C++ Programming Basics

February 16, 2023

Email any questions to:
ritu@wayne.edu

Administrative Trivia

- Assignment-1, part-2 has been posted

- Assignment 2 will be posted later today

- Mid-term will be held in class on February 21, 2023

Dynamic Memory Allocation

- C++ enables programmers to control the allocation and deallocation of memory in a program for any built-in type or user-defined type

- This is dynamic memory management and is accomplished by the operators `new` and `delete`
  - Or use functions `malloc` and `free`

- **Note: When we use arrays, static memory allocation takes place**

Comparing **malloc/free &** **new/delete**

```
//Using malloc and free functions
int* ip;
ip = (int*)malloc(sizeof(int) * 100);
...
free((void*)ip);
```

```
//Using new and delete operators
int* ip;
ip = new int[100];
...
delete ip;
```

new & delete Example: newDelete.cc

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

class myclass {
public:
  myclass() {cout <<"myclass constructed\n";}
  ~myclass() {cout <<"myclass destroyed\n";}
};

int main () {
  myclass * pt;
  pt = new myclass[3];
  delete[] pt;
  return 0;
}
```

Output:

```
myclass constructed
myclass constructed
myclass constructed
myclass destroyed
myclass destroyed
myclass destroyed
```

In Class Exercise

- Using **new**/**delete**
  - Write a **main** function that asks the user to enter the number of students
  - Allocate memory dynamically for the number of students entered by the user
  - Prompt the user for the marks for each student and save the marks in the dynamically created memory
  - Print the marks entered for each student

```
Output:
Enter the num of students : 2

Enter the marks of student_1 21

Enter the marks of student_2 22
student_1 has 21 marks
student_2 has 22 marks
```

Solution to the Exercise: testNewDelete.cc

```
#include <iostream>
using namespace std;
int main(){
  int numStudents, *ptr, i;
  cout << "Enter the num of students : ";
  cin >> numStudents;
  ptr = new int[numStudents];
  for (i=0; i<numStudents; i++){
    cout << "\nEnter the marks of student_" << i +1 <<" ";
    cin >> ptr[i];
  }
  for (i=0; i<numStudents; i++){
   cout <<"student_"<< i+1 <<" has "<<ptr[i] << "marks\n";
  }
  delete [] ptr;
  return 0;
}
```

Friends

- As a rule, private and protected members of a class cannot be accessed from outside the class in which they are declared
  - However, a **`friend`** can break this rule!
  - Functions or classes declared with the keyword **`friend`** are friends
  - An external function can be declared as friend of a class by declaring a prototype of this external function within the class, and preceding it with the keyword **`friend`**
  - A class can also be defined as a **`friend`** of another class to grant it access to the protected and private members

Code Snippet from testFriend.cpp

```cpp
class Parent{
  int xNumber, yNumber;
  public:
    virtual void clean();
    friend void accessXY(Parent);
};

void Parent::clean(){
  cout << "\nIn Parent's clean method\n" ;
}
void accessXY(Parent pObj){
  pObj.xNumber = 100;
  pObj.yNumber = 200;
  cout<< "xNumber is: " << pObj.xNumber << endl;
  cout<< "yNumber is: " << pObj.yNumber << endl;
}
```

`inline` function

- `inline` function instructs the compiler to insert complete body of the function wherever that function got used/called in code
- It is an optimization technique and the compiler can ignore the request to `inline` a function
- Use `inline` keyword in front of the function-prototype to make a function inline

In Class Exercise

- Use the code snippet shown on slide 4
- Write the main function in which
  - Declare an object of class `Parent`
  - Call function `accessXY` and pass the object of class Parent to it
  - Call the `clean` method on the object of `Parent` class

Exception, Exception Handling

- Exceptions are infrequent problems that occur when the program is running – example, division by zero

- The mechanism of exception handling enables programmers to resolve exceptions - improves program’s fault-tolerance – in separate blocks of code

- Often the program continues to run normally when an exception occurs

- In some severe situations, when the program encounters exception and cannot continue further, the user is notified of the problem before the program terminates

C++ Syntax for Exception Handling

- `#include <exception>`
  - Also, `#include <stdexcept>`

- Three keywords to be used : `try`, `catch`, and `throw`

- Keyword `try` followed by braces is used to define a try block
  - Block of code in which exceptions might occur
  - Example: invocation of a function that can result in division by zero

- An exception is thrown by using `throw` keyword from the try block

- The thrown exception is handled by catch handlers defined using keyword `catch`

Termination Model of Exception Handling

- Some Notes
  - The try block is immediately followed by one or more catch blocks
  - Each catch handlers can only take one parameter
  - Each catch block should handle unique type of error

- When an exception occurs in the try block, the try block terminates immediately and the program control goes to the catch block that can handle the type of exception raised
  - The appropriate catch block is found by matching the thrown exception’s type with the catch’s parameter type
  - When statements in the matching catch block are executed then the program control goes to the next statement after the catch handlers

What if the Anticipated Exception is not Raised?

- The catch handlers are ignored if no exceptions occur in the try block and the program executes the first statement after the try and catch blocks

- If no matching catch handler is found for an exception raised in a try block or if an exception occurs in a statement that is not in the try block, the function call stack is unwound – stack unwinding
  - Next outer try-catch block is sought for exception handling
  - Unwinding the function call stack - the function in which the exception was raised but not caught terminates and the program control returns to the place from where the function was invoked
  - If the program control returns to the place within a try block, an attempt to catch the exception is made there

Exception Handling Example # 1

