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

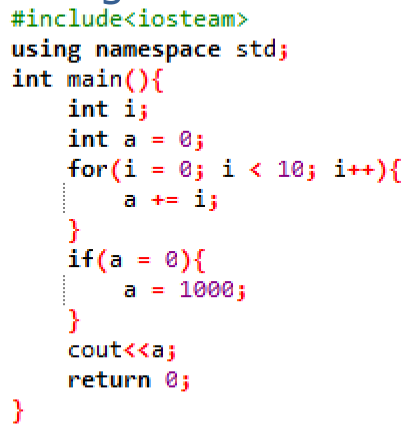
- How to do effective Debugging?
- What are pointers?
- Using pointers in C++
- C++ Dynamic Memory Allocation
- Dynamic Allocation of Objects
- Task

The Definition of Bug:

Part of the code which would result in an error, fault or malfunctioning of the program.

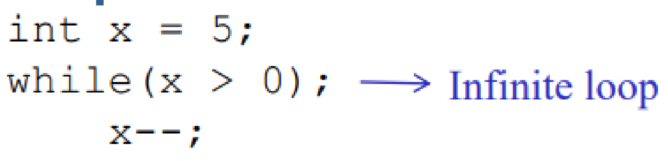
Common Bugs:

1. Using '=' instead of '=='

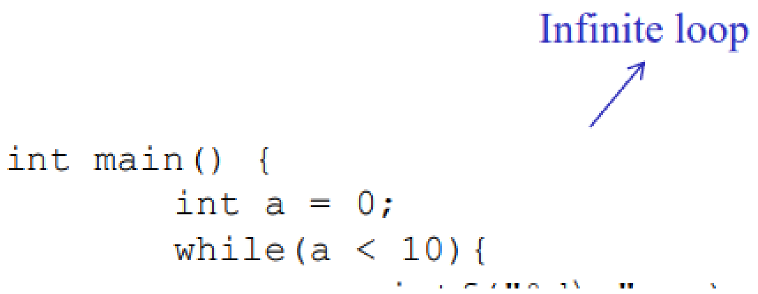


```
#include<iosteam>
using namespace std;
int main(){
    int i;
    int a = 0;
    for(i = 0; i < 10; i++){
        a += i;
    }
    if(a = 0){
        a = 1000;
    }
    cout<<a;
    return 0;
}
```

2. Loop error



```
int x = 5;
while(x > 0);
     x--;
```



```
int main() {
        int a = 0;
        while(a < 10){
```

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.

How to use the Dev C++ debugger?

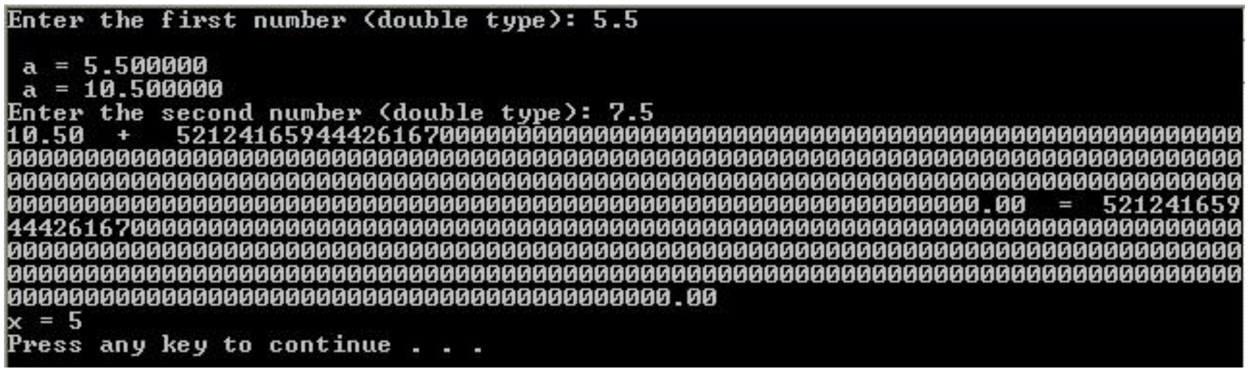
Debuggers are programs which allow you to execute your program in a controlled manner, so that you can look inside your program to find a logic or run-time bug.

