C/C++ Day 3

윤광진, 광주과학기술원 02/07/2018

1 Debugging with VS2017

Visual studio는 매우 편리하고 강력한 디버깅 환경을 제공해 준다. 디버깅이란 작성한 프로그램이 오류가 있는지 정확성을 테스트 하거나, 프로그램이 정상적으로 작동하지 않을때 오류(버그, 메모리 충돌 등)를 찾아내는 과정들을 말한다. 때로는 프로그램이 원하는 성능(연산 속도, 메모리 사용율 등)이 나오지 않을 경우 성능 최적화를 하기 위해서도 디버깅을 할 수 있다. 디버깅을 할때 주로 사용되는 방법은 코드를 단계적으로 재개하거나, 메모리에 저장된 값을 살펴보는 방법이 이용된다. 여기서는 프로그램에 오류가 있을때 그것을 찾아내는 방법과 Visual Studio 디버거를 사용하는 방법에 대해서 알아본다.

우선 2일차에 사용했던 **`Matrix`** 클래스 예제의 **`main`** 함수를 Listing 1에 옮겨 놓았다.

Listing 1: Debugging

```
1  #include <iostream>
2  #include "Matrix.h"
3  using namespace std;
4
5  void main(int argc, char* argv[])
6  {
7      if (argc != 3)
8      {
9          cout << "err: Must input two arguments\n";
10         return;
11     }
12     char* program_name = argv[0];
13     cout << "This program is '" << program_name << "'." << endl;
14     int in1 = atoi(argv[1]); // first argument
15     int in2 = atoi(argv[2]); // second argument
16     Matrix mat(in1, in2);
17     printf("Number of rows: %d\n", mat.getRows());
18     printf("Number of columns: %d\n", mat.getCols());
19 }
```

Visual studio를 이용해 코드 중단점을 쉽게 설정 할 수 있다. 원하는 코드로 커서를 위치 시킨후 **`F9`**를 누르거나 코드의 줄번호 앞을 클릭하면 중단점(빨간색 동그라미)이 설정된다. 이 중단점의 의미는 코드를 순차적으로 실행하다가 중단점이 위치한 코드에 도달하면 그 코드가 실행되기전 바로 멈추라는 뜻이다. 그럼 7번 줄에 중단점을 위치시키고 코드를 실행해보자. 코드를 실행 할때는 **`F5`**를 눌러 실행시키거나 **Debug** → **Start Debugging**을 클릭한다. 코드가 실행되면 빨간 동그라미

안에 노란색 화살표가 표시된 것이 보이고, 실행이 그 위치에서 멈춰진다. 아직 7번줄은 실행된 것이 아니다. 그럼 **argc**변수위에 마우스를 올려보자. 2일차 강의대로 따라왔다면 **argc**에 3이 들어있을것이다[1]. 그럼 7번줄의 비교 연산의 결과는 **false** 이므로 8-11번 줄을 건너띄고 12번 줄로 이동할 것이다. 코드를 한단계씩 진행하려면 **step over**(**F10**)를 사용하면된다. **F10**을 눌러 코드를 한단계 진행시켜서 12번줄로 이동하는지 확인해보자. 노란색 화살표가 12번줄로 이동한다. 12번줄은 포인터 변수인 **argv** 배열의 첫번째 원소를 **program_name** 지역변수에 복사를 하는 작업을 한다. **argv[0]**에는 프로그램 실행시 첫번째 인자(실행파일 이름)이 들어있다. **F10**을 눌러 코드를 한단계 더 진행시키고 **program_name**에 마우스를 올려 값이 제대로 복사되었는지 확인하자. 변수들의 내용을 확인하는 방법은 마우스를 변수위에 올리려서 확인하는 방법도 있지만, Visual studio 하단의 Debug window를 이용해 확인 할 수도 있다[2]. 아직 코드가 실행되지 않은 변수들에는 쓰레기 값(의미없는 값)들이 들어 있음을 확인하자. 디버깅중에 다시 코드를 실행하려면 **F5**를 누르면된다. 그러면 해당위치부터 코드가 실행된다. 프로그램을 실행할때 **F10** 또는 **F11**을 눌러 실행하면 프로그램의 시작점 위치에서부터 디버깅을 할 수 있다. **F10**을 눌러 디버깅을 시작하고 다시 7번줄까지 화살표를 이동시켜보자. 이번에는 **argc**의 내용을 변경해보겠다. 하단의 Local window(로컬 창)에서 argc를 찾아 값을 3이 아닌 정수로 변경해보자. 그리고 **F10**을 눌러 코드를 한단계 진행시키면 아까와 다르게 9번줄로 이동하는 것을 볼 수 있다. 7번줄의 비교 연산의 결과가 **참**이기 때문이다. 코드를 계속 진행시키면 에러 메세지를 출력하고 종료하게 된다.

