><code>!</code> (unary logical negation)</td>
<td></td>
</tr>
<tr>
<td><code>~</code> (unary bitwise complement)</td>
<td></td>
</tr>
<tr>
<td><code>sizeof</code> (determine size in bytes)</td>
<td></td>
</tr>
<tr>
<td><code>&amp;</code> (address)</td>
<td></td>
</tr>
<tr>
<td><code>*</code> (dereference)</td>
<td></td>
</tr>
<tr>
<td><code>new</code> (dynamic memory allocation)</td>
<td></td>
</tr>
<tr>
<td><code>new[]</code> (dynamic array allocation)</td>
<td></td>
</tr>
<tr>
<td><code>delete</code> (dynamic memory deallocation)</td>
<td></td>
</tr>
<tr>
<td><code>delete[]</code> (dynamic array deallocation)</td>
<td></td>
</tr>
<tr>
<td><code>(type)</code> (C-style unary cast)</td>
<td>right to left</td>
</tr>
<tr>
<td><code>.=</code> (pointer to member via object)</td>
<td>left to right</td>
</tr>
<tr>
<td><code>-&gt;*</code> (pointer to member via pointer)</td>
<td></td>
</tr>
</tbody>
</table>
## Operator Precedence Table 3

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>* (multiplication)</td>
<td>left to right</td>
</tr>
<tr>
<td>/ (division)</td>
<td>left to right</td>
</tr>
<tr>
<td>% (modulus)</td>
<td>left to right</td>
</tr>
<tr>
<td>+ (addition)</td>
<td>left to right</td>
</tr>
<tr>
<td>- (subtraction)</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt; (bitwise left shift)</td>
<td>left to right</td>
</tr>
<tr>
<td>&gt;&gt; (bitwise right shift)</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt; (less than)</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;= (less than or equal to)</td>
<td>left to right</td>
</tr>
<tr>
<td>&gt; (greater than)</td>
<td>left to right</td>
</tr>
<tr>
<td>&gt;= (greater than or equal to)</td>
<td>left to right</td>
</tr>
<tr>
<td>== (is equal to)</td>
<td>left to right</td>
</tr>
<tr>
<td>!= (is not equal to)</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp; (bitwise AND)</td>
<td>left to right</td>
</tr>
<tr>
<td>^ (bitwise exclusive OR)</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td>(bitwise inclusive OR)</td>
</tr>
</tbody>
</table>
## Operator Precedence Table 4

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;&amp;</code> (logical AND)</td>
<td>left to right</td>
</tr>
<tr>
<td>`</td>
<td></td>
</tr>
<tr>
<td><code>? :</code> (ternary conditional)</td>
<td>right to left</td>
</tr>
<tr>
<td><code>=</code> (assignment)</td>
<td>right to left</td>
</tr>
<tr>
<td><code>+=</code> (addition assignment)</td>
<td></td>
</tr>
<tr>
<td><code>-=</code> (subtraction assignment)</td>
<td></td>
</tr>
<tr>
<td><code>*=</code> (multiplication assignment)</td>
<td></td>
</tr>
<tr>
<td><code>/=</code> (division assignment)</td>
<td></td>
</tr>
<tr>
<td><code>%=</code> (modulus assignment)</td>
<td></td>
</tr>
<tr>
<td><code>&amp;=</code> (bitwise AND assignment)</td>
<td></td>
</tr>
<tr>
<td><code>^=</code> (bitwise exclusive OR assignment)</td>
<td></td>
</tr>
<tr>
<td>`</td>
<td>=` (bitwise inclusive OR assignment)</td>
</tr>
<tr>
<td><code>&lt;&lt;=</code> (bitwise left-shift assignment)</td>
<td></td>
</tr>
<tr>
<td><code>&gt;&gt;=</code> (bitwise right-shift assignment)</td>
<td></td>
</tr>
<tr>
<td><code>,</code> (sequencing)</td>
<td>left to right</td>
</tr>
</tbody>
</table>
Order of Evaluation 1

It is important to note that the operator precedence table does not always specify the precise order of evaluation. It tells us that `i++*j--` means `(i++)*(j--)` so the increment and decrement must be performed before the multiplication but we do not know which of the increment and decrement will be performed first. In this example it does not matter – the outcome will be the same.

In some cases the order of evaluation is specified – the operands of the logical operators, the ternary conditional and the comma operator are always evaluated from left to right and in the case of the logical operators and conditional some of the operands may not be evaluated.
Order of Evaluation 2

In an expression of the form `e1 ? e2 : e3` only one of `e2` and `e3` will be evaluated; if `e1` evaluates to true (or a non-zero value) the value of the expression is the value of `e2`, otherwise it is the value of `e3`.

