Chapter 1

Introduction to C/C++ Programming

This book is about learning numerical programming skill and the software development process. Therefore, it requires a lot of hands-on programming exercises. In consideration that not every reader has programmed in C/C++ under Microsoft Visual Studio, we start this book with the basics of C/C++ programming. After overcoming the initial hurdles of C/C++ language syntax, readers will find that numerical programming is simply translating mathematic logic and evaluation into C/C++ programming statements.

In scientific and engineering calculations, exact symbolic results are usually very desirable when they can be found via, for example, Macsyma (originally developed at MIT) and Maple (originated at the University of Waterloo). However, it is not always possible to get symbolic results. In such cases, we must resort to numerical methods. Since most engineering and scientific computations involve evaluation of mathematical functions, they do not necessarily require *graphical user interface (GUI)* and hence can be implemented and maintained efficiently as *console applications*. A console application is a computer program designed to be used via a text-only computer interface. It accepts input and sends output to the console, which is also known as the *command prompt*. In much older operating systems, the command prompt was widely known as DOS (*Disk Operating System*) or MS-DOS prompt. Most programs in this book will be implemented as console applications. Therefore, we start this section with the creation of few simple console applications to help readers get familiar with basic C/C++ syntax and Microsoft Visual Studio development environment.

In the early days of programming, most programmers used *Emacs* (the text editor developed initially at MIT AI lab in 1970's) to implement FORTRAN, PASCAL, and C programs. When the implementation is completed, the source file is compiled at the command line to generate an executable program whose suffix is usually `.exe`. To see how this works, we will use a text editor (either `Notepad` or `Emacs`) to implement a program to compute the factorial

$$n! = n(n-1)(n-2)\cdots 1.$$

Translating the above formula into C or C++ code would be something like:

```
Result = n;
for (i=n-1; i>1; i--)
{
   Result = Result * i;
}
```

Here, we use a `for` loop to perform repeated execution. Loop index `i` is initialized to $n-1$. The statement

```
Result = Result * i;
```

is repeatedly executed if `i` is larger than 1. After each execution, the index is reduced by 1 with *post-decrement* arithmetic operator. In C and C++, there are four commonly-used *unary increment* or *decrement* operators:

Expression	Called	Explanation
++i	pre-increment	Increment i by 1, then use the new value of i in the expression in which i resides
i++	post-increment	Use the current value of i in the expression in which i resides, then increment i by 1
−−i	pre-decrement	Decrement i by 1, then use the new value of i in the expression in which i resides
i−−	post-decrement	Use the current value of i in the expression in which i resides, then decrement i by 1

If you do not want to use a unary decrement operator, it is fine to write the above code as

```
Result = n;
for (i=n-1; i>1; i -= 1)
{
   Result *= i;
}
```

It is noted that *compound assignment operator* `*=` and `-=` are used in the above code. In C/C++ programming, addition (`+=`) and division (`/=`) are also widely-used compound assignment operators.

Based on the preference, we may also replace the `for` loop by a `while` loop as

```
Result = 1;
while (n > 1)
{
   Result *= n;
   n--;
}
```

We can also combine two statements inside the `while` loop into one as

```
Result = 1;
while (n > 1)
{
   Result *= n--;
}
```

However, I personally do not recommend it since the code becomes more difficult to understand for beginners. To be able to test the implementation of factorial computation, we need a driver `main` to initialize n, call the method `Factorial`, and print out the result. The full implementation is listed below:

```
1   #include <stdio.h>
2
3   int Factorial(int n)
4   {
5      int Result = 1;
6      while (n > 1)
7      {
8         Result *= n;
9         n--;
10     }
11     return Result;
12  }
13
14  int main (void)
15  {
16     int result, n = 6;
17     result = Factorial(n);
18     printf("result = %d\n", result);
19     return 0;
20  }
```