```
#include <iostream>
using namespace std;
int main () {
  try{
    throw 20;
  }
  catch (int e){
    cout << "Exception Number is: " << e << endl;
  }
  return 0;
}
```

Exception Handling Example # 2

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

int main (){
   try {
         throw overflow_error("Divide by zero exception");
   } catch (overflow_error e) {
         cout << e.what();
   }
   return 0;
}
```

what function: gets string identifying exception

Exception Handling Example # 3 (1)

```
#include <iostream>
#include <stdexcept>
int intDiv (int numerator, int denominator) {
 if (denominator == 0){
  throw std::overflow_error("Divide by zero exception");
 }
 return numerator/denominator;
}

int main() {
    int i = 42;
    try {
        i = intDiv(10, 2);
    } catch (std::overflow_error e) {
        std::cout << e.what() << " -> ";
    }
```

Exception Handling Example # 3 (2)

```
    std::cout << i << std::endl;
    try {
        i = intDiv(10, 0);
    } catch (std::overflow_error e) {
        std::cout << e.what() << " -> ";
    }
    std::cout << i << std::endl;
    return 0;
}
```

Standard Template Library (STL)

- STL is a library of classes, algorithms, and iterators that provide many of the basic algorithms and data structures

- The classes are often known as container classes
  - They contain other objects

- Templates are used for implementation

- Examples: vector, list, deque, set, map

Linked List

- It is a linear collection of class objects called nodes that are connected by pointer links
- Allow the program to increase or decrease the size of data structure at run-time and hence can provide better memory utilization than arrays
  - Consume extra memory though to maintain the link to other nodes
- Insertion and deletion in a sorted array can be time-consuming as the elements after the inserted or deleted element would need to be shifted in the appropriate direction
  - A linked list allows efficient insertion or deletion anywhere in the list
- Accessing individual elements in a linked list can be more time consuming as compared to arrays

STL: List

- Double linked list that provides rapid insertion and deletion anywhere in the list in constant time, can be iterated in forward or backward direction

- Useful in sorting algorithms

- Header file to include: `<list>`
  - Contents of the header file are in the namepsace `std`

- Some member functions: `front`, `back`, `push_front`, `pop_front` , `begin`, `end`, `size`, `insert`

http://www.cplusplus.com/reference/list/list/

Using List Container

- Include the header file

- Declare a list object:

```
std::list<double> double_list;
```

- Use built-in functions to insert elements, example:
  - `push_back` function adds new elements to the back
  - `push_front` function adds elements to the front of the list

- For inserting elements in the middle, use the `insert` function. `insert` requires an iterator pointing to the position into which the element should be inserted such that the new element is inserted right before the element currently being pointed to

Iterator

- Iterator provides a means for accessing data stored in container classes, it can point to an item that is part of a larger container of items

- Different containers support different iterator behavior- check documentation – and remember that you can always call the container's begin function to get an iterator

- To create an iterator:

std::*class_name*<*template_parameters*>::iterator *name*

- Example: creating a vector and an iterator of the vector class

```
std::vector<int> myIntVector;
std::vector<int>::iterator myIntVectorIterator;
```

List Example

```
#include <iostream>
#include <list>
using namespace std;
int main() {
 list<int> L;
 L.push_back(0);
 L.push_front(100);
 L.insert(++L.begin(),8);
 L.push_back(50);
 L.push_back(60);
 list<int>::iterator i;
 for(i=L.begin(); i != L.end(); ++i) {
        cout << *i << " ";
 }
 cout << endl;
 return 0;
}
```

Vectors

- Container whose elements are stored in contiguous locations (in a linear sequence) – just like arrays

- Implemented as dynamic arrays

- Unlike regular arrays, storage in a **`vector`** is handled automatically - it can be expanded and contracted as needed

- Vectors consume more memory than arrays when their capacity is handled automatically

- To use a vector container, include the header file **`vector.h`**

Vector Declaration & Initialization

- Syntax of declaring Vectors:

```
vector<type> variable_name (number_of_elements);
```

- In the above declaration, the `number_of_elements` is optional and can be skipped. The below declaration would result in a vector that contains 0 elements:

```
vector<type> variable_name;
```

- Examples:

```
vector<int> age (5);
vector<double> grades (20);
vector<string> names;
```

Vectors: Some Ready-To-Use Functions

Functions	Description
`capacity()`	Return size of allocated storage capacity, that is the number of elements it can hold
`size()`	Returns the number of elements in a vector
`push_back(type element)`	Adds an element to the end of a vector
`empty()`	Returns true if the vector is empty
`clear()`	Erases all elements of the vector
`at(int n)`	Returns the element at index n

Vector: Operators

Operator	Description
`==`	An element by element comparison of two vectors
`[]`	Random access to an element of a vector (usage is similar to that of the operator with arrays)
`=`	Assignment replaces a vector's contents with the contents of another

Using Vectors: testVector.cpp

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

int main (){
  vector<int> storeNumbers (10);
  unsigned int i;

  for (i=0; i<storeNumbers.size(); i++){
    storeNumbers.at(i)=i;
  }
  cout << "The vector storeNumbers contains:";
  for (i=0; i<storeNumbers.size(); i++){
    cout << " " << storeNumbers.at(i);
  }
  cout << endl;
  return 0;
}
```

Comparing Vector and List

- Insertions and deletions: vector has relatively costly insertions and deletions into the middle of the vector, whereas the list allows cheap insertions or deletions

- Random access: vector offers fast random access but list offers slow access

- For operations like sorting, you might need a scratch vector if you are sorting a vector but with list no scratch space is needed

- Note that the header files are different if you would like to use vector and list

References

- http://www.cplusplus.com

- C++, How to Program, Dietel & Dietel

- http://www.sgi.com/tech/stl/List.html

- http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.htm

- http://www.cprogramming.com/tutorial/stl/stllist.html