The logical operators are evaluated lazily: `e1 && e2` must be false if `e1` is false, so `e2` will not be evaluated, and similarly `e1 || e2` must be true if `e1` is true, so again `e2` will not be evaluated.
Both `i++` and `++i` may be used to increment the value held in the variable `i`. However the values of the two expressions are different: the value of `i++` is the value that was stored in `i` before the increment, whereas the value of `++i` is the value that is stored in `i` after the increment. This difference is significant only if the increment is used as a sub-expression: `a[i++] = 5;` and `a[++i] = 5;` will update different elements of the array. The same applies to the `--` operator.

Note that expressions such as `++i++` and `++i--` are not allowed; the operand of these operators must be a reference not a value, and the precedence rules tell us that `++i++` would mean `++(i++)` and hence the operand of the first `++` would be a value.
Classes and Objects – an Example 1

// Time.h - declaration of class Time.
// member functions are defined in Time.cpp

#ifndef _TIME_H
#define _TIME_H

class Time
{
    public:
        Time(); // constructor
        void setTime(int, int, int); // set hour, minute and second
        void printUniversal(); // print time in universal-time format
        void printStandard(); // print time in standard-time format

    private:
        int hour; // 0 - 23 (24-hour clock format)
        int min;  // 0 - 59
        int sec;  // 0 - 59
}; // end of class Time

#endif
//Time.cpp - member-function definitions for class Time.
#include <iostream>
#include <iomanip>
#include "Time.h"
using namespace std;

// constructor initializes each data member to zero.
// ensures all Time objects start in a consistent state.
Time::Time()
{ hour = min = sec = 0;
}

// set new Time value using universal time;
// ensure that the data remains consistent by setting
// invalid values to zero
void Time::setTime(int h, int m, int s)
{ hour = (h>=0 && h<24) ? h : 0; // validate hour
  min = (m>=0 && m<60) ? m : 0; // validate minute
  sec = (s>=0 && s<60) ? s : 0; // validate second
}
/ Time.cpp (continued)

// print time in universal-time format (HH:MM:SS)
void Time::printUniversal()
{
cout << setfill('0') << setw(2) << hour << ":" << setw(2) << min << ":" << setw(2) << sec;
}

// print time in standard-time format
// ([H]H:MM:SS am/pm)
void Time::printStandard()
{
cout << (hour==0 || hour==12 ? 12 : hour%12) << ":" << setfill('0') << setw(2) << min << ":" << setw(2) << sec << (hour<12 ? " am" : " pm");
}

/*
setw sets the field width of the next item to be output
setfill sets the fill character to be used for all subsequent outputs - we need leading zeroes*/
// program to test class Time
#include <iostream>
#include "Time.h"

using namespace std;

int main()
{ Time t; // instantiate object t of class Time

    // output t's initial value in both formats
    cout << "The initial universal time is ";
    t.printUniversal(); // 00:00:00
    cout << "\nThe initial standard time is ";
    t.printStandard(); // 12:00:00 am

    t.setTime(13, 27, 6); // change time

    // output t's new value in both formats
    cout << "\n\nUniversal time after setTime is ";
    t.printUniversal(); // 13:27:06
    cout << "\nStandard time after setTime is ";
    t.printStandard(); // 1:27:06 pm
}
Notes on the Example 1

As in Java there are three visibility specifiers: public, private and protected. Private members can be accessed only in functions of the class and its friend functions (these will be explained in part 4). Protected members can also be accessed in functions and friend functions of subclasses of the class. Public members can be accessed anywhere in the program (as long as the class is in scope).

Unlike Java we label groups of members as being public, private and protected; we may have more than one group of each type. If any member is declared before the occurrence of any of the keywords it will be treated as private.
Notes on the Example 2

As in Java a constructor is used to initialise the members of an object. This must be a member function with the same name as the class and no return type. It should normally be public. If no constructor is provided the compiler will generate a default constructor with no arguments – this will call the no-argument constructors of any members of the class that are themselves class objects but will not initialise any objects of primitive type or any pointers or arrays.
Notes on the Example 3

A class may have more than one constructor as long as each has different parameter types. Constructors may have default arguments. We could have provided a second constructor for our class by adding to the class declaration

```cpp
Time(int h, int m = 0, int s = 0);
```

and adding to the `.cpp` file

```cpp
Time::Time(int h, int m, int s)
{
    setTime(h, m, s);
}
```

Note that the default argument values should be specified only once; this should be in the class declaration in the header file since the caller needs to know that there are default values.
Invoking Constructors

When a class object is declared with no initialisation (e.g. `Time t;`) the no-argument constructor for the class will be called implicitly to initialise it. If we wish to use a different constructor we must supply the arguments in parentheses after the variable name, e.g. `Time t2(12,15);`.

