



**National University of Computer & Emerging Sciences,
Karachi**



**Computer Science Department
Lab Manual - 01**

Course Code: CL-218	Course : Data Structure Lab
Instructors	Mubashra Fayyaz, Aqsa Zahid

Contents:

- 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:

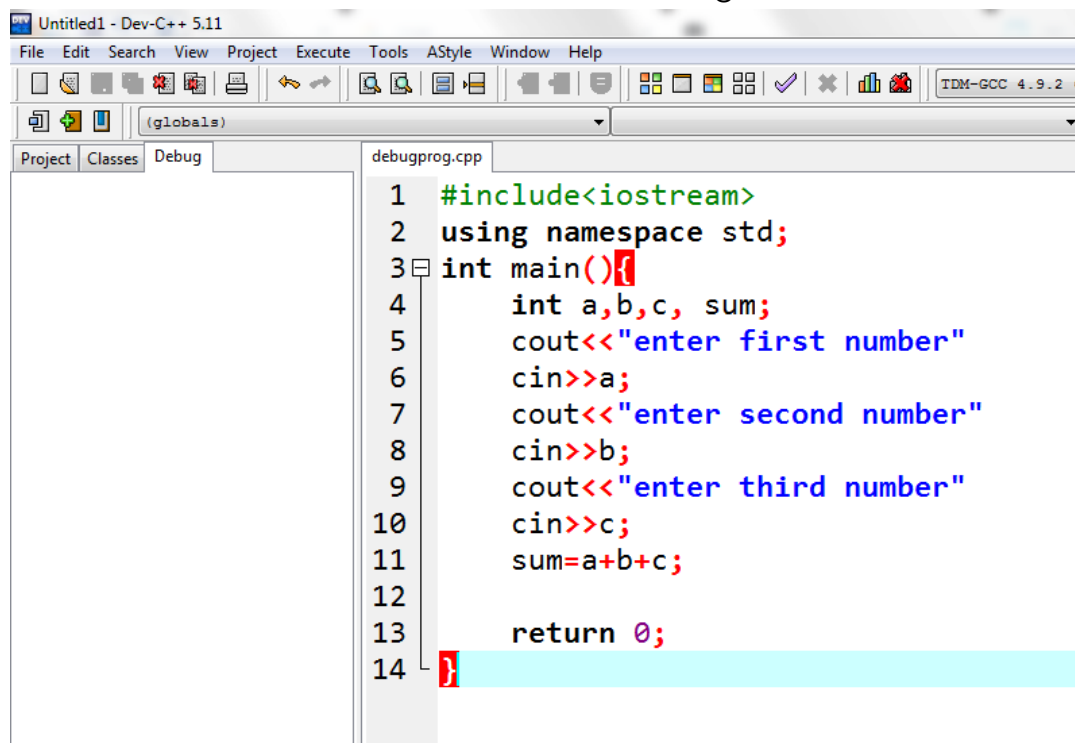
1. At carefully selected places, output statements are inserted to print intermediate values (test output).
2. Program parts which are not yet developed are absent or replaced by dummy versions.

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

Debugging using dev

Step 1:

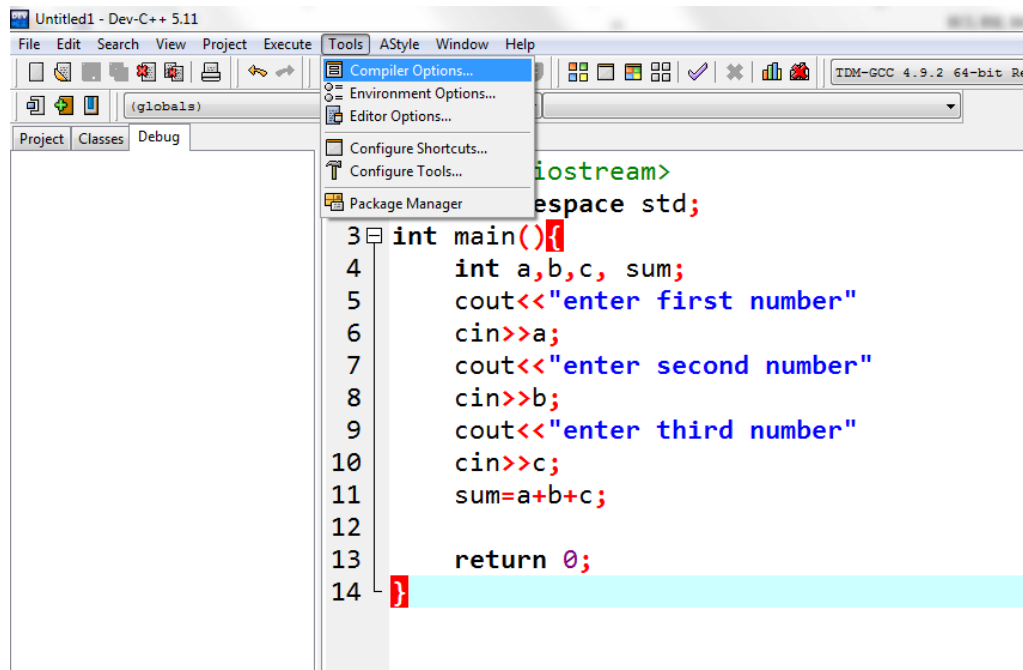
Write and save the code that u want to debug.



```
1  #include<iostream>
2  using namespace std;
3  int main(){
4      int a,b,c, sum;
5      cout<<"enter first number"
6      cin>>a;
7      cout<<"enter second number"
8      cin>>b;
9      cout<<"enter third number"
10     cin>>c;
11     sum=a+b+c;
12
13     return 0;
14
```