이번에는 함수 내부로 진입할 수 있는 **step into**(**F11**)에대해 알아보자. 위 예제에서 16번줄에 중단점을 설정해 보자. 16번줄은 **Matrix** 클래스를 **mat**라는 이름으로 생성하는 부분으로 생성자를 호출하는 부분이다 (생성자도 함수다). **F5**로 디버깅을 시작하고 코드가 16번줄에서 멈추면 **F11**로 함수 내부로 진입하자. **Matrix(int _r, int _b)** 생성자 안으로 진입을 한다. 함수 내부의 **parameter** 값들을 확인해보고 디버깅을 종료하자.

2 메모리

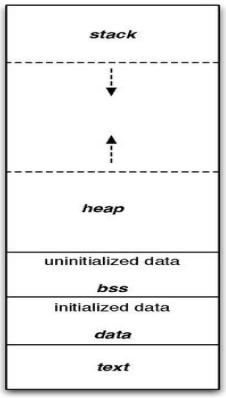

Figure 1: 프로그램의 메모리 레이아웃

이번 절에서는 프로그램의 메모리 구조에 대해서 설명한다[3]. C/C++ 프로그램의 전형적인

[1] 입력 인자의 갯수가 3이란 뜻이다. 실행 파일 명, 행렬의 행수, 행렬의 열수 이렇게 세 개를 입력했기 때문이다.
[2] Auto window: **Debug** → **Windows** → **Autos**, Local window: **Debug** → **Windows** → **Locals**
[3] 참고자료: https://www.geeksforgeeks.org/memory-layout-of-c-program/, https://en.wikipedia.org/wiki/Data_segment

메모리 구조는 **`Text segment, Iniitalized data segment, Uninitialized data segment, Stack, Heap`**으로 구성 되어있다 (그림 1).

2.1 Text Segment

텍스트 세그먼트는(=텍스트 영역, 코드 세그먼트, 코드 영역)는 프로그램 코드가 위치하는 영역이다. 이 영역에는 프로그램이 실행해야하는 명령들이 위치해 있다. 일반적으로 Heap과 Stack 영역보다 낮은 곳에 위치해 있으며, Heap 또는 Stack 영역의 overflow 현상이 코드 영역을 침범하는 것을 막기 위해서 이다.

2.2 Initialized Data Segment

Data Segment(데이터 영역)이라고도 불린다. 이 영역에는 프로그램의 전역변수, 정적변수(static variable)들이 위치한다. 이 변수들은 프로그램 작성 단계 혹은 컴파일 단계에서 초기화가 되기 때문에 Initialized Data Segment라고 불린다. 이 영역의 변수는 read-only 와 read-write 영역으로 한번 더 세분화 된다. 예를 들어 전역변수 **`const char* str = "hello world"`**를 보자. **`str`** 이 가지고있는 데이터 'hello world'는 데이터 영역의 read-only에 저장될것이고 포인터 변수 **`char* str`**은 read-write 영역에 저장된다 [4].

2.3 Uninitialized Data Segment

BSS 영역이라고도 불리며 데이터 영역 바로 다음에 이어진다 (Block Started by Symbol의 약어). 이 영역에는 소스코드에서 초기화 되지 않은 전역 변수와 정적 변수들이 포함된다. 그리고 프로그램이 실행될때 이 변수들을 '0'으로 초기화 한다. 예를들어 정적변수 **`static int i;`** 또는 전역변수 **`int j;`** 모드 BSS에 포함되고 프로그램이 실행될때 이 변수들의 값을 '0'으로 초기화한다.

2.4 Heap

Heap Segment(힙 영역)은 동적 메모리 할당이 일어나는 곳이다 (프로그램이 실행되면서 동적으로 메모리가 할당되고 해제된다). 힙 영역은 BSS 영역에 이어서 위치하고 그곳에서부터 주소를 높여가면서 메모리를 사용한다 (그림 1). 힙 영역은 **`malloc, calloc, new, free, delete`** 등을 이용하여 관리된다.