Since this is our first console application implemented in C/C++, we shall explain the above code in detail. The C and C++ programming languages are closely related. C++ grew out of C and was designed to support *object oriented programming*. The incompatibilities between C and C++ were reduced as much as possible in order to maximize interoperability between the two languages. Examples in this book are mostly numerical algorithms that are usually implemented as functions rather than as object oriented *classes*. Therefore, it is possible to implement such algorithms so that they are compatible with both C and C++. If any code is specific to C++, it will be mentioned. Otherwise, we simply say in this book that "*the implementation in C/C++ is ...*"

Referring to the above code list, the statement `#include <stdio.h>` in line 1 is to include a standard C/C++ library header file that contains the prototypes or signatures of input/output (I/O) functions such as `printf`. Without it, we would get a compiling error about `printf` since the compiler does not know the prototype of `printf`.

Lines 3 through 12 illustrate the implementation of the computation of factorial $n!$. In particular, line 3 is a function *declaration* (also known as *prototype* or *signature*). It

precedes the function definition and specifies the name, return type, storage class, and other attributes of a function. To be a prototype, the function declaration must also establish types and identifiers for the function's arguments. In our case, the return type of this function is `int`, indicating that an integer will be returned by this function. It is also noted that this function has one argument `int n` that is sent to this function from `main` via *passing-by-value* mechanism. When an argument is passed by value, a copy of the argument is passed to the function. Modification of this argument in the callee function has no effect on the variable in the caller function. Accordingly, $n = 6$ in `main` will not be changed when the copy of `n` gets modified inside the `while` loop in the callee function.

Implementation of `main` starts from line 14 and ends at line 20. For a console application, the only absolute requirement is a function called `main`. It is the starting point of execution for all C and C++ console programs. The operating system calls `main` when we run the program, and from that point on, we implement our algorithm and can use any programming structure we want. The argument in `main` declaration is `void`, indicating that no specific input arguments are declared. In chapter 4 we shall see how we can add input arguments to `main` to access the command-line arguments. From line 16 to 18 inside the `main` function, we initialize `n`, call the function `Factorial` to compute $n!$, and print out the result at the command prompt. The `printf` function at line 18 formats and prints a series of characters and values to the standard output stream. If arguments follow the format string, the format string must contain specifications (beginning with %) that determine the output format for the arguments. At line 18, `%d` indicates its corresponding argument is a signed decimal integer. Additional frequently-used format specifiers in this book are `%lf` (long float) and `%e` (scientific notation). For example, the following code fragment

```
double pi = 3.14159265358979323846;
printf("pi = %lf or %.10lf or %.15e\n", pi, pi, pi);
```

will print π in three different formats:

```
pi = 3.141593 or 3.1415926536 or 3.141592653589793e+000
```

It is noted that the first, second, and third printout has 6, 10, and 15 decimal places. Besides `printf`, we also frequently use `fprintf` in this book. It formats and prints a series of characters and values to a user-defined stream as oppose to the standard output stream. We will cover it in subsequent chapters.

We now discuss how to compile and run the `Factorial` program. Let us create a new folder in `D:` drive and name it `ArtProgram`. It is all right if you prefer to create this folder in another drive. In this case, however, you have to make necessary changes whenever the directory path is presented. We create also a subfolder `test` and save the above code as `ComputeFactorial.cpp` in this subfolder. We then invoke Microsoft's Visual C++ compiler `cl.exe` at the command prompt as

```
D:\ArtProgram\test>cl ComputeFactorial.cpp
```

The following information will be displayed at the command line

```
Microsoft (R) 32-bit C/C++ Optimizing Compiler
    Version 16.00.40219.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ComputeFactorial.cpp
Microsoft (R) Incremental Linker Version 10.00.40219.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:ComputeFactorial.exe
ComputeFactorial.obj
```

We may suppress the display of the above copyright information by using `/nologo` options as follows:

```
D:\ArtProgram\test>cl /nologo ComputeFactorial.cpp
```

Upon the completion of compilation, we should see `ComputeFactorial.exe` in the `test` folder and may invoke this executable to generate the desired output at the command line as

```
D:\ArtProgram\test>ComputeFactorial
result = 720
```

The above development process is fine if we just do some simple and casual programming. Even so, we may still have a problem in debugging the code when a program does not generate the expected results. In this case, we may have to detect bugs by putting print instructions inside our programs to print out intermediate results. By doing so, we may be able to isolate the problematic code and eventually track down the bugs or problems.

