

C++ Program Design

-- How to design your first program

Junjie Cao @ DLUT

Summer 2016

<http://jjcao.github.io/cPlusPlus>

design your program

- design *before coding*
- programming is like architecture
 - Spend lots of time fixing problems that could been avoided with a little thinking ahead



Step 1: Define the problem

- what problem your program is attempting to solve
- state this in a sentence or two
 - I want to write a phone book application to help me keep track of my friend's phone numbers.
 - I want to write a program that will read information about stocks from the internet and predict which ones I should buy.
- The worst thing you can do is write a program that doesn't actually do what you (or your boss) wanted!

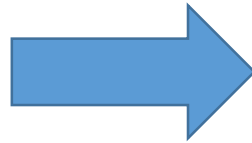
Step 2: Define your tools, targets, and backup plan

- When you are an experienced programmer:
 - Understanding who your target users are and what they want.
 - Defining what target architecture and/or OS your program will run on.
 - Determining what set of tools you will be using.
 - Determining whether you will write your program alone or as part of a team.
 - Collecting requirements (a documented list of what the program needs to do).
 - Defining your testing/feedback/release strategy.
 - **Determining how you will back up your code - version control (e.g. github).**

Step 3: Break hard problems down into easy problems

- **continuously** splitting complex tasks into simpler ones until each of them is manageable
- **top down**

Write report on carrots

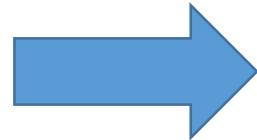


- Write report on carrots
 - Do research on carrots
 - Write outline
 - Fill in outline with detailed information about carrots
 - Add table of contents

Write report on carrots



- Write report on carrots
 - Do research on carrots
 - Write outline
 - Fill in outline with detailed information about carrots
 - Add table of contents



- Write report on carrots
 - Do research on carrots
 - Go to library and get book on carrots
 - Look for information about carrots on internet
 - Take notes on relevant sections from reference material
 - Write outline
 - Information about growing
 - Information about processing
 - Information about nutrition
 - Fill in outline with detailed information about carrots
 - Add table of contents

bottom up

- Pick out clothes
- Get dressed
- Eat breakfast
- Drive to work
- Brush your teeth
- Get out of bed
- Prepare breakfast
- Get in your car
- Take a shower



- Get from bed to work
 - Bedroom things
 - Get out of bed
 - Pick out clothes
 - Bathroom things
 - Take a shower
 - Brush your teeth
 - Breakfast things
 - Prepare breakfast
 - Eat breakfast
 - Transportation things
 - Get in your car
 - Drive to work

task hierarchy

- The top level task => main()
 - “Write a report on carrots”
 - “Get from bed to work”
- The subitems => functions

Step 4: Figure out the sequence of events

```
1  int main()
2  {
3      getOutOfBed();
4      pickOutClothes();
5      takeAShower();
6      getDressed();
7      prepareBreakfast();
8      eatBreakfast();
9      brushTeeth();
10     getInCar();
11     driveToWork();
12 }
```

```
1  int main()
2  {
3      // Get first number from user
4      getUserInput();
5
6      // Get mathematical operation from user
7      getMathematicalOperation();
8
9      // Get second number from user
10     getUserInput();
11
12     // Calculate result
13     calculateResult();
14
15     // Print result
16     printResult();
17 }
```

- comment each of these out until you actually write them
- work on them one at a time, testing each as you go.

Step 5: Figure out the data inputs and outputs for each task

- a hierarchy and a sequence of events
- input data => parameters.
- output => a return value or parameters

function prototype

```
1 | int getUserInput();
```

```
1 | int calculateResult(int input1, int op, int input2);
```

Step 6: Write the task details

- write implementation.
 - simple & straightforward or
 - be broken down into subtasks.

```
1  int getMathematicalOperation()
2  {
3      std::cout << "Please enter which operator you want (1 = +, 2 =
4      -, 3 = *, 4 = /): ";
5
6      int op;
7      std::cin >> op;
8
9      // What if the user enters an invalid character?
10     // We'll ignore this possibility for now
11
12     return op;
13 }
```