Step 2:

Click tools then compiler options

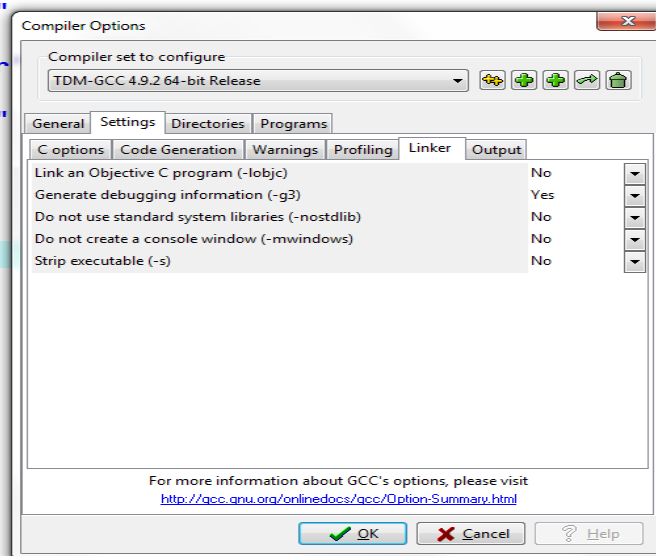


Step 3:

In compiler options popup window, Go to Settings->Linker. Make sure that Generate debugging information is set to yes and click ok.

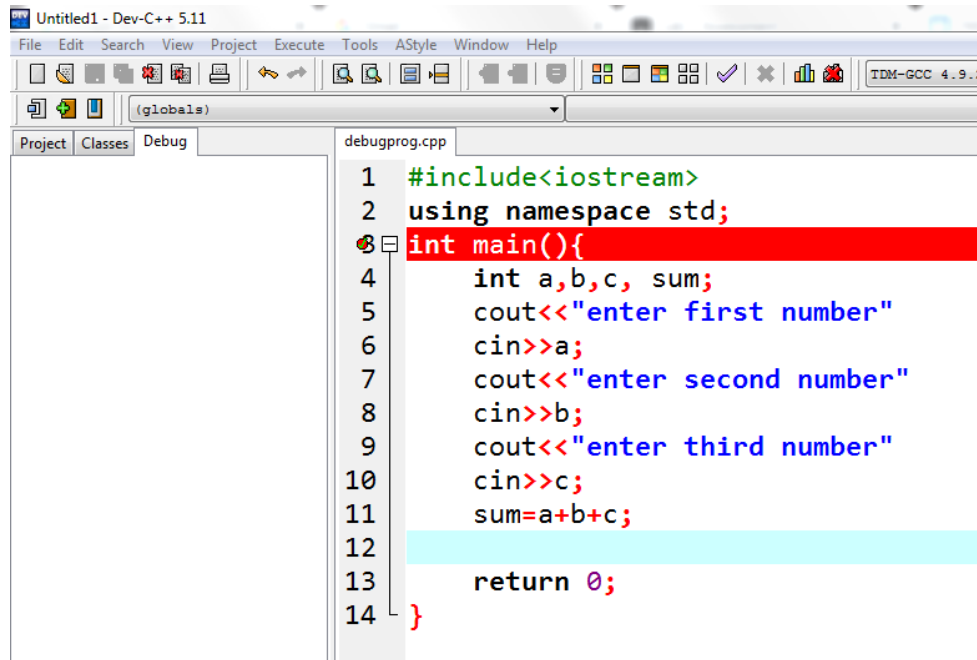
```
std;
```

```
um;
first number"
second number"
third number"
```