In software companies nowadays, professional software developers usually reply on an *integrated development environments (IDEs)* to develop their programs. IDEs normally consist of a source code editor, source code management, build automation tools, and a debugger. They are designed to maximize programmers' productivity by providing tight-knit components with similar user interfaces.

In developing windows-based applications, Microsoft Visual Studio is unarguably the primary choice of IDEs. For this reason, Microsoft Visual studio is the chosen integrated development environment for this book. At the write of this book, Microsoft Visual Studio 2012 was just released. In consideration that many readers may not have upgraded their versions, we use Microsoft Visual Studio 2010 exclusively in this book. If you do not have Microsoft Visual Studio installed in your computer, you may download Microsoft Visual Studio Express from the Microsoft web site. The Express version is a lightweight freeware that has almost everything we need except for the support of parallel programming via OpenMP.

We now look how to use Visual Studio IDE to develop a console application. Start Visual Studio 2010 and click `File->New->Project` from Visual Studio *menu bar* to bring up `New Project` wizard. The Visual Studio menu bar is also known as IDE menu bar. It contains menu categories such as `File`, `Edit`, `View`, $\cdots$, `Window`, and `Help` as shown below. This IDE menu bar will be referred to frequently in subsequent chapters.

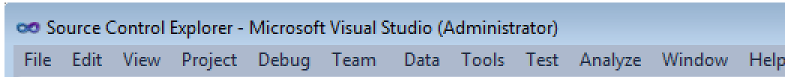

Figure 1.1: IDE menu bar

In the Visual C++ project types pane, click `Win32 Console Application` and type or browse the location of the project. In our case, the location is the folder in which `ComputeFactorial.cpp` resides. We then type the project name `ComputeFactorial` as shown below and click the `OK` button.

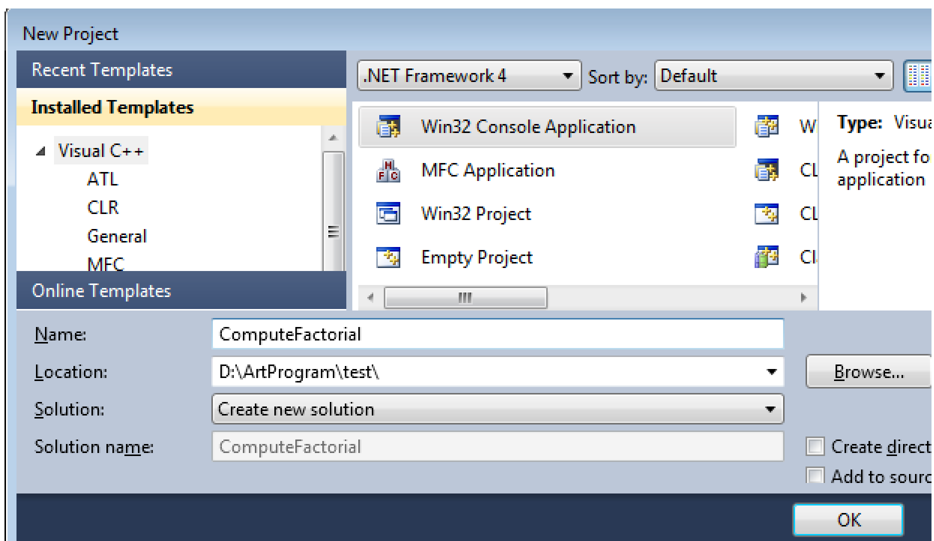

Figure 1.2: Create a Visual C++ project

When the Win32 Application Wizard pops up, we click `Next`, then select the `Empty Project` check box and click the `Finish` button. The wizard will create several empty folders under `ComputeFactorial` project in *Solution Explorer* as seen in figure 1.3. If Solution Explorer is not visible, click Solution Explorer on the View menu.