Example: When we compile, and then execute the following C program:

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(void)
{
   int x = 5;
   double a, b;
   printf("Enter the first number (double type): ");
   scanf("%lf", &a);
   printf("\n a = %f", a);
   a = a + x;
   printf("\n a = %f", a);
   printf("\nEnter the second number (double type): ");
   scanf("%f", &b);
   printf("%.2f  +   %.2f  =  %.2f\n", a, b, a + b);
   printf("x = %d\n", x);
   system("PAUSE");
   return 0;
}
```

we get the following output for input values a = 5.5 and b = 7.5:



```
Enter the first number (double type): 5.5

 a = 5.500000
 a = 10.500000
Enter the second number (double type): 7.5
10.50  +   5212416594442616700000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000.00  =  521241659
44426167000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000.00
x = 5
Press any key to continue . . .
```

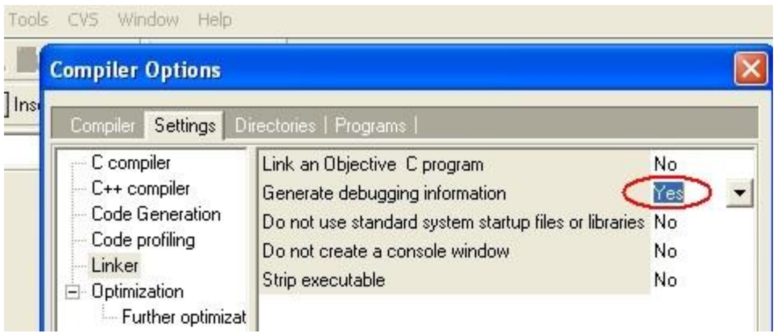
The output indicates that the program has one or more logic errors. We use the Dev C++ debugger to find the error(s).

The following are the steps to debug a single source C program by the Dev C++ debugger:

Step 1: Configure Linker Options

- Go to the "Tools" menu and select "Compiler Options".

- In the "Settings" tab, click on "Linker" in the left panel, and change "Generate debugging information" to "Yes":



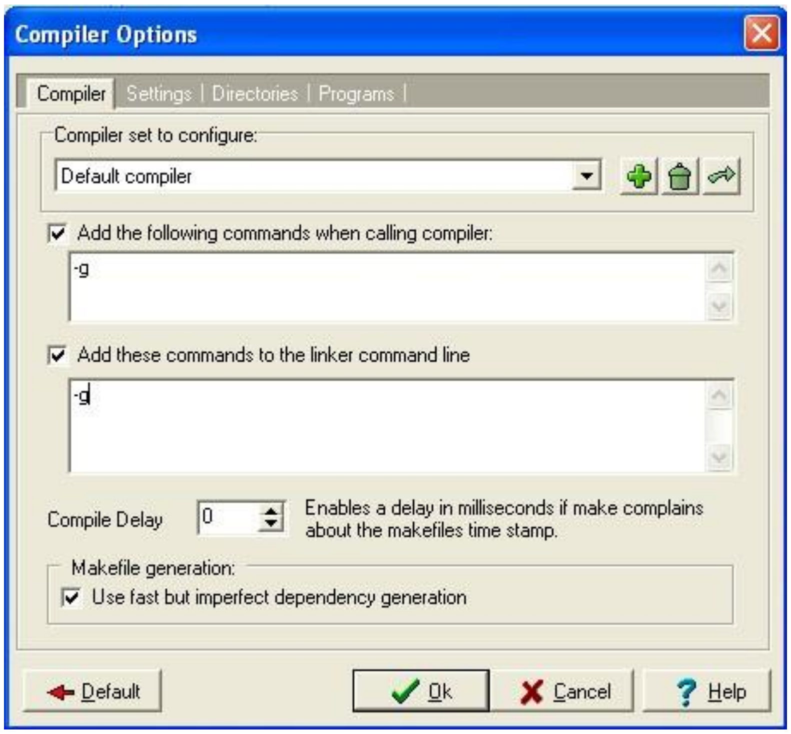
Link an Objective C program No
Generate debugging information [✓ Yes]
Do not use standard system startup files or libraries No
Do not create a console window No
Strip executable No

Make sure you are not using any optimization options (they are not good for debug mode).

- Click "OK".

Step 2 Configure Compiler Options:

1. "Go to the "Tools/Compiler Options" menu.
2. Put "-g" (without the quotation marks) in *both* boxes on the Compiler tab.
3. Make sure to check *both* boxes labeled "Add the following commands ..."."



Compiler set to configure: Default compiler

☑ Add the following commands when calling compiler:

```
-g
```

☑ Add these commands to the linker command line

```
-g
```

Compile Delay 0 Enables a delay in milliseconds if make complains about the makefiles time stamp.

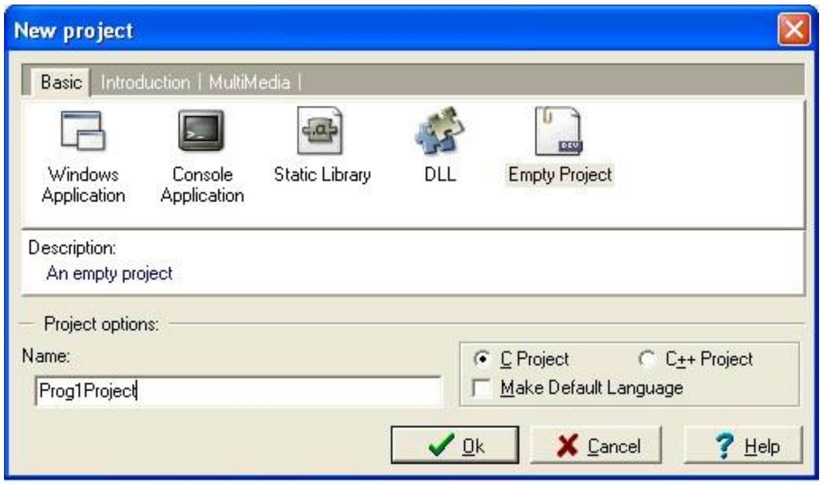
Makefile generation:

☑ Use fast but imperfect dependency generation

Step 3: Create a new project.

A "project" can be considered as a container that is used to store all the elements that are required to compile a single-source or a multi-source program.

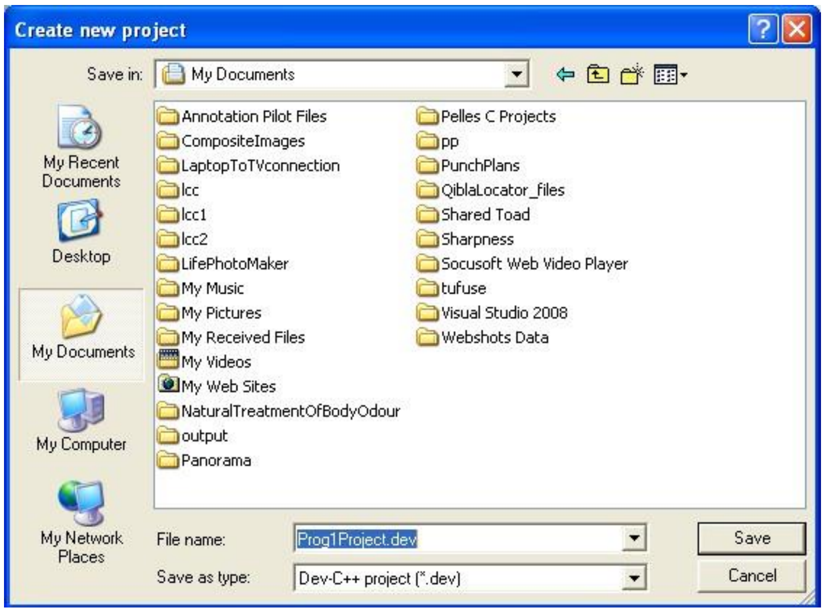
- Go to the "File" menu and select "New", "Project...".
- Choose "Empty Project" and make sure "C project" is selected.
  Here you will also give your project a name. You can give your project any valid filename, but keep in mind that the name of your project will also be the name of your final executable.
- Once you have entered a name for your project, click "OK".