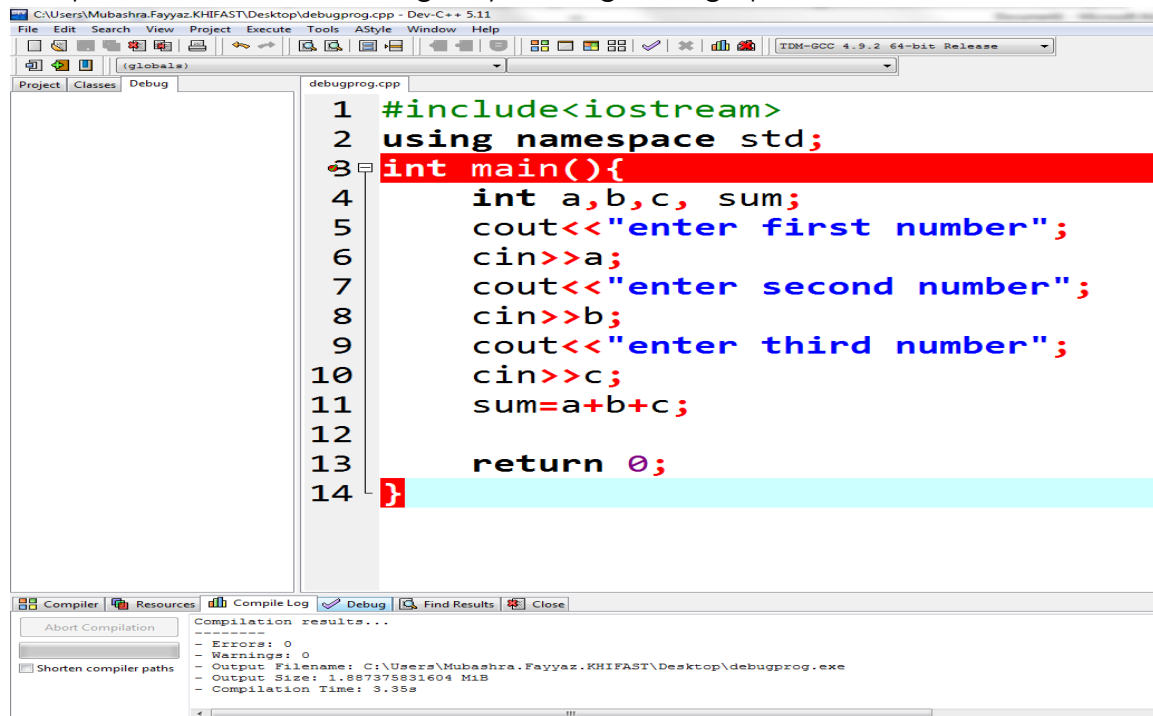
Step 4:

Adding breakpoints, Note that you can add more than one breakpoint in a program. Breakpoints are added by clicking the line number in the Gutter.



Step 5:

