C++

Session #7

#7

- Dynamic Memory
- Templates
- Exception Handling
- Files and Streams

Dynamic Memory

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts

- **The stack** – All variables declared inside the function will take up memory from the stack.
- **The heap** – This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Dynamic Memory

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

Dynamic Memory

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

Dynamic Memory

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory that was previously allocated by new operator.

Dynamic Memory

There is following generic syntax to use new operator to allocate memory dynamically for any data-type.

new data-type;

Here, data-type could be any built-in data type including an array or any user defined data types include class or structure.

Dynamic Memory

Example:

```
double* pvalue  = NULL; // Pointer initialized with null
pvalue  = new double;   // Request memory for the variable
```

Dynamic Memory

Same after having 10 beers within 2 hours, your brain will fail to save your memory and on a computer the memory may not have been allocated successfully as well

If the free store had been used up. So it is good practice to check if new operator is returning **NULL** pointer and take appropriate action as below

Dynamic Memory

```
double* pvalue  = NULL;
if( !(pvalue  = new double )) {
   cout << "Error: out of memory." <<endl;
   exit(1);
}
```

Dynamic Memory

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

int main () {
  double* pvalue  = NULL; // Pointer initialized with null
  pvalue  = new double;   // Request memory for the variable

  *pvalue = 29494.99;     // Store value at allocated address
  cout << "Value of pvalue : " << *pvalue << endl;

  delete pvalue;        // free up the memory.

  return 0;
}
```

Dynamic Memory - Arrays

Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below.

```
char* pvalue  = NULL;       // Pointer initialized with null
pvalue  = new char[20];     // Request memory for the variable
```

Dynamic Memory - Arrays

To remove the array that we have just created the statement would look like this

```
delete [] pvalue;          // Delete array pointed to by pvalue
```

Dynamic Memory - Arrays

Following the similar generic syntax of new operator, you can allocate for a multi-dimensional array as follows

```
double** pvalue  = NULL;     // Pointer initialized with null
pvalue  = new double [3][4];  // Allocate memory for a 3x4 array
```

Dynamic Memory - Arrays

However, the syntax to release the memory for multi-dimensional array will still remain same as before:

```
delete [] pvalue;        // Delete array pointed to by pvalue
```

Dynamic Memory - Objects

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

class Box {
   public:
      Box() {
         cout << "Constructor called!" <<endl;
      }
      ~Box() {
         cout << "Destructor called!" <<endl;
      }
};
int main() {
   Box* myBoxArray = new Box[4];
   delete [] myBoxArray; // Delete array

   return 0;
}
```

Dynamic Memory - Objects

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times.

Templates

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

Templates

There is a single definition of each container, such as vector, but we can define many different kinds of vectors for example, vector <int> or vector <string>.

Templates

Function Template:

```
template <class type> ret-type func-name(parameter list) {
  // body of function
}
```

Templates

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here

```
template <class type> class class-name {
  .
  .
  .
}
```

Templates

type is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.

Templates

Templates can be applied to:

- Functions
- Class
- Methods

Exception Handling

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exception Handling

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

- throw − A program throws an exception when a problem shows up. This is done using a throw keyword.
- catch − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- try − A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Exception Handling

Assuming a block will raise an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception.

Exception Handling

Exceptions can be thrown **anywhere** within a code block using throw statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Exception Handling

```
double division(int a, int b) {
  if( b == 0 ) {
    throw "Division by zero condition!";
  }
  return (a/b);
}
```

Exception Handling

The catch block following the try block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

Exception Handling

```
try {
   // protected code
} catch( ExceptionName e ) {
  // code to handle ExceptionName exception
}
```

Exception Handling

The code on the previous slide will catch an exception of ExceptionName type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows

Exception Handling

```
try {
  // protected code
} catch(...) {
  // code to handle any exception
}
```

Exception Handling

- **Std::exception**
  - An exception and parent class of all the standard C++ exceptions.
- **Std::bad_alloc**
  - This can be thrown by new.
- **Std::bad_cast**
  - This can be thrown by dynamic_cast.
- **Std::bad_exception**
  - This is useful device to handle unexpected exceptions in a C++ program.
- **Std::bad_typeid**
  - This can be thrown by typeid.
- **Std::logic_errorstd::out_of_range**
  - This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[]().

Exception Handling

You can define your own exceptions by inheriting and overriding exception class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way.

Exception Handling

```
struct MyException : public exception {
  const char * what () const throw () {
    return "C++ Exception";
  }
};
```