- Dev-C++ will now ask you to save your project. It is saved with the extension **.dev**



Description: An empty project

Name: Prog1Project

[✓ C Project]



Save in: My Documents

File name: Prog1Project.dev

Save as type: Dev-C++ project (*.dev)

Go to **Project → Project Options → Parameters** make sure you do not have any optimization options (like -O2 or -O3; but -O0 is okay because it means no optimization) or strip option (-s).

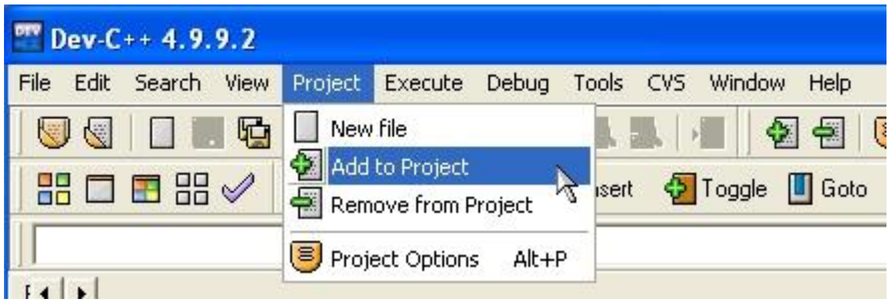
Step 4: Create/add source file(s).

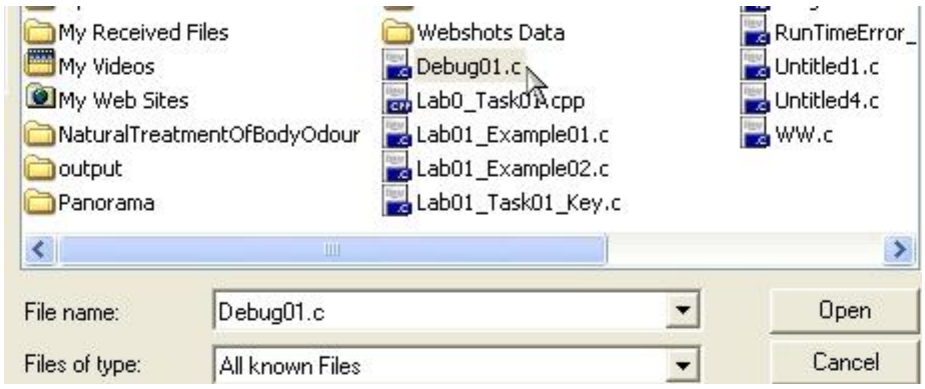
You can add empty source files one of two ways:

- Go to the "File" menu and select "New Source File" (or just press CTRL+N) OR
- Go to the "Project" menu and select "New File".
  Note that Dev-C++ will not ask for a filename for any new source file until you attempt to:
  1. Compile
  2. Save the project
  3. Save the source file
  4. Exit Dev-C++

You can add pre-existing source files one of two ways:

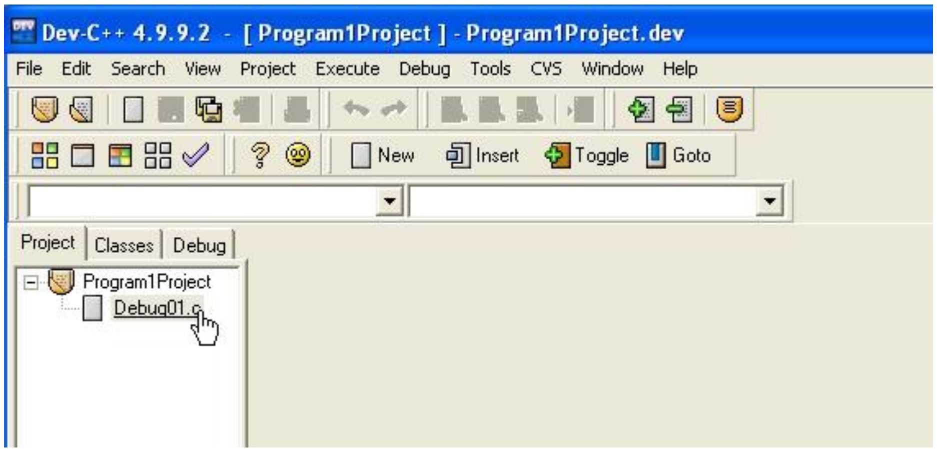
- Go to the "Project" menu and select "Add to Project" OR
- Right-click on the project name in the left-hand panel and select "Add to Project".





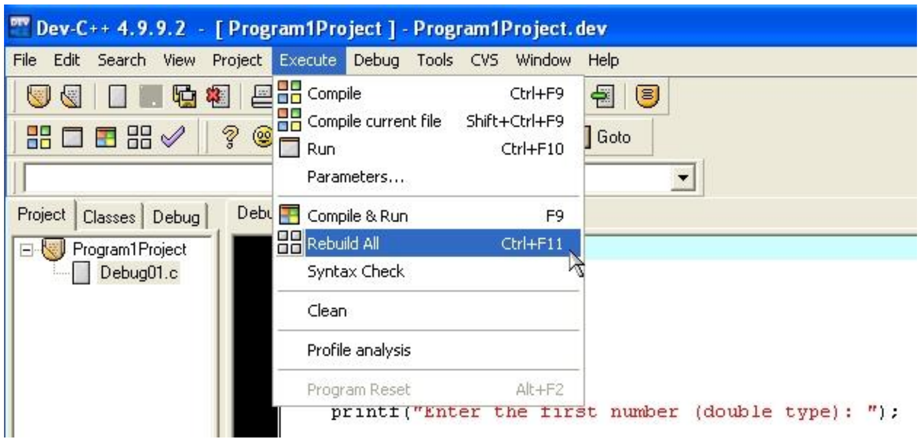
Display the C file added to the project in the editor window by clicking on its file name under

the **Project** button:

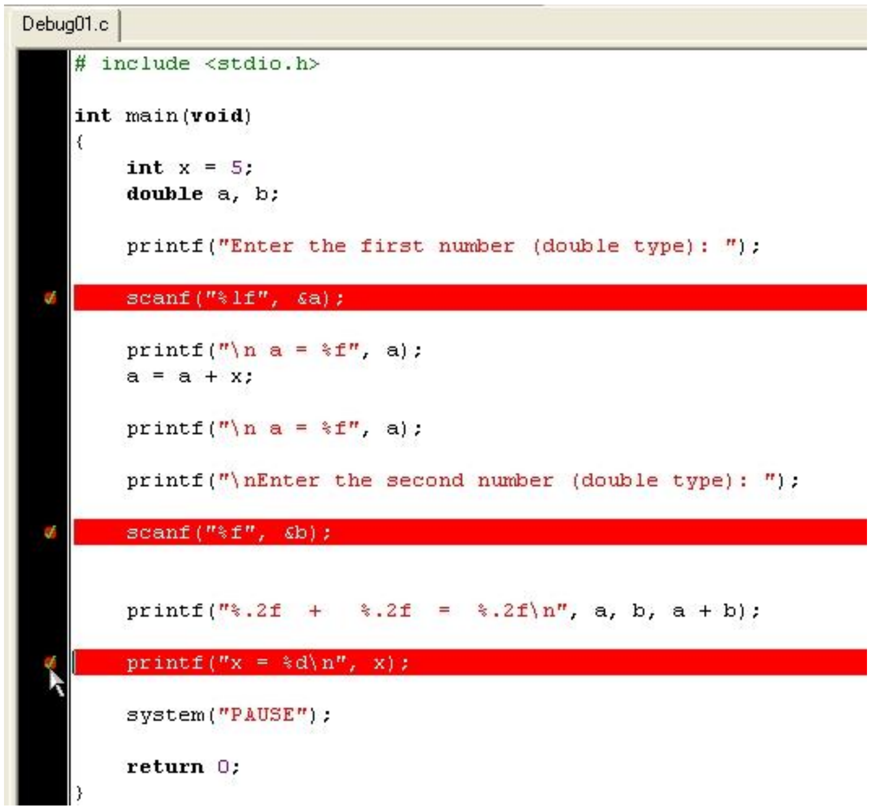

Dev-C++ 4.9.9.2 - [Program1Project] - Program1Project.dev
File Edit Search View Project Execute Debug Tools CVS Window Help
New Insert Toggle Goto
Project Classes Debug
Program1Project
Debug01.c

Step 7: Debug

After that, do a full rebuild (**Ctrl-F11**), then set breakpoint(s) where you want the debugger to stop (otherwise it will just run the program). To set a breakpoint on a line, just click on the gutter (the gray band on the left), or press **Ctrl-F5**.

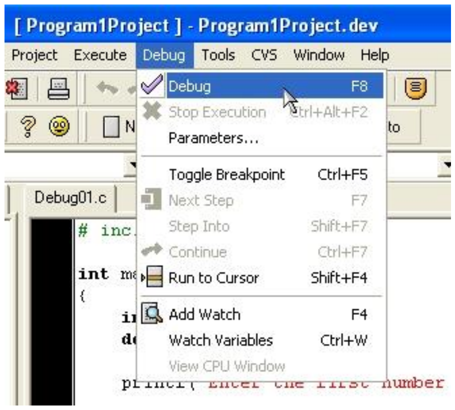