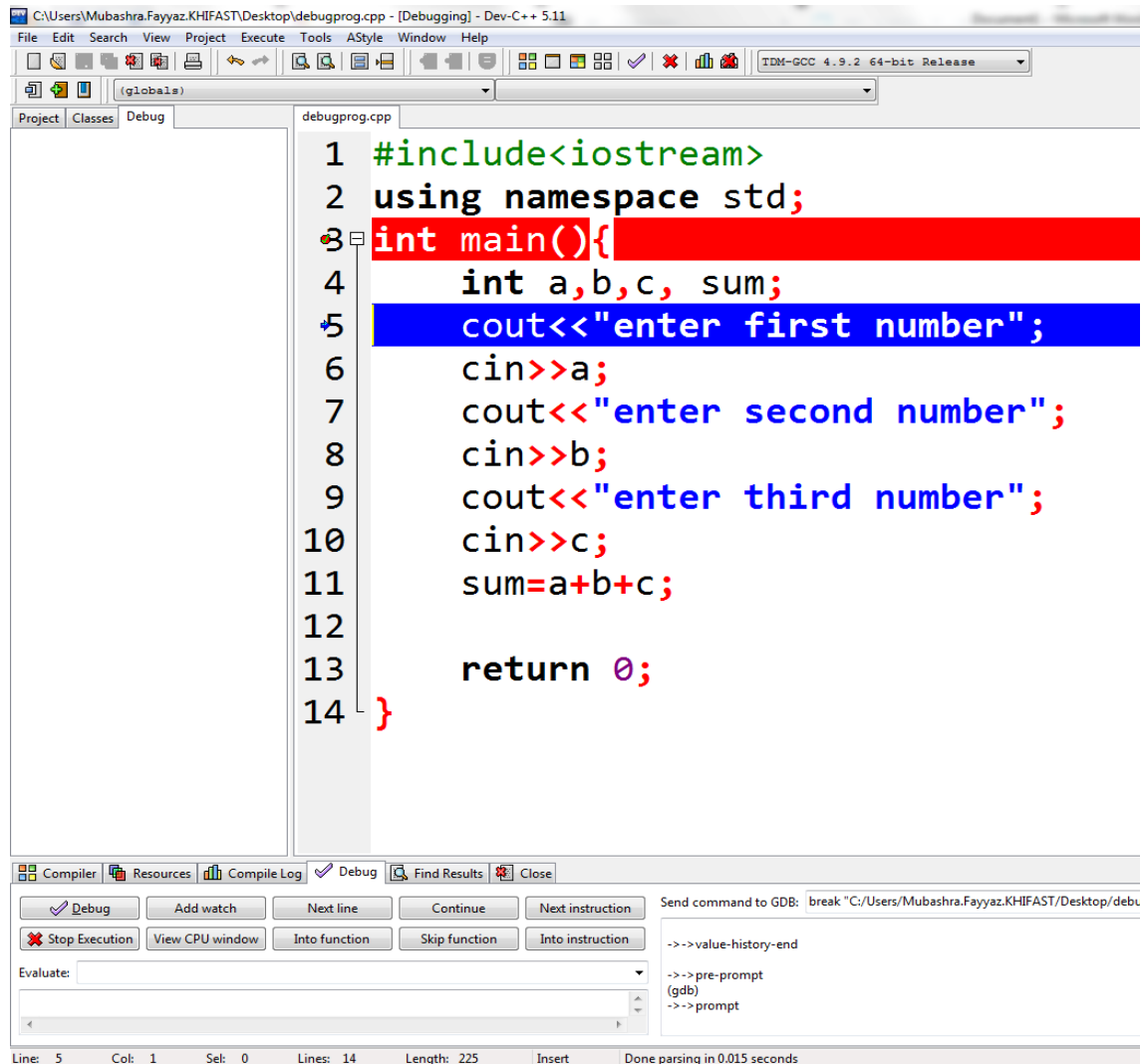
compile the code and debug it by clicking debug option in the bottom left.



Now you are ready to launch the debugger, by pressing F8 or clicking the debug button. If everything goes well, the program will start, and then stop at the first breakpoint.

Then you can step through the code, entering function calls, by pressing Shift-F7 or the "step into" button, or stepping over the function calls, by pressing F7 or the "next step" button.