Let's move the mouse cursor to `Source Files` folder and click the right mouse button to select `Add->Existing Item` as shown in figure 1.3. Browse to find the existing source file `ComputeFactorial.cpp` and click the `Add` button. We should now find that the source file has been added to the `Source Files` folder as illustrated in figure 1.4. If a `.cpp` source file is created from scratch by using Microsoft Visual Studio IDE text editor, we need to select `Add->New Item`. In this case, the Wizard will ask whether it is a `.cpp`, or `.h`, or any other type of file. By selecting a correct file type and entering the name of file, the Wizard will add the specified file to the `Source Files` folder in Solution Explorer and a tabbed window will appear where we type in the code.

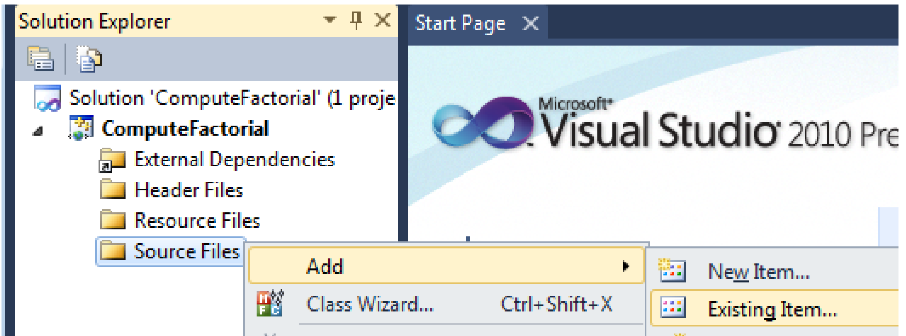

Figure 1.3: Add existing files to source folder

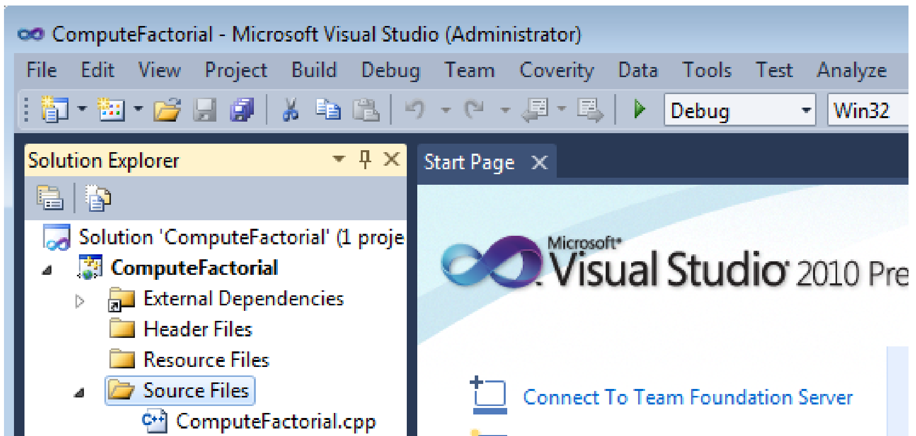

Figure 1.4: Appearance of Visual C++ project (build mode is `Debug`)

We are ready to compile and run the program under Visual Studio IDE. From the menu bar, click `Build->Build Solution` from the menu bar or press the `F7` key to build the project. By default it is set to `Debug` mode as shown in the Solution Configurations list box in figure 1.4. Compiling the program with `Debug` setting generates a debug version of executable with full symbolic debug information so that we can set a *breakpoint* at any statement to watch the execution. The use of breakpoints is best illustrated with an example. Let us move the mouse cursor to this statement:

```
result = Factorial(n);
```

and press `F9` to set a breakpoint that is indicated by the red solid spherical glyph as shown in figure 1.5. Next, press the `F5` key to run the program. The execution halts and breaks into the debugger when it hits the breakpoint. We can now use a `Watch` window to watch or evaluate variables and expressions. To open the `Watch` window, the debugger must be running or in break mode. From the `Debug` menu, select `Windows`, then `Watch` and click either `Watch 1`, `Watch 2`, `Watch 3`, or `Watch 4`. For this simple case, we just need one `Watch` window. Let's add `n` and `result` in `Watch 1` and press

F10 to execute the program. As we do so, the values of variables in Watch 1 window change as shown in figure 1.5.