Step 7: Connect the data inputs and outputs

```
1 // result is a temporary value used to transfer the output of calcul
2 ateResult()
3 // into an input of printResult()
4 int result = calculateResult(input1, op, input2); // temporarily sto
   re the calculated result in result
   printResult(result);
```

```
1 printResult( calculateResult(input1, op, input2) );
```

```

1 // #include "stdafx.h" // uncomment if using visual studio
2 #include <iostream>
3
4 int getUserInput()
5 {
6     std::cout << "Please enter an integer: ";
7     int value;
8     std::cin >> value;
9     return value;
10 }
11
12 int getMathematicalOperation()
13 {
14     std::cout << "Please enter which operator you want (1 = +, 2 = -, 3 = *, 4 = /): ";
15     int op;
16     std::cin >> op;
17
18     // What if the user enters an invalid operator?
19     // We'll ignore that for now
20
21     return op;
22 }

```

```

53 int main()

```

```

54 {
55     // Get first number from user
56     int input1 = getUserInput();
57
58     // Get mathematical operation from user
59     int op = getMathematicalOperation();
60
61     // Get second number from user
62     int input2 = getUserInput();
63
64     // Calculate result and store in temporary variable (for readability/debug-ability)
65     int result = calculateResult(input1, op, input2 );
66
67     // Print result
68     printResult(result);
69 }

```

```

27 int calculateResult(int x, int op, int y)
28 {
29     // note: we use the == operator to compare two values to see if they are equal
30     // we need to use if statements here because there's no direct way to convert op into the appropriate operator
31
32     if (op == 1) // if user chose addition (#1)
33         return x + y; // execute this line
34     if (op == 2) // if user chose subtraction (#2)
35         return x - y; // execute this line
36     if (op == 3) // if user chose multiplication (#3)
37         return x * y; // execute this line
38     if (op == 4) // if user chose division (#4)
39         return x / y; // execute this line
40
41     return -1; // default "error" value in case user passed in an invalid op
42     // note: This isn't a good way to handle errors, since -1 could be returned as a legitimate value
43
44 }

```

```

< std::endl;

```

Words of advice when writing programs

- **Keep your programs simple to start.**
 - Grand vision
 - “I want to write a role-playing game with graphics and sound and random monsters and dungeons, with a town you can visit to sell the items that you find in the dungeon”
 - overwhelmed and discouraged at your lack of progress
- **Add features over time.**
 - have your simple program working and working well
 - then you can add features to it

Words of advice when writing programs

- **Focus on one area at a time**
- **Test each piece of code as you go**
 - Not write the entire program in one pass
 - the compiler reports hundreds of errors
 - not only be intimidating, if your code doesn't work, it may be hard to figure out why
 - write a piece of code, and then compile and test it immediately.
 - If it doesn't work, you'll know exactly where the problem is, and it will be easy to fix.
 - Once you are sure that the code works, move to the next piece and repeat

Too many steps & suggestions?

- following these steps will definitely save you a lot of time in the long run.
- A little planning up front saves a lot of debugging at the end.
- Good news:
 - they will start coming naturally to you without even thinking about it.
 - Eventually you will get to the point where you can write entire functions without any pre-planning at all.

Debugging your program

Syntax and semantic errors

- Errors: syntax errors, & semantic errors (logic errors).
- A **syntax error** not valid according to the grammar of the C++
 - missing semicolons
 - undeclared variables
 - mismatched parentheses or braces
 - unterminated strings

```
1  #include <iostream>; // preprocessor statements can't have a semicolon on the end
2
3  int main()
4  {
5      std::cout < "Hi there; << x; // invalid operator (:), unterminated string (missing "),
        and undeclared variable
6      return 0 // missing semicolon at end of statement
7  }
```

- compiler will generally catch syntax errors and generate warnings or errors

Syntax and semantic errors

- Errors: syntax errors, & semantic errors (logic errors).
- A **semantic error** occurs when a statement is syntactically valid, but does not do what the programmer intended.

```
1  #include <iostream>
2
3  int add(int x, int y)
4  {
5      return x - y; // function is supposed to add, but it doesn't
6  }
7
8  int main()
9  {
10     std::cout << add(5, 3); // should produce 8, but produces 2
11     return 0;
12 }
```

- compiler will not be able to catch these types of problems =>
Debugger

The debugger

- A **debugger** is a computer program that allows the programmer to control how a program executes and watch what happens as it runs.



