Short Course on Programming in C/C++

Organized by Onur Pekcan

Contributor Selim Temizer Instructor Hasan Yılmaz

Week 3 – Lecture1

Today

We will cover;

- **Advanced Concepts In C++**
  - Templates
  - Namespaces
  - Exceptions
  - Type Casting
- **C++ Standar Library**
  - Input / Output with Files in C++

Templates

- The template allows us to write routines that work for arbitrary types without having to know what these types will be.
  - Similar to `typedef` but more powerful
- Two types:
  - Function templates
  - Class templates

Function Templates

- A function template is not an actual function; instead it is a design (or pattern) for a function.
- This design is expanded (like a preprocessor macro) as needed to provide an actual routine.

```
// swap function template.
// Object: must have copy constructor and operator =

template < class Object>
void swap( Object &lhs, Object &rhs )
{
        Object tmp = lhs;
        lhs = rhs;
        rhs = tmp;
}
```

The `swap` function template

Using a template

- Instantiation of a template with a particular type, logically creates a new function.
- Only one instantiation is created for each parameter-type combination.

```
int main()
{       int x = 5, y = 7;
        double a = 2, b = 4;
        swap(x,y);
        swap(x,y); //uses the same instantiation
        swap(a,b);
        cout << x << " " << y << endl;
        cout << a << " " << b << endl;
//      swap(x, b); // Illegal: no match
        return 0;
}
```

Class templates

- Class templates are used to define more complicated data abstractions.
  - e.g. it may be possible to use a class that defines several operations on a collection of integers to manipulate a collection of real numbers.

```
// Form of a template interface
template <class T>
class class-name
{
        public:
        // list of public members
        ...
        private:
        // private members
        ...
};
```

Interpretation:

Class *class-name* is a template class with parameter T. T is a placeholder for a built-in or user-defined data type.

Implementation

- Each member function must be declared as a template.

```
// Typical member implementation.
template <class T>
ReturnType
class-name<T>::MemberName( Parameter List ) /*
   const*/
{
   // Member body
}
```

Object declarations using template classes

Form:

class-name <type> an-object;

Interpretation:

– *Type* may be any defined data type. *Class-name* is the name of a template class. The object *an-object* is created when the arguments specified between < > replace their corresponding parameters in the template class.

Example 1

```
// Memory cell interface

template <class Object>
class MemoryCell
{
   public:
      MemoryCell( const Object & initVal = Object() );
      const Object & read( ) const;
      void write( const Object & x);

   private:
      Object storedValue;
};
```

Class template implementation

```
// Implementation of class members.
#include "MemoryCell.h"

template <class Object>
MemoryCell<Object>::MemoryCell(const Object & initVal) :
   storedValue( initVal){ }

template <class Object>
const Object & MemoryCell<Object> :: read() const
{
   return storedValue;
}
template <class Object>
void MemoryCell<Object>::write( const Object & x )
{
   storedValue = x;
}
```

A simple test routine

```
int main()
{
  MemoryCell<int> m;

  m. write(5);
  cout << "Cell content: " << m.read() <<
  endl;
  return 0;
}
```

Namespaces

- Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name.

  The format of namespaces is:

```
namespace identifier
{
entities
}
```

Namespaces

- Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace. For example:

```
namespace myNamespace{
  int a, b;
}
```

- In order to access these variables from outside the myNamespace namespace we have to use the scope operator ::
- For example, to access the previous variables from outside myNamespace we can write

```
myNamespace::a
myNamespace::b
```

Example 2

- The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors. For example:

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

namespace first
{
  int var = 5;
}

namespace second
{
  double var = 3.1416;
}

int main () {
  cout << first::var << endl;
  cout << second::var << endl;
  return 0;
}
```

Exceptions

C++ Exception Handling: `try`, `throw`, `catch`

- A function can **`throw`** an exception object if it detects an error
  - Object typically a character string (error message) or class object
  - If exception handler exists, exception caught and handled
  - Otherwise, program terminates

Exceptions

- Format
  - enclose code that may have an error in **`try`** block
  - follow with one or more **`catch`** blocks
    - each **`catch`** block has an exception handler
  - if exception occurs and matches parameter in **`catch`** block, code in catch block executed
  - if no exception thrown, exception handlers skipped and control resumes after catch blocks
  - **`throw`** point - place where exception occurred
    - control cannot return to **`throw`** point