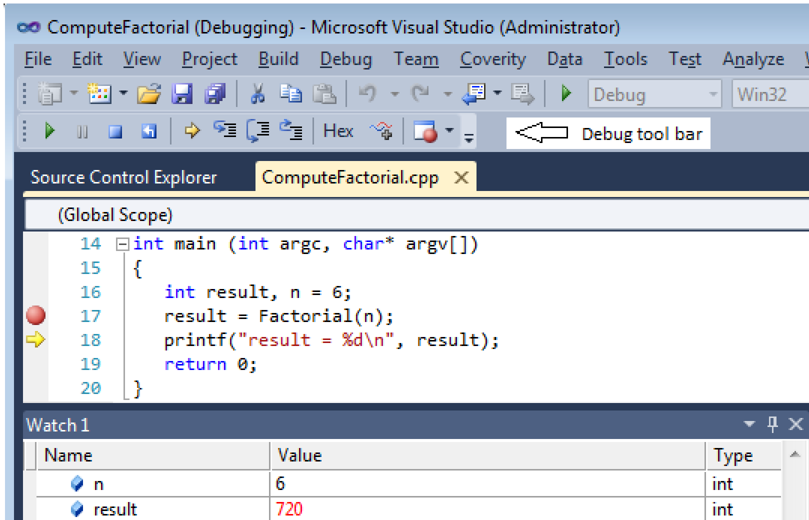

Figure 1.5: Watch variables in debug mode

As opposed to pressing F10 to step over the subfunction Factorial(n) when the execution halts at the breakpoint, we may also press the F11 key to step into Factorial to watch how the value of Result changes at each iteration.

It is noted that the Debug Toolbar is shown in figure 1.5. We can access the debugger's most commonly-used commands and windows through the Debug Toolbar. To display the Debug Toolbar, on the View menu click Toolbars and then select Debug. It is also noted that the line numbers are displayed in code for reference. If they are not shown in your Microsoft Visual Studio IDE, you can turn on the display of line numbers by choosing Tool->Option from menu bar, then expanding the Text Editor node and selecting either All Languages or the specific language (e.g., C/C++), and marking the Line numbers checkbox.

A breakpoint can be set and deselected by pressing F9 key. We can also press Ctrl+F9 to disable and enable a breakpoint instead of deselecting it. In this case, the disabled breakpoint is marked by a hollow glyph. It is also possible to set a *conditional breakpoint*. Conditional breakpoints can be very useful when we are trying to find bugs in our code. They cause a break to occur only if a specific condition is satisfied. A conditional breakpoint is created by combining a breakpoint command with a valid expression and is best illustrated with an example. Let us set a breakpoint at line 11. In a source, right-click a line containing a breakpoint glyph and choose Condition from Breakpoints in the shortcut menu as shown below.

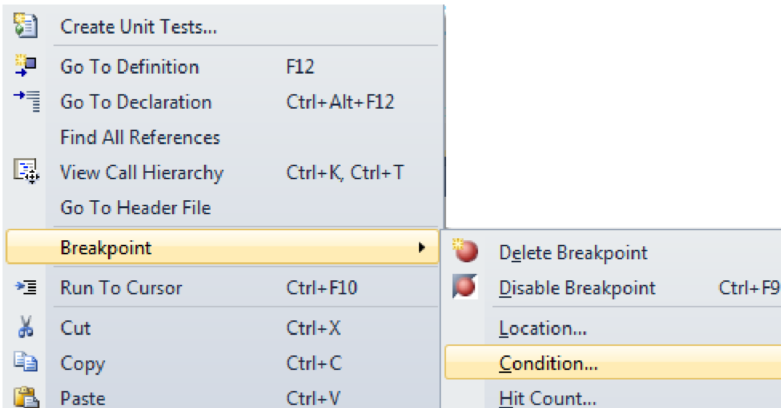

Figure 1.6: Conditional breakpoint

In the `Breakpoint Condition` dialog box, enter `n < 5` in the `Condition box` as illustrated below. A conditional breakpoint is indicated by a solid glyph with + sign in the middle. When `n` is reduced to 4 or less in the `while` loop, the execution halts at the conditional breakpoint.