Before proceeding: Make sure your program is set to use the [debug build configuration](#).

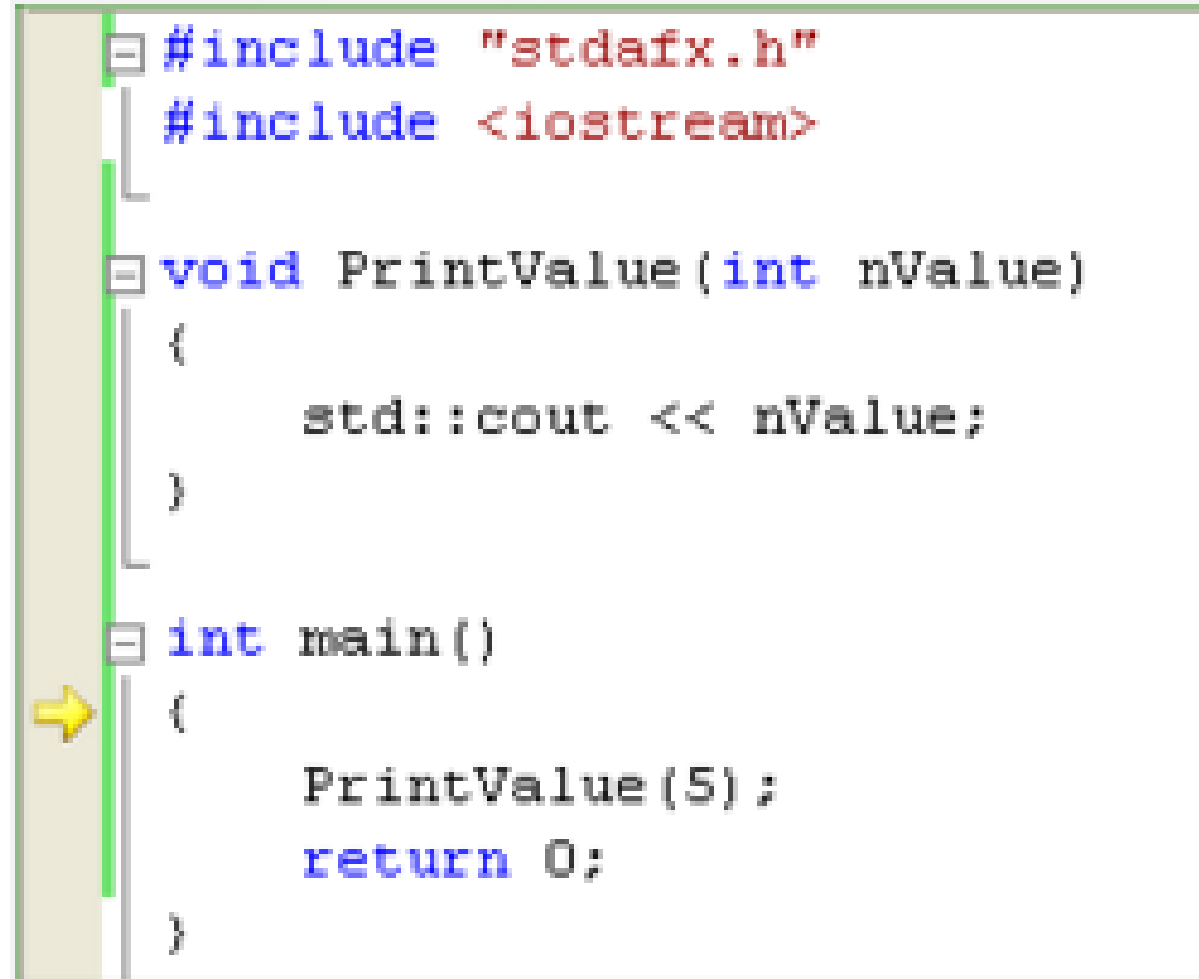


Six Stages of Debugging

1. That can't happen.
2. That doesn't happen on my machine.
3. That shouldn't happen.
4. Why does that happen?
5. Oh, I see.
6. How did that ever work?

