NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
CS 201 – DATA STRUCTURES LAB
(FALL 2018)

Instructors: Mr. Faizan Yousuf, Ms. Safia, Ms. Maham Mobin

LAB SESSION # 01

Outline

- How to do effective Debugging?
- What are pointers?
- Using pointers in C++
- C++ Dynamic Memory Allocation
- Dynamic Allocation of Objects
- Task

How to do effective Debugging?

Debugging (or program testing) is the process of making a program behave as intended. The difference between intended behavior and actual behavior is caused by 'bugs' (program errors) which are to be corrected during debugging.

Debugging is often considered a problem for three reasons:

1. The process is too costly (takes too much effort).

2. After debugging, the program still suffers from bugs.

3. When the program is later modified, bugs may turn up in completely unexpected places.

In general, there are two sources for these problems: poor program design and poor debugging techniques. For instance, the problem of too costly debugging may be due to the presence of many bugs (poor program design), or to a debugging technique where too few bugs are found for each test run or each man-day (poor debugging technique).

Assuming that program design is adequate. In other words, we will consider this situation: A program or a set of intimately related programs are given. They contain an unknown number of bugs. Find and correct these bugs as fast as possible.

Debugging is carried out through test runs: execution of the program or parts of it with carefully selected input (so-called test data). Execution is normally done on a computer, but in some cases it can be advantageous to do a 'desk execution'.

TOP-DOWN DEBUGGING VERSUS BOTTOM-UP

In bottom-up debugging, each program module is tested separately in special test surroundings (module testing). Later, the modules are put together and tested as a whole (system testing).

In top-down debugging, the entire program is always tested as a whole in nearly the final form. The program tested differs from the final form in two ways:

1. At carefully selected places, output statements are inserted to print intermediate values (test output).

2. Program parts which are not yet developed are absent or replaced by dummy versions.

With this technique, test data has the same form as input for the final program.

What are pointers?

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is –

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration –

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *ch     // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Using pointers in C++

There are few important operations, which we will do with the pointers very frequently. **(a)** We define a pointer variable. **(b)** Assign the address of a variable to a pointer. **(c)** Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

```
#include <iostream>
using namespace std;
int main () {
  int  var = 20;   // actual variable declaration.
  int  *ip;        // pointer variable
  ip = &var;       // store address of var in pointer variable
  cout << "Value of var variable: ";
  cout << var << endl;
```

```
   // print the address stored in ip pointer variable
   cout << "Address stored in ip variable: ";
   cout << ip << endl;
   // access the value at the address available in pointer
   cout << "Value of *ip variable: ";
   cout << *ip << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20
```

C++ Dynamic Memory Allocation:

Remember that memory allocation comes in two varieties:

Static (compile time): Sizes and types of memory (including arrays) must be known at compile time, allocated space given variable names, etc.

Dynamic (run-time): Memory allocated at run time. Exact sizes (like the size of an array) can be variable. Dynamic memory doesn't have a name (names known by compiler), so pointers used to link to this memory

Allocate dynamic space with operator new, which returns address of the allocated item. Store in a pointer:

```
int * ptr = new int;                    // one dynamic integer
double * nums = new double[size];     // array of doubles, called "nums"
```

Clean up memory with operator delete. Apply to the pointer. Use delete [] form for arrays:

```
delete ptr;          // deallocates the integer allocated above
delete [] nums;    // deallocates the double array above
```

Remember that to access a single dynamic item, dereference is needed:

```
cout << ptr;              // prints the pointer contents
cout << *ptr;             // prints the target
```

For a dynamically created array, the pointer attaches to the starting position of the array, so can act as the array name:

```
nums[5] = 10.6;
cout << nums[3];
```

Dynamic Allocation of Objects:

Just like basic types, objects can be allocated dynamically, as well.

But remember, when an object is created, the constructor runs. Default constructor is invoked unless parameters are added:

```
Fraction * fp1, * fp2, * flist;
fp1 = new Fraction;          // uses default constructor
fp2 = new Fraction(3,5);     // uses constructor with two parameters

flist = new Fraction[20];    // dynamic array of 20 Fraction objects
                             //  default constructor used on each
```

Deallocation with delete works the same as for basic types:

```
delete fp1;
delete fp2;
delete [] flist;
```

Notation: dot-operator vs. arrow-operator:

dot-operator requires an object *name* (or effective name) on the left side

```
objectName.memberName          // member can be data or function
```

The arrow operator works similarly as with structures.

```
pointerToObject->memberName
```

Remember that if you have a pointer to an object, the pointer name would have to be dereferenced first, to use the dot-operator:

```
(*fp1).Show();
```

Arrow operator is a nice shortcut, avoiding the use or parintheses to force order of operations:

```
fp1->Show();                   // equivalent to (*fp1).Show();
```

When using dynamic allocation of objects, we use pointers, both to single object and to arrays of objects. Here's a good rule of thumb:
For pointers to single objects, arrow operator is easiest:

```
fp1->Show();
fp2->GetNumerator();
```

```
fp2->Input();
```

For dynamically allocated arrays of objects, the pointer acts as the array name, but the object "names" can be reached with the bracket operator. Arrow operator usually not needed:

```
flist[3].Show();
flist[5].GetNumerator();


// note that this would be INCORRECT, flist[2] is an object, not a pointer
flist[2]->Show();
```

LAB TASK:

Question 1:

Create a Student class (having appropriate attributes and functions) and do following steps.

1. Dynamically allocate memory to 5 objects of a class.
2. Order data in allocated memories by Student Name in descending.

Question Number 2:

Write a program to calculate salary of n employees, when following information is given input through keyboard:

- If age of employee is greater than 50, experience is greater than 10 &working hours are greater than 240 than the hourly wage rate is 500rs per hour.
- If age of employee is less than equal to 50 and greater than 40, experience is less than equal to 10 and greater than 6 & working hours are greater than 200 and less than equal to 240 than the hourly wage rate is 425rs per hour.
- If age of employee is less than equal to 40 and greater than 30, experience is less than equal to 6 and greater than 3 & working hours are greater than 160 and less than equal to 200 than the hourly wage rate is 375rs per hour.
- If age of employee is less than equal to 30 and greater than 22, experience is less than equal to 3 and greater than 1 & working hours are greater than 120 and less than equal to 160 than the hourly wage rate is 300rs per hour.
- Otherwise print invalid parameters.

Note: Use pointers for memory allocation. Set of Employees should be traverse by increasing Address Pointer. Use debugging techniques to analyze change in memory address.