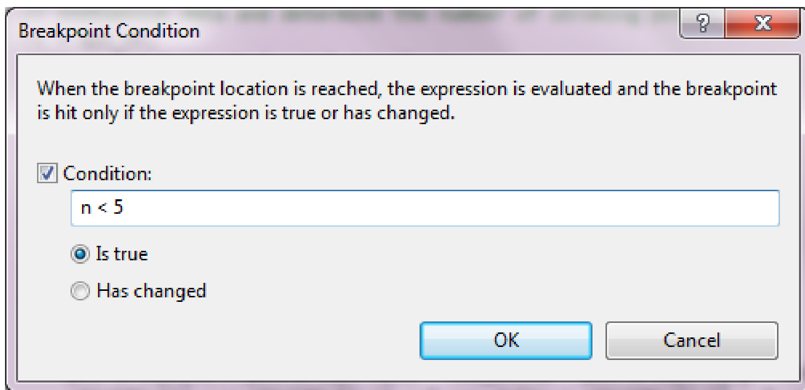

Figure 1.7: Specify a valid break condition

So far we have been running the program in debug mode. On the Standard toolbar, we can also choose `Release` from the `Solution Configurations` list box so that the compiler will generate an optimized executable that runs much faster. This is usually the final version delivered to customers. It should be pointed that you can debug the code in release mode to investigate a problem specific to the release code (e.g., local variables not explicitly initialized by the program are initialized differently in debug and release modes). However, you may find some strange behaviors such as the data in `Watch` windows not making any sense at all. It is possible to turn off optimization and enable debugging of a release build. However, the behavior under such configuration may be different from the fully optimized version.

Our next example demonstrates how to find the smallest and largest integer number in a one-dimensional array. It is designed to help readers without C/C++ programming experience become familiar with additional argument passing mechanisms and C/C++ pointers. Start Visual Studio to create an empty project named `FindMinMax` in `test` folder (referring to figure 1.2). We then select `Add->New Item` (referring to figure 1.3) to create a `apiFindMinMax.cpp` file with the following implementation

```
1   #include <stdio.h>
2
3   void apiFindMinMax(int n, int *myArray, int nMin, int nMax)
4   {
5      int   i;
6      nMin = nMax = myArray[0];
7      for (i=1; i<n; i++)
8      {
9         if (nMin > myArray[i])
10           nMin = myArray[i];
11        else if (nMax < myArray[i])
12           nMax = myArray[i];
13     }
14     return;
15  }
16
17  int main(void)
18  {
19     int   nMin = 0, nMax = 0, n = 10;
20     int   myArray[10] = {6,7,3,10,22,7,11,18,20,19};
21
22     apiFindMinMax(n, myArray, nMin, nMax);
23     return 0;
24  }
```

Referring to line 3, the return type of function `apiFindMinMax` is `void`. If a function was declared with return type `void`, the `return` statement at the end of this function should be either omitted or left there without any return value. There are four arguments in this function: `n` specifies the size of the array, `myArray` is sent to this function via *passing-by-pointer mechanism* (more discussion later), `nMin` and `nMax` are intended to be the output arguments that store respectively the smallest and largest integer value. Save and compile this project. Set a breakpoint at line 14 and press `F5` to run the program. The execution will halt at the breakpoint. By adding `nMin` and `nMax` to a `Watch` window we will see that they have correct results: 3 and 22. Press the `F10` key to step over and return to the calling function. At this moment we shall see that both `nMin` and `nMax` at the `Watch` window are back to zero – the computed results are gone! This is because both `nMin` and `nMax` were passed to `apiFindMinMax` via passing-by-value, which means copies of `nMin` and `nMax` were created for `apiFindMinMax`. Changes to these local copies have no effect on `nMin` and `nMax` in `main` and these copies were destroyed when `apiFindMinMax` ends.