Stepping

- execute (step through) your code line by line
- 3 different stepping commands: step into, step over, and step out
- **step into**
 - executes the next line of code
 - If this line is a function call, step into enters the function and returns control at the top of the function.
 - go to the debug menu and choose “Step Into”, or press F11.
 - arrow indicates:
 - the line being pointed to will be executed next



```
#include "stdafx.h"
#include <iostream>

void PrintValue(int nValue)
{
    std::cout << nValue;
}


int main()
{
    PrintValue(5);
    return 0;
}
```

step into

```
= #include "stdafx.h"
  #include <iostream>

= void PrintValue(int nValue)
  {
    std::cout << nValue;
  }


= int main()
  {
    PrintValue(5);
    return 0;
  }
```



```
= #include "stdafx.h"
  #include <iostream>

= void PrintValue(int nValue)
  {
    std::cout << nValue;
  }

= int main()
  {
    PrintValue(5);
    return 0;
  }
```




step into

```
▢ #include "stdafx.h"
  #include <iostream>

▢ void PrintValue(int nValue)
  {
    std::cout << nValue;
  }


▢ int main()
  {
    PrintValue(5);
    return 0;
  }
```



```
▢ #include "stdafx.h"
  #include <iostream>

▢ void PrintValue(int nValue)
  {
    std::cout << nValue;
  }


▢ int main()
  {
    PrintValue(5);
    return 0;
  }
```



```
▢ #include "stdafx.h"
  #include <iostream>

▢ void PrintValue(int nValue)
  {
    std::cout << nValue;
  }

▢ int main()
  {
    PrintValue(5);
    return 0;
  }
```



- choose “Stop Debugging” from the debug menu. This will terminate your debugging session.

step into, step over, and step out

- **Step over**

- command executes the next line of code.
- If this line is a function call, “Step over” executes all the code in the function
- and returns control to you after the function has been executed.

- **Step out**

- executes all remaining code in the function you are currently in,
- and returns control to you when the function has finished executing.

Run to cursor, Run

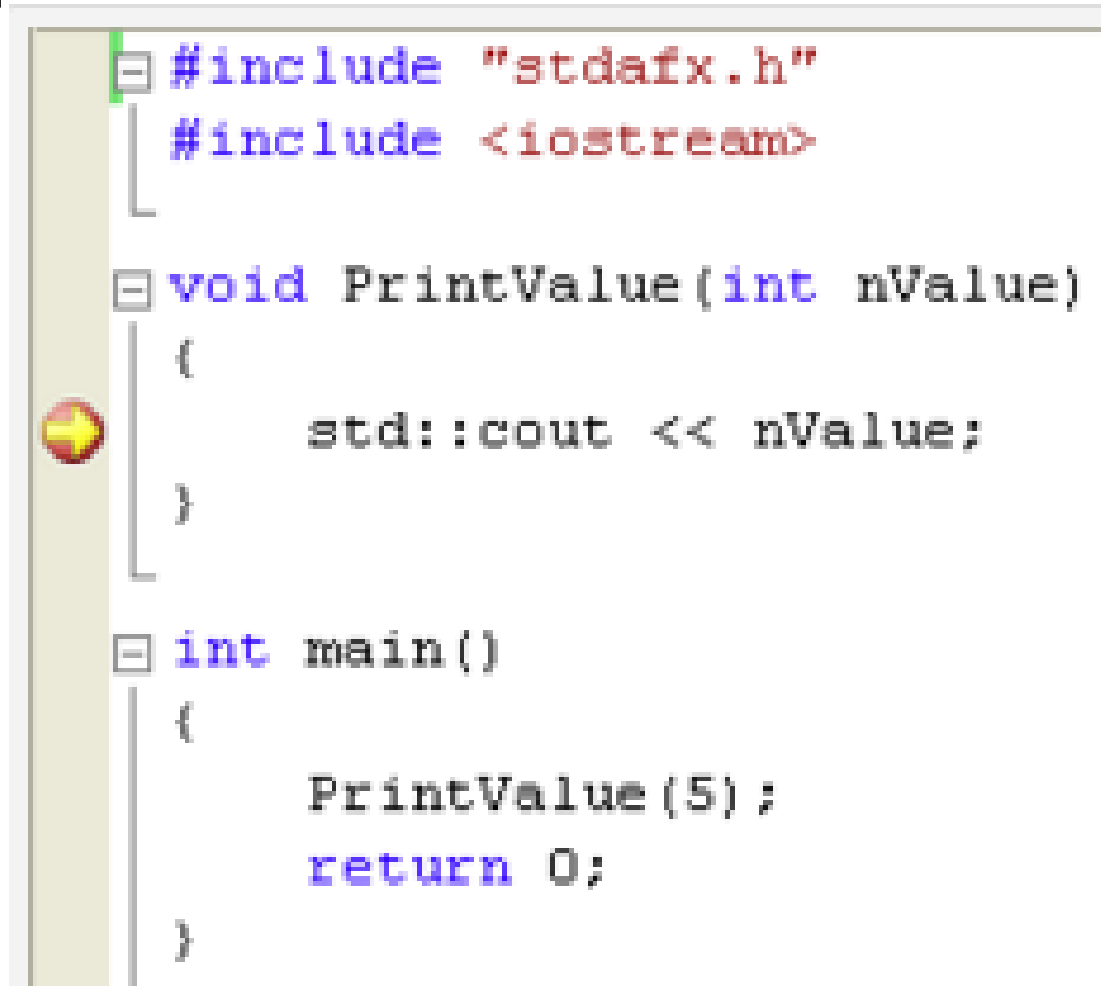
- Run to cursor: executes the program like normal until it gets to the line of code selected by your cursor.

```
1  #include <iostream>
2
3  void printValue(int nValue)
4  {
5      std::cout << nValue;
6  }
7
8  int main()
9  {
10     printValue(5);
11     return 0;
12 }
```