2.5 Stack

Stack(스택 영역)은 힙 영역의 반대 방향으로 메모리를 사용한다 (그림 1). 만약 스택과 힙의 메모리 주소가 서로 만난게 된다면 프로그램 메모리의 빈 공간은 없게 된다. 스택은 LIFO (Last In First Out) 메모리 구조를 갖는다. 만약 함수 A가 함수 B를 호출한다면 A의 변수 상태 및 주소 등을 스택에 저장하고 B를 불러 그 스택위에 올린다. 그리고 함수 B가 실행을 종료하면 완료된 데이터를 함수 A에 반환하고 다시 A를 수행하는 방식이다. 재귀함수(recursive function)를 사용한다면 함수가 자기자신을 이러한 방식으로 스택에 계속 쌓는다.

[4] 이것을 이해하려면 const 키워드에 대한 공부가 필요함.

3 포인터와 배열

C++ 프로그램에서 컴퓨터의 메모리는 8bit(byte)의 크기로 고유의 메모리 주소를 가지고 있다[5]. 또한 인접한 메모리의 주소는 연속이다. 즉 메모리 주소 1778은 메모리 주소 1777 바로 다음에 나타나며, 100번 건너뛴 메모리의 위치는 1878이다.

사용자가 정의한 변수는 컴퓨터 메모리의 어딘가에 할당되고 변수의 이름(식별자, identifier)을 통해서 변수에 접근해 데이터를 불러오거나 변경할 수 있다. 하지만 변수가 저장될 메모리의 위치는 사용자가 변수를 선언할때 이름을 지어준것과는 달리 프로그램이 실행될때 운영체제에 의해 결정된다. 이때 포인터 연산자들(**`*`, `&`**)은 변수들이 저장된 메모리 위치를 이용할 수 있게 해준다.

3.1 배열

Listing 2에 배열의 사용법에 대해 예제를 보였다.

Listing 2: Pointer

```
1  #include <stdio.h>
2  void main()
3  {
4      int arr[3];
5      for (int i = 0; i < 3; ++i)
6          arr[i] = i+1;
7
8      printf("%d, %d, %d\n", arr[0], arr[1], arr[2]);
9
10     int arr2[3] = { 1, 2, 3 };
11     int arr3[] = { 1, 2, 3, 4 };
12 }
```

4번줄에 **`int arr[3]`** 으로 정수형 배열 3개를 정의했다. 하지만 초기화는 하지 않아서 어떤 값이 들어 있을지 알수는 없다. **`arr`**의 초기화는 5-6번줄에서 해주고 있다. 여기서 눈여겨 볼것은 배열의 첨자가 0부터 시작한다는 점이다. 따라서 **`arr`**의 첨자는 0에서 시작해서 2까지이다. 8번줄에서 배열의 내용을 출력하고 있다[6]. 4-6번줄처럼 배열을 정의하고 초기화를 할 수도 있지만 10번 11번 줄처럼 초기화 할 수도 있다. 10번 줄은 배열을 정의할때 배열의 원소를 중괄호 안에 지정해 주는 형태다. 이때 배열의 크기보다 많은 원소를 입력하면 Visual Studio에서 에러를 출력한다[7]. 11번 줄은 배열의 크기를 제시하지 않고 배열을 정의 및 초기화하는 형태다. 이 배열의 크기는 프로그램이 컴파일 되는 순간에 우측의 배열 원소의 갯수를 세면서 정해진다. 즉 **`arr3`**는 4개의 정수형 배열이 되게 된다.

3.2 주소 연산자

변수의 주소는 변수의 이름 앞에 주소 연산자(**`&`**, ampersand)를 써서 알아 낼수 있다.

[5] 1 바이트가 넘는 데이터는 필요한 크기 만큼 메모리 공간을 연속해서 차지한다

[6] **`arr[3]`**을 출력하도록 시도하면 운좋게 에러없이 실행 될 수 있다. 하지만 이것은 코드 작성자의 엄연한 실수다. 세개짜리 변수의 네번째 원소는 있을 수 없다.

[7] Visual Studio의 장점이다.

Listing 3: Pointer

```
1  foo = &myvar;
```

myvar의 주소값이 **foo**에 할당된다 (Listing 3). 변수가 할당받은 실제 주소는 프로그램이 실행되기 전에는 알기 힘들지만 **myvar** 변수가 할당받은 주소 값이 1776이라고 해보자. 그리고 다음 Listing 4을 보자.

Listing 4: Pointer

```
1  myvar = 25;
2  foo = &myvar;
3  var = myvar;
```

이것을 그림으로 나타내면 그림 2과 같다[8].