One solution to this problem is to use the passing-by-pointer mechanism. *Pointers* are a very powerful feature of the C/C++ languages that has many uses in advanced programming. However, they are among C/C++'s most difficult capabilities to master. A very intimate relationship exists between arrays and pointers, which is why an array was used in the above program for discussion of pointers. Referring to line 20 of the above program, a static integer array of size ten was declared and initialized. When this program is compiled and executed, a block of memory is reserved and initialized to the specified values. Memory addresses and corresponding values can be viewed when the execution breaks into the debugger. Let's set a breakpoint at line 22 and run the program by pressing `F5`. When the execution halts at the breakpoint, we add `myArray` in the `Watch` window and click `Debug->Windows->Memory->Memory 1` from the main menu bar. When `Memory 1` window pops up, type the address of `myArray` in the address field as shown in figure 1.8. As is seen, each integer value is stored in a uniquely addressed memory "cell" that is intended for integer. Since an integer in 32bit machine has 4 bytes, the memory address from one cell to the next one is increased by 4, which can be verified by using hexadecimal number arithmetic. It should be pointed out that figure 1.8 is for illustration only. The actual memory addresses vary at each execution.

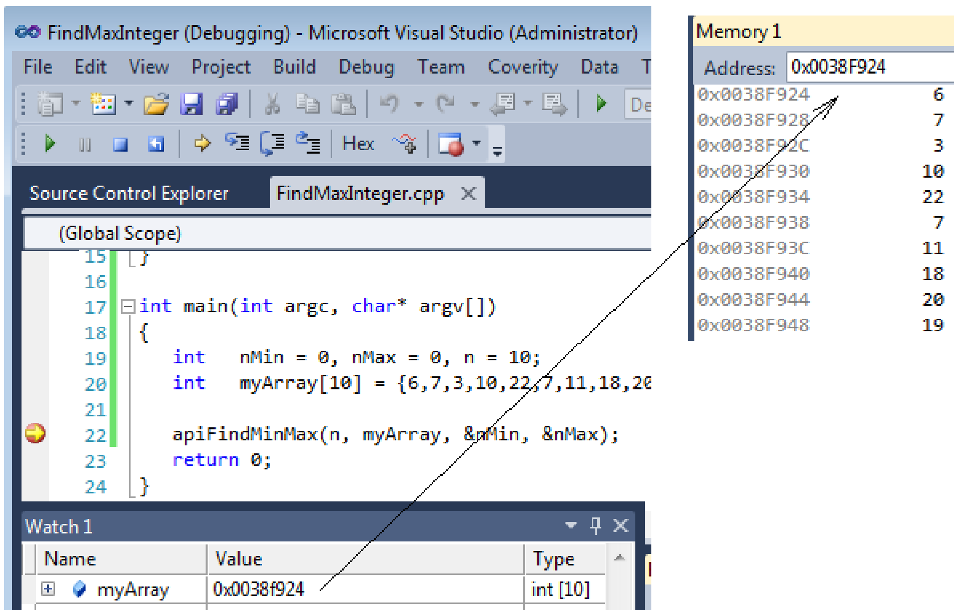

Figure 1.8: Watch memory addresses and corresponding contents

C and C++ are the programming languages that include features of both high-level and low-level programming languages. For example, they allow direct access to memory address locations via pointers. A pointer is a special variable whose value is the *address* of a memory location where a specific data type is stored. When a pointer variable is declared, the data type it can point to is also specified. The notations

```
int     *pA;
double  *pD;
```

declare pA as an integer-pointer variable and pD as a double-pointer variable. A pointer declaration merely creates the pointer variables; it neither initializes the pointer nor allocates memory space for the pointer. Therefore, before it is used, a pointer variable must be assigned the address of a variable, of an array, or of dynamically allocated memory space. Two unary operators are important in dealing with pointers:

& – the address-of operator,

* – the value-of operator.