```
printf("Enter the first number (double type): ");
```



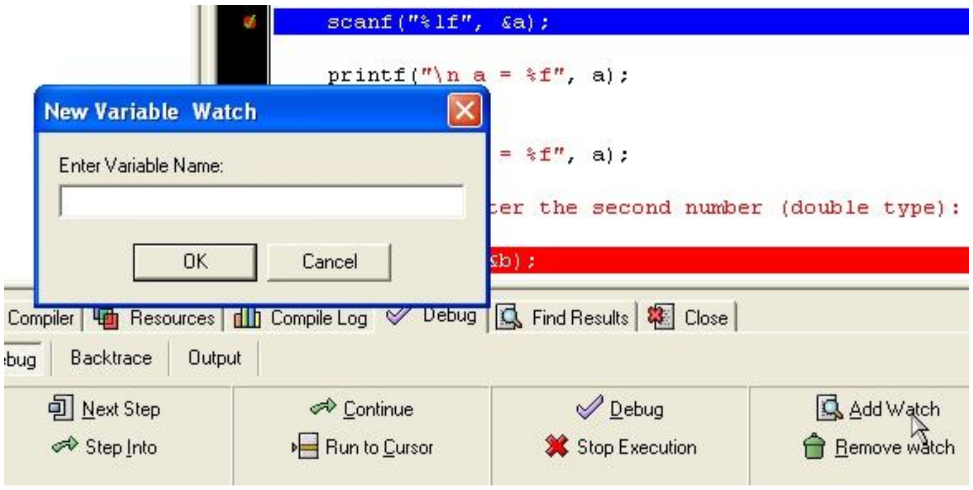
```
# include <stdio.h>