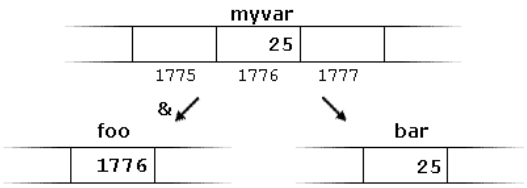

Figure 2: 주소 연산자

첫째줄은 **myvar**에 25을 대입을 하고있고, 가정한대로 **myvar**가 할당받은 주소값은 1776이다. 둘째줄은 **foo**에 **myvar**의 주소값 1776을 복사하고 있다. 그리고 세번째 줄은 **bar**라는 또 다른 이름의 변수를 만들어 **myvar**가 가지고 있는 값을 복사하는 일반적인 연산을 하고 있다. 두 번째 줄의 **foo**처럼의 다른 변수의 메모리 주소를 저장하려면 **포인터**(pointer)라고 불리는 변수를 이용해야한다.

3.3 *

***** 는 포인터다. 이 연산자는 쓰이는 위치에 따라서 포인터 연산자가 되기도하고 포인터 선언자가 되기도 한다.

3.3.1 포인터 선언자

포인터 변수는 다른 변수의 주소값을 저장 할수 있는 변수다. 이 변수를 선언하려면 포인터 변수 선언자를 사용해야한다. 포인터 선언자(*****)는 변수를 선언할때 타입명 다음, 변수명 전에 온다 (Listing 5).

Listing 5: Pointer

```
1  type * name;
```

예를 들어 Listing 6과 같이 하면 된다.

Listing 6: Pointer

```
1  int * intptr;
```

[8] http://www.cplusplus.com/doc/tutorial/pointers/

```
2 int * charptr;
3 double * dbptr;
```

3.3.2 포인터 연산자

포인터 변수는 다른 변수의 주소값을 가지고 있다. 포인터 변수가 가지고 있는 주소값을 참조해 해당 위치에 접근 할수 있는데, 이때 포인터 연산자(*)를 사용한다. 포인터 연산자를 이용해서 주소 위치의 데이터를 읽거나 변경할 수 있다. 포인터 연산자는 포인터 변수 앞에 위치한다. 다시 Listing 4 예제로 돌아가자. **foo**가 가리키고있는 주소의 데이터를 **baz** 라는 변수에 저장해보자 (Listing 9).

Listing 7: Pointer

```
1 baz = *foo;
```

Listing 9는 baz에는 25가 대입된다. 이것을 그림으로 그리면 그림 3과 같다[9].

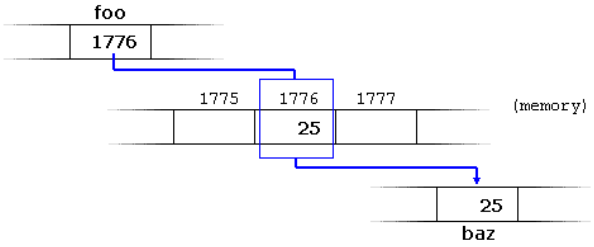

Figure 3: 주소 연산자

중요한 점은 **foo**와 ***foo**가 다르다는 것이다. **foo**는 1776이라고 하는 주소값이고, ***foo**는 메모리 주소 1776이 가지고 있는 데이터, 즉 25를 의미한다. 이제 다음 예제들을 통해 포인터에 익숙해지자.

Listing 8: Pointer

```
1  #include <iostream>
2  using namespace std;
3
4  int main ()
5  {
6    int firstvalue, secondvalue;
7    int * mypointer;
8
9    mypointer = &firstvalue;
10   *mypointer = 10;
11   mypointer = &secondvalue;
12   *mypointer = 20;
13   cout << "firstvalue is " << firstvalue << '\n';
```

[9] http://www.cplusplus.com/doc/tutorial/pointers/

```
14    cout << "secondvalue is " << secondvalue << '\n';
15    return 0;
16 }
```

Listing 9의 13번줄 출력을 눈여겨보자.

Listing 9: Pointer

```
1  #include <iostream>
2  #include "Matrix.h"
3  using namespace std;
4
5  void main(int argc, char* argv[])
6  {
7      if (argc != 3)
8      {
9          cout << "err: Must input two arguments\n";
10         return;
11     }
12     char* program_name = argv[0];
13     printf("%p , %p\n", program_name, argv[0]);
14     cout << "This program is '" << program_name << "'." << endl;
15     int in1 = atoi(argv[1]); // first argument
16     int in2 = atoi(argv[2]); // second argument
17     Matrix mat(in1, in2);
18     printf("Number of rows: %d\n", mat.getRows());
19     printf("Number of columns: %d\n", mat.getCols());
20 }
```

Listing 10은 배열과 포인터를 이용한 예제이다.