Having already declared *pA, the statement

```
pA = &myArray[2];
```

assigns to pA the address of myArray[2]. Since the array index starts from 0 in C/C++, pA holds the address of the third memory cell, i.e., 0x0038F92C (referring to figure 1.8). To access the content of the pointer, we use the value-of operator as *pA. It was said that pA points at the third cell (or holds the address of third cell), the content of pA is thus 3. If we print *pA via

```
printf("%d\n", *pA);
```

we should see the value 3 at the command prompt. With the use of a pointer, we may pass to the callee function the address of a variable defined inside the main. By modifying the content of this variable through the pointer, the callee function is permitted to modify the actual argument declared in caller function. Let us modify our code to pass the addresses of nMin and nMax to the callee so that apiFindMinMax can indirectly modify the contents of nMin and nMax that are declared in main:

```
1   #include <stdio.h>
2
3   void apiFindMinMax(int n, int *myArray, int *nMin, int *nMax)
4   {
5      int   i;
6      *nMin = *nMax = myArray[0];
7      for (i=1; i<n; i++)
8      {
9         if (*nMin > myArray[i])
10           *nMin = myArray[i];
11        else if (*nMax < myArray[i])
12           *nMax = myArray[i];
13     }
14     return;
15  }
16
17  int main(void)
18  {
19     int   nMin = 0, nMax = 0, n = 10;
```

```
20      int   myArray[10] = {6,7,3,10,22,7,11,18,20,19};
21
22      apiFindMinMax(n, myArray, &nMin, &nMax);
23      return 0;
24   }
```

Pointers are powerful and efficient in that C/C++ allows pointer arithmetic. Arithmetic on pointers takes into account the size of the type. For example, adding an integer number to a pointer produces another pointer that points to an address that is higher by that number times the size of the type. Therefore, `pA += 2` is equivalent to assigning (referring to figure 1.8)

```
0x0038F934 (= 0x0038F92C + 2 * sizeof(int) = 0x0038F92C + 8)
```

to `pA`. Accordingly, `pA += 2` changed the address of `pA` from the current `&myArray[2]` to `&myArray[4]`. With basic knowledge about pointer arithmetic, we may potentially improve the efficiency of code by avoiding array indexing as follows:

```
1   void apiFindMinMax(int n, int *myArray, int *nMin, int *nMax)
2   {
3      int   i, *ptr;
4      *nMin = *nMax = myArray[0];
5      ptr = myArray + 1;
6      for (i=1; i<n; i++)
7      {
8         if (*nMin > *ptr)
9            *nMin = *ptr;
10        else if (*nMax < *ptr)
11           *nMax = *ptr;
12        ptr++;
13     }
14  }
```

However, I personally do not encourage this for beginners. In addition, modern compilers are likely to generate the same code for pointer accesses and array accesses. Therefore, we should probably not be worrying about that level of performance.

In C++ (not in C), it is also possible to pass the address of a variable to the callee via *passing-by-reference*. This is done by placing `&` in front of an argument in the decoration of function. Passing an argument by reference means that the function or method is permitted to modify the actual argument. For illustration we modify our implementation of `apiFindMinMax` as follows:

```
1   #include <stdio.h>
2
3   void apiFindMinMax(int n, int *myArray, int &nMin, int &nMax)
4   {
5      int   i;
6      nMin = nMax = myArray[0];
7      for (i=1; i<n; i++)
```

```
8      {
9         if (nMin > myArray[i])
10           nMin = myArray[i];
11        else if (nMax < myArray[i])
12           nMax = myArray[i];
13     }
14     return;
15  }
16
17  int main(void)
18  {
19     int   nMin = 0, nMax = 0, n = 10;
20     int   myArray[10] = {6,7,3,10,22,7,11,18,20,19};
21
22     apiFindMinMax(n, myArray, nMin, nMax);
23     return 0;
24  }
```

For now, this is adequate information to allow us to move on and be comfortable with implementing algorithms that will be discussed in the next two chapters. When encountering new terminologies and new Microsoft IDE environments, I shall continue to provide necessary explanations.