```
1  // Fig. 23.1: fig23_01.cpp
2  // A simple exception handling example.
3  // Checking for a divide-by-zero exception.
4  #include <iostream>
5
6  using std::cout;
7  using std::cin;
8  using std::endl;
9
10 // Class DivideByZeroException to be used in exception
11 // handling for throwing an exception on a division by zero.
12 class DivideByZeroException {
13 public:
14    DivideByZeroException()
15       : message( "attempted to divide by zero" ) { }
16    const char *what() const { return message; }
17 private:
18    const char *message;
19 };
20
21 // Definition of function quotient. Demonstrates throwing
22 // an exception when a divide-by-zero exception is encountered.
23 double quotient( int numerator, int denominator )
24 {
25    if ( denominator == 0 )
26       throw DivideByZeroException();
27
28    return static_cast< double > ( numerator ) / denominator;
29 }
```

EXAMPLE

- Class definition
- Function definition

```
30
31 // Driver program
32 int main()
33 {
34    int number1, number2;
35    double result;
36
37    cout << "Enter two integers (end-of-file to end): ";
38
39    while ( cin >> number1 >> number2 ) {
40
41       // the try block wraps the code that may throw an
42       // exception and the code that should not execute
43       // if an exception occurs
44       try {
45          result = quotient( number1, number2 );
46          cout << "The quotient is: " << result << endl;
47       }
48       catch ( DivideByZeroException ex ) { // exception handler
49          cout << "Exception occurred: " << ex.what() << '\n';
50       }
51
52       cout << "\nEnter two integers (end-of-file to end): ";
53    }
54
55    cout << endl;
56    return 0;      // terminate normally
57 }
```

- Initialize variables
- Input data
- `try` and `catch` blocks
- Function call
- Output result

Example of a `try-catch` Statement

```
try
{
        // Statements that process personnel data and may throw
        // exceptions of type int, string, and SalaryError
}
catch ( int )
{
        // Statements to handle an int exception
}
catch ( string s )
{
       cout << s << endl;  // Prints "Invalid customer age"
        // More statements to handle an age error
}
catch ( SalaryError )
{
       // Statements to handle a salary error
}
```

Type Casting

- Converting an expression of a given type into another type is known as *type-casting*.
- We have already seen some ways to type cast:

➢ **Implicit Conversion**

➢ **Explicit Conversion**

Type Casting

Implicit Conversion

- Implicit conversions do not require any operator.

- They are automatically performed when a value is copied to a compatible type. For example:

```
short a=2000;
int b;
b=a;
```

- Here, the value of a has been promoted from short to int and we have not had to specify any type-casting operator. This is known as a standard conversion

Type Casting

Implicit Conversion

- Implicit conversions also include constructor or operator conversions, which affect classes that include specific constructors or operator functions to perform conversions. For example:

```
class A {};
class B { public: B (A a) {} };
A a;
B b=a;
```

- Here, a implicit conversion happened between objects of class A and class B, because B has a constructor that takes an object of class A as parameter. Therefore implicit conversions from A to B are allowed.

Type Casting

Explicit Conversion

- C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. Two notations for explicit type conversion: **functional** and **c-like** casting:

```
short a=2000;
int b;
b = (int) a;      // c-like cast notation
b = int (a);      // functional notation
```

- The functionality of these explicit conversion operators is enough for most needs with fundamental data types. However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that while being syntactically correct can cause **runtime errors**.

Example 3

- This code is syntactically correct

- Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. The subsequent call to member result will produce either a run-time error or a unexpected result

```
// class type-casting
#include <iostream>
using namespace std;

class CDummy {
   float i,j;
};

class CAddition {
                int x,y;
  public:
                CAddition (int a, int b) { x=a; y=b; }
                int result() { return x+y;}
};

int main () {
  CDummy d;
  CAddition * padd;
  padd = (CAddition*) &d;
  cout << padd->result();
  return 0;
}
```

C++ Standar Library

Input / Output with Files in C++

I/O with Files

- C++ provides the following classes to perform output and input of characters to/from files
  - **ofstream:** Stream class to write on files
  - **ifstream:** Stream class to read from files
  - **fstream:** Stream class to both read and write from/to files.

Example 4

```
// basic file operations
#include <iostream>
#include <fstream>

using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

Output:

[file example.txt]
Writing this to a file.

This code creates a file called example.txt and inserts a sentence into it in the same way we are used to do with cout, but using the file stream myfile instead.

Opening and Closing a File

- **open (filename, mode);**

```
ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app | ios::binary);

if (myfile.is_open()) { /* ok, proceed with output */ }

myfile.close();
```

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

Example 5

```
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile ("example.txt");
  if (myfile.is_open())
  {
    myfile << "This is a line.\n";
    myfile << "This is another line.\n";
    myfile.close();
  }
  else cout << "Unable to open file";
  return 0;
}
```

Output:

[file example.txt]

This is a line.

This is another line.

Example 6

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile ("example.txt");
  if (myfile.is_open())
  {
    while ( myfile.good() )
    {
      getline (myfile,line);
      cout << line << endl;
    }
    myfile.close();
  }
  else cout << "Unable to open file";

  return 0;
}
```

Output:

This is a line.

This is another line.