- Simply put your **cursor** on the `std::cout << nValue;` line inside of `printValue()`, then right click and choose “Run to cursor”.
- Run: it may be called “Go” or “Continue”

Breakpoints

- tells the debugger to stop execution of the program at the breakpoint when running in debug mode.
- “Toggle Breakpoint” (right click, choose Breakpoint -> Insert Breakpoint).



```
#include "stdafx.h"
#include <iostream>

void PrintValue(int nValue)
{
    std::cout << nValue;
}

int main()
{
    PrintValue(5);
    return 0;
}
```

The image shows a code editor window with a C++ program. A yellow arrow icon, representing a breakpoint, is positioned to the left of the line `std::cout << nValue;` inside the `PrintValue` function. The code is color-coded: keywords like `void`, `int`, and `return` are blue, and string literals and headers are red.

Debugging your program (watching variables and the call stack)

watching variables and the call stack

- stepping through a program
- examine the value of variables
- **Watching variables**
 - inspecting the value of a variable while the program is executing in debug mode

```
#include "stdafx.h"
#include <iostream>

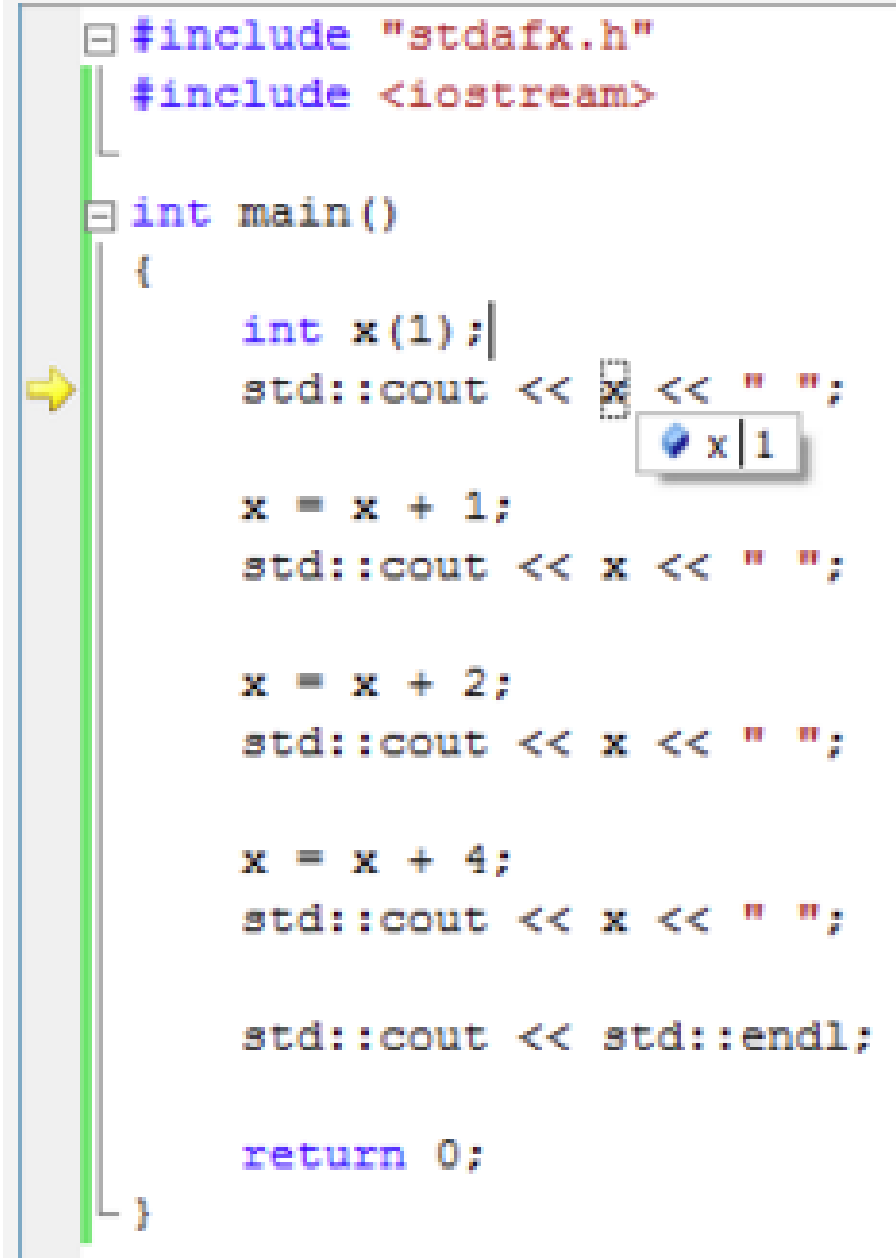
int main()
{
    int x(1);
    std::cout << x << " ";
    x = x + 1;
    std::cout << x << " ";

    x = x + 2;
    std::cout << x << " ";

    x = x + 4;
    std::cout << x << " ";

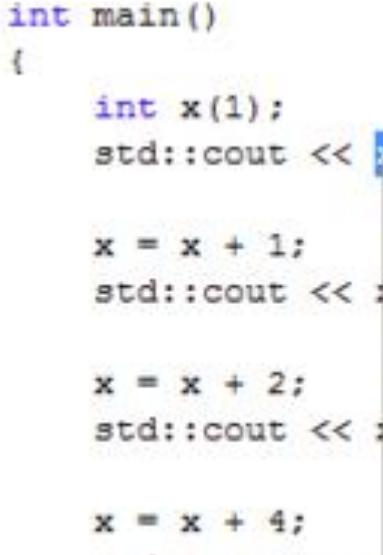
    std::cout << std::endl;

    return 0;
}
```