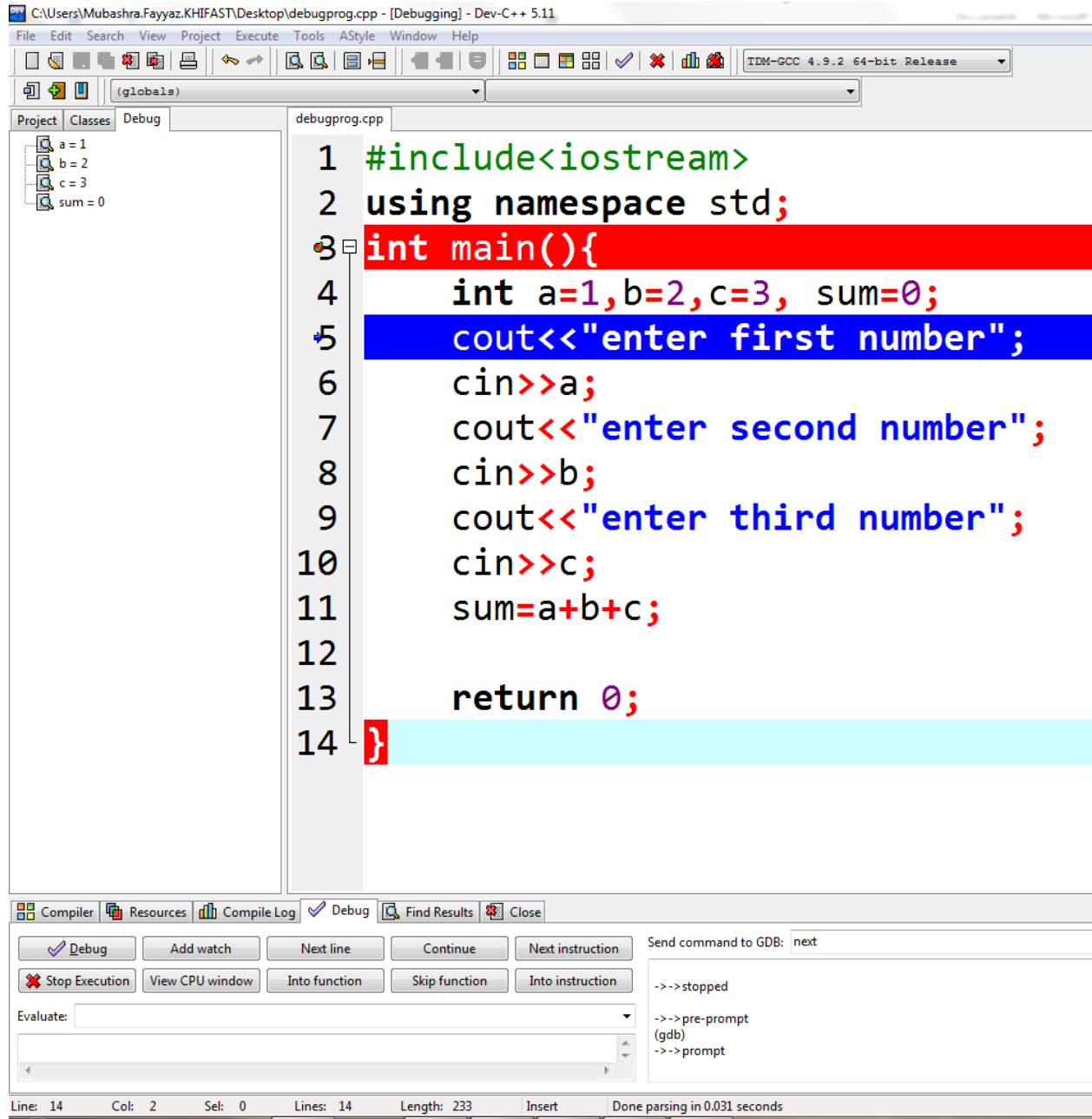
You can press Ctrl-F7 or the "continue" button to continue execution till the next breakpoint. At any time, you can add or remove breakpoints.



When the program stopped at a breakpoint and you are stepping through the code, you can display the values of various variables in your program by putting your mouse over them, or you can display variables and expressions by pressing F4 or the "add watch" button and typing the expression.

Adding watch:

1. Click Add watch button
2. Enter the variable name in the popup window.
3. Click debug and you can move further by next line and other such buttons.



Values of variables can be monitored throughout the program. Also works for user input.

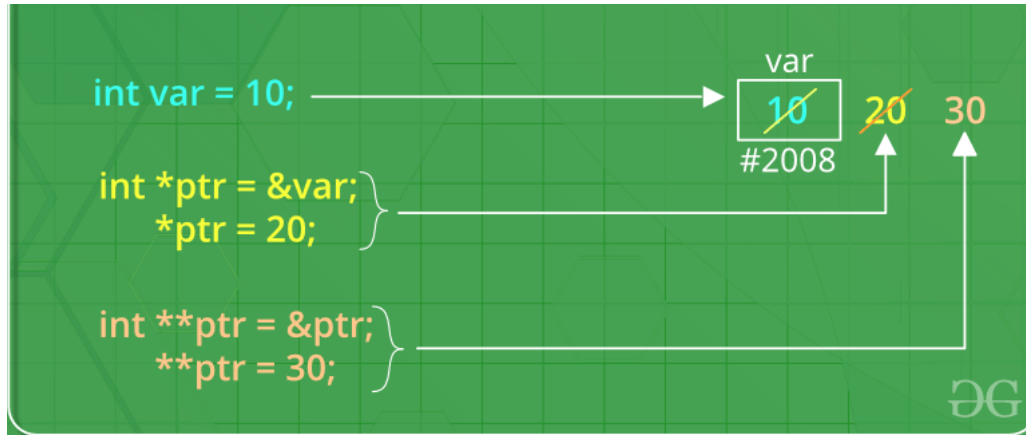
Pointers:

Pointers are symbolic representation of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. It's general declaration in C/C++ has the format:

Syntax:

`datatype *var_name;`

`int *ptr; //ptr can point to an address which holds int data`



How to use a pointer?