int main(void)
{
    int x = 5;
    double a, b;

    printf("Enter the first number (double type): ");

    scanf("%lf", &a);

    printf("\n a = %f", a);
    a = a + x;

    printf("\n a = %f", a);

    printf("\nEnter the second number (double type): ");

    scanf("%f", &b);


    printf("%.2f  +  %.2f  =  %.2f\n", a, b, a + b);

    printf("x = %d\n", x);

    system("PAUSE");

    return 0;
}
```

Now you are ready to launch the debugger, by pressing **F8** or clicking the **Debug** button:



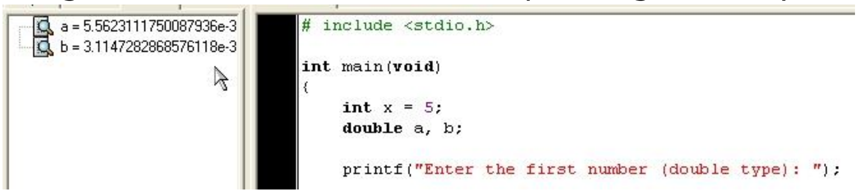
The program will start, and then stop at the first breakpoint. You can then step through the code, entering function calls, by pressing Shift-F7 or the "**step into**" button, or stepping over the function calls, by pressing F7 or the "**next step**" button. You can press Ctrl-F7 or the "**continue**" button to continue execution untill the next breakpoint. Click the "**Run to cursor**" icon to run your program and pause at the current source code cursor location. At any time, you can add or remove breakpoints.

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



```
scanf("%lf", &a);
printf("\n a = %f", a);
= %f", a);
er the second number (double type):
&b);
```

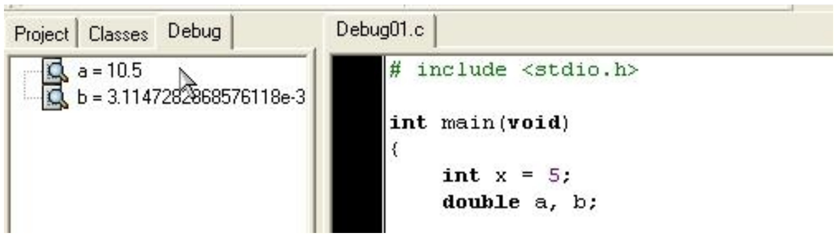
The variables added by "Add Watch" are initially given undefined values (?) or whatever garbage value that is in the corresponding memory location:



```
a = 5.5623111750087936e-3
b = 3.1147282868576118e-3
```

```
# include <stdio.h>

int main(void)
{
    int x = 5;
    double a, b;

    printf("Enter the first number (double type): ");
```

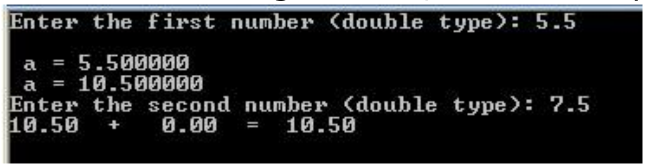
When we execute the **scanf** to input the value of the variable ***a***, the value of the variable changes accordingly:



```
a = 10.5
b = 3.1147282868576118e-3
```

```
# include <stdio.h>

int main(void)
{
    int x = 5;
    double a, b;
```

We detect the logic error in the program when we enter **7.5** as the value of the variable ***b***; the watch for ***b*** does not change to **7.5**, and the output for b is **0**:



```
Enter the first number (double type): 5.5

 a = 5.500000
 a = 10.500000
Enter the second number (double type): 7.5
10.50  +   0.00  =  10.50
```

What are pointers?

A **pointer** is a variable whose value is the address of a memory location where data is to be kept. 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.

Few important points are as follows:

1. Declare and assign an initial value for pointer variable.
2. Always assign address of similar data type.
3. Use de-reference to access the content store at the pointed location.

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: ";

   // 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 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 multiply 2 Matrices and store the result in a new matrix.

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