AddWatch

- Highlight the variable name `x` with your mouse,
- choose “AddWatch” from the right-click menu.



```
#include "stdafx.h"
#include <iostream>

int main()
{
    int x(1);
    std::cout << x << " ";

    x = x + 1;
    std::cout << x << " ";

    x = x + 2;
    std::cout << x << " ";

    x = x + 4;
    std::cout << x << " ";

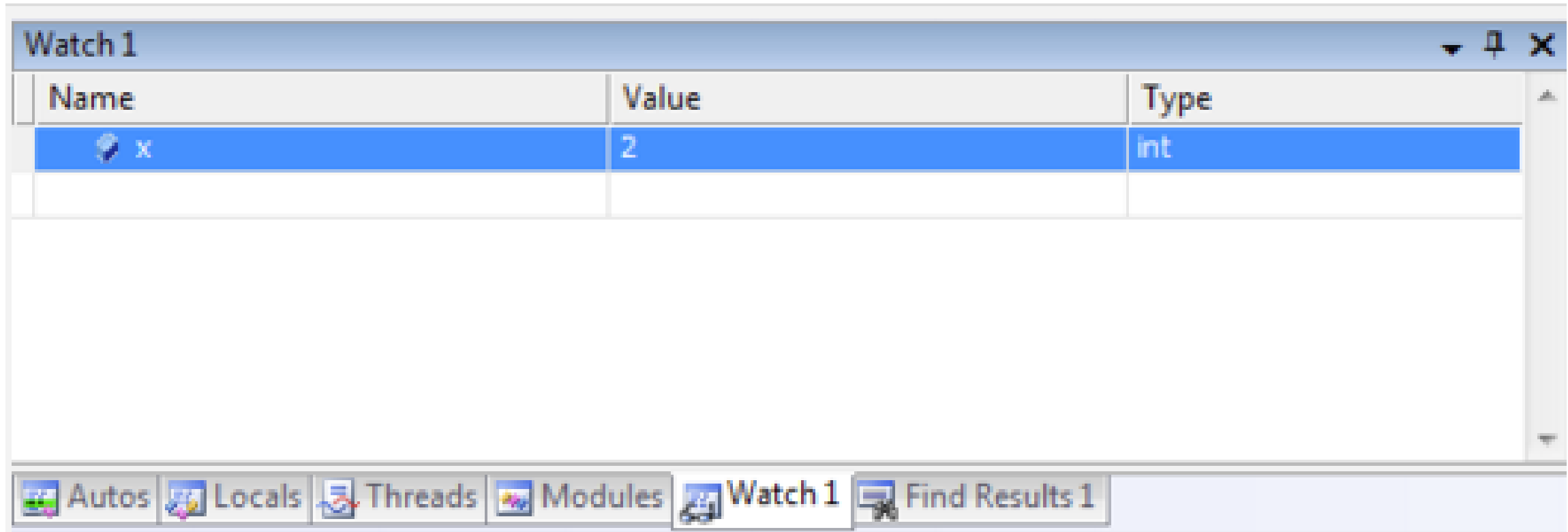
    std::cout << " ";

    return 0;
}
```

- Call Browser
- Create Unit Tests...
- Go To Definition
- Go To Declaration
- Find All References
- Go To Header File
- Breakpoint
- Add Watch
- QuickWatch...**
- Show Next Statement
- Run To Cursor
- Set Next Statement
- Go To Disassembly
- Cut
- Copy
- Paste
- Outlining


The watch window

- where you can add variables you would like to continually inspect
- and these variables will be updated as you step through your program.
- Debug Menu->Windows->Watch->Watch 1
 - note: you have to be in debug mode, so step into your program first



The call stack window

- a list of all the active functions that have been called to get to the current point of execution.
- Debug Menu->Windows->Call Stack

Call Stack	
	Name
	Test2.exe!CallC() Line 9
	Test2.exe!CallB() Line 15
	Test2.exe!CallA() Line 20
	Test2.exe!main() Line 27
	Test2.exe!__tmainCRTStartup() Line 586 + 0x19 bytes
	Test2.exe!mainCRTStartup() Line 403
	kernel32.dll!76a0338a()
	Messages below may be incorrect and/or missing, see symbols for details.

- **double-click** on the various lines

```
1  #include "stdafx.h"
2  #include <iostream>
3
4  void CallC()
5  {
6      std::cout << "C called" << std::endl;
7  }
8
9  void CallB()
10 {
11     std::cout << "B called" << std::endl;
12     CallC();
13 }
14
15 void CallA()
16 {
17     CallB();
18     CallC();
19 }
20
21 int main()
22 {
23     CallA();
24
25     return 0;
26 }
```

Conclusion

- Design, test, coding
 - Purpose
 - Divide and conquer => Functions
 - Sequence/hierarchy of events
 - Function prototype
- Debug
 - Stepping
 - Breakpoints
 - Watches
 - Call stack window
- Takes practice, trial & error
- Definitely worth your time investment!