To create objects on the heap we would use statements such as:

```
Time *tPtr1 = new Time;
Time *tPtr2 = new Time(15,30,40);
```

Note that parentheses are not used if no arguments are to be supplied to the constructor.
Accessing Class Members

We have already seen that outside of the member functions of a class its members are accessed using the . operator or the \rightarrow operator. The left-hand operand of the . operator must be a class object or a reference to an object, whereas the left-hand operand of \rightarrow must be a pointer to a class object, e.g.

```cpp
Time t;
Time tArray[5];
Time &tRef = t;
Time *tPtr = &t;

t.setTime(13, 27, 6);
tArray[2].setTime(13, 27, 11);
tRef.printStandard();
tPtr->setTime(13, 27, 19);
(tArray+2)->printUniversal();
```
Constant Objects and Member Functions

It is possible to declare constant class objects. e.g.

```cpp
const Time noon(12); // or (12, 0, 0)
```

The only functions that can be applied to such objects are \textit{constant member functions}. Hence a statement such as `noon.printStandard();` would be rejected by the compiler when using the current version of our class. To allow such a statement to be able to be used we should have defined `printStandard` as a constant member function by adding the `const` keyword at the end of the declaration:

```cpp
void printStandard() const;
```

The compiler would then check that the function does not change the contents of the object to which it is applied.
Initialising Members of Objects 1

It is not possible to initialise a member of a class in its declaration, unless it is a constant static member. Hence

```cpp
class X
{ int y = 3;
    ....
};
```

would not be allowed.

All initialisation of non-static class members must be performed in the constructor(s) of the class. There is an alternative to the initialisation by assignment seen in the constructors for our `Time` class; we can initialise members using an `initialiser list`:

```cpp
Time::Time() : hour(0), sec(0), min(0) { }
```
Initialising Members of Objects 2

If one of the members of a class is an object of another class we must invoke one of that class's constructors to initialise it. Consider the following incomplete class.

```cpp
class Lecture
{
private:
    Time startTime;

public:
    Lecture(......, int startHour);
};
```

In the constructor we need to initialise the `startTime` member. We cannot simply write in the body

```cpp
startTime(startHour);
```

since the compiler would try to treat this as a function call.
One option for the initialisation of the startTime member would be to allow the no-argument constructor to be invoked implicitly and then use the setTime function:

```java
startTime.setTime(startHour, 0, 0);
```

This is not an ideal approach; it is somewhat inefficient and relies on the fact that the Time class has a no-argument constructor and a setTime function; in other examples this might not be the case.

Instead we should perform the initialisation as part of an initialiser list:

```cpp
Lecture::Lecture(......, startHour) :
    startTime(startHour)
{
    ...... }
```
Initialising Members of Objects 4

If a class has as a member an object of another class and has a constructor in which that object is not initialised in an initialiser list, the no-argument constructor for the object is invoked implicitly. Hence if we want the initial value of the object to be the default initial value for objects of that class there is no need to perform any explicit initialisation.

If the compiler generates a default no-argument constructor for a class, this will initialise any object members using their no-argument constructors, so if any object member belongs to a class that has no no-argument constructor we cannot use a default constructor and must provide one that initialises the object in its initialiser list.
A class may have static members; these belong to the class rather than to individual objects of the class and exist even if no objects of the class exist. Hence initialisation of such members cannot be performed in a constructor. Furthermore we are not allowed to initialise a non-constant member in the class declaration. A non-constant static class member must hence be initialised outside of any declaration.

Consider a class to store information about employees. Assume that each employee needs to be given a unique payroll number, with the first being given the number 1, the second 2 and so on. Hence we need to keep a count of how many Employee objects have been created. We could use a global variable but it is better to use a static member.
Static Members of Classes 2

The following gives an outline of how we would initialise and use the static member:

```cpp
// Employee.h (incomplete)
class Employee:
{ private:
    int payRollNum;
    static int count:

public:
    Employee(......);

......

// Employee.cpp (incomplete)
Employee::count = 0;
Employee::Employee(......)
{ payRollNum = ++count;
...... }
```
Using this

If within a member function we need to supply the object to which the function is being applied as an argument to another function we use the keyword *this*, as in Java.

However, in C++, *this* is a pointer to the object, not a reference. If the function being called expects an object or a reference to an object as its argument we have to use *this*. 