- Define a pointer variable
- Assigning the address of a variable to a pointer using unary operator (&) which returns the address of that variable.
- Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.

The reason we associate data type to a pointer is **that it knows how many bytes the data is stored in**. When we increment a pointer, we increase the pointer by the size of data type to which it points.

There are 3 ways to pass C++ arguments to a function:

- call-by-value
- call-by-reference with pointer argument
- call-by-reference with reference argument


```

#include <bits/stdc++.h>
using namespace std;
//Pass-by-Value
int square1(int n)
{
    //Address of n in square1() is not the same as n1 in main()
    cout << "address of n1 in square1(): " << &n << "\n";

    // clone modified inside the function
    n *= n;
    return n;
}
//Pass-by-Reference with Pointer Arguments
void square2(int *n)
{
    //Address of n in square2() is the same as n2 in main()
    cout << "address of n2 in square2(): " << n << "\n";

    // Explicit de-referencing to get the value pointed-to
    *n *= *n;
}
//Pass-by-Reference with Reference Arguments
void square3(int &n)
{
    //Address of n in square3() is the same as n3 in main()
    cout << "address of n3 in square3(): " << &n << "\n";

    // Implicit de-referencing (without '*')
    n *= n;
}
void CHECK()
{
    //Call-by-Value
    int n1=8;
    cout << "address of n1 in main(): " << &n1 << "\n";
    cout << "Square of n1: " << square1(n1) << "\n";
    cout << "No change in n1: " << n1 << "\n";

    //Call-by-Reference with Pointer Arguments
    int n2=8;
    cout << "address of n2 in main(): " << &n2 << "\n";
    square2(&n2);
    cout << "Square of n2: " << n2 << "\n";
    cout << "Change reflected in n2: " << n2 << "\n";

    //Call-by-Reference with Reference Arguments
    int n3=8;
    cout << "address of n3 in main(): " << &n3 << "\n";

```

```

square3(n3);
cout << "Square of n3: " << n3 << "\n";
cout << "Change reflected in n3: " << n3 << "\n";
}
//Driver program
int main()
{
    CHECK ();
}

```

Output:

```

address of n1 in main(): 0x7ffcdb2b4a44
address of n1 in square1(): 0x7ffcdb2b4a2c
Square of n1: 64
No change in n1: 8
address of n2 in main(): 0x7ffcdb2b4a48
address of n2 in square2(): 0x7ffcdb2b4a48
Square of n2: 64
Change reflected in n2: 64
address of n3 in main(): 0x7ffcdb2b4a4c
address of n3 in square3(): 0x7ffcdb2b4a4c
Square of n3: 64
Change reflected in n3: 64

```

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];`

EXAMPLE CODE:

`#include <iostream>`

`using namespace std;`

`int main ()`

`{`

`// Pointer initialization to null`

`int* p = NULL;`

`// Request memory for the variable`

`// using new operator`

`p = new(nothrow) int;`

`if (!p)`

`cout << "allocation of memory failed\n";`

`else`

`{`

`// Store value at allocated address`

`*p = 29;`

`cout << "Value of p: " << *p << endl;`

`}`

`// Request block of memory`

`// using new operator`

`float *r = new float(75.25);`

`cout << "Value of r: " << *r << endl;`

`// Request block of memory of size n`

`int n = 5;`

`int *q = new(nothrow) int[n];`

`if (!q)`

`cout << "allocation of memory failed\n";`

`else`

`{`

`for (int i = 0; i < n; i++)`

`q[i] = i+1;`

```

        cout << "Value store in block of memory: ";
        for (int i = 0; i < n; i++)
            cout << q[i] << " ";
    }

    // freed the allocated memory
    delete p;
    delete r;

    // freed the block of allocated memory
    delete[] q;

    return 0;
}
Output:

```

Value of p: 29

Value of r: 75.25

Value store in block of memory: 1 2 3 4 5

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;
```

```
fp2 = new Fraction(3,5);
```

```
// uses default constructor
```

```
// 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 of parentheses 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 then 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 then 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 then 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 then 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.