Listing 10: Pointer

```
1  #include <iostream>
2  using namespace std;
3
4  int main ()
5  {
6    int numbers[5];
7    int * p;
8    p = numbers;  *p = 10;
9    p++;  *p = 20;
10   p = &numbers[2];  *p = 30;
11   p = numbers + 3;  *p = 40;
12   p = numbers;  *(p+4) = 50;
```

```
13    for (int n=0; n<5; n++)
14      cout << numbers[n] << ", ";
15    return 0;
16  }
```

Listing 11은 행렬 클래스를 1차원 배열과 더블포인터를 이용해 만들었다. Listing 11의 1-20번 줄은 **Matrix.h** 파일이고, 22-61번 줄은 **Matrix.cpp** 파일이다.

Listing 11: Pointer

```
1  // Matrix.h
2  #ifndef _MATRIX_HEADER_H_
3  #define _MATRIX_HEADER_H_
4
5  class Matrix
6  {
7      int r;
8      int c;
9      double* data;
10     double** access;
11 public:
12     Matrix();
13     ~Matrix();
14     Matrix(int _r, int _c);
15     Matrix(int _r, int _c, double *_data);
16     int rows() { return r; };
17     int cols() { return c; };
18     double& elem(int _r, int _c) { return access[_r][_c]; };
19 };
20 #endif
21
22 // Matrix.cpp
23 #include <iostream>
24 #include "Matrix.h"
25
26 Matrix::Matrix()
27 {
28     r = 0;
29     c = 0;
30 }
31
32 Matrix::~Matrix()
33 {
```

```
34      delete[] data;
35      delete[] access;
36  }
37
38  Matrix::Matrix(int _r, int _c)
39  {
40      r = _r;
41      c = _c;
42      data = new double[r*c];
43      access = new double*[r];
44
45      for (int i = 0; i < r; ++i)
46          access[i] = &data[i*c]; // data + (i*c);
47      memset(data, 0, r*c * sizeof(double)); // init
48  }
49
50  Matrix::Matrix(int _r, int _c, double *_data)
51  {
52      r = _r;
53      c = _c;
54      data = new double[r*c];
55      access = new double*[r];
56
57      for (int i = 0; i < r; ++i)
58          access[i] = &data[i*c]; // data + (i*c);
59
60      memcpy(data, _data, r*c * sizeof(double));
61  }
```

4 파일 입력: 데이터 받기

여기서는 **ifstream**을 써서 데이터를 받는 법을 공부합니다. 데이터를 받아서 행렬 클래스에 저장하는 것을 해보도록 하겠다. 데이터 파일의 형태는 Listing 12와 같다.

Listing 12: 데이터 파일 (A.txt)

```
1  3 3
2  1 0 0
3  0 1 0
4  0 0 1
```

첫째줄의 두개의 숫자는 각각 행렬의 행과 열의 수를 의미한다. 그리고 이어지는 두번째 줄부터 행렬의 원소들이 나열된다. 같은 행에서는 공백으로 열을 구분하고 행은 개행(newline)으로 구분

해서 쓴다. 그럼 이러한 데이터를 받는 코드를 작성해보자. Listing 13에서 5-16번줄에 **A.txt**에서 데이터를 받아 **double** 형 변수인 **raw**에 저장하는 코드가 있다. 18번줄 코드는 **A.txt**에 적혀진 행렬의 크기대로 데이터가 들어왔는지 확인하는 코드이다. 24번 줄은 Listing 11의 **Matrix** 클래스에 파일로부터 입력받은 데이터를 **mat** 클래스에 넘겨주는 코드다. 25-32번줄은 **mat**의 내용을 출력하고 있다. 34-37번줄은 2×2크기의 행렬 클래스를 만들고 임의의 데이터를 대입하는 코드다. 38-45는 그 내용을 출력하고 있다. 47번줄에서 힙영역에 할당한 메모리를 해제한다.

Listing 13: 데이터 파일 (A.txt)

```
1  #include <iostream>
2  #include <fstream>
3  #include "Matrix.h"
4
5  using namespace std;
6
7  void main()
8  {
9      ifstream fin("A.txt");
10     if (!fin.is_open())
11         return;
12
13     int row, col;
14     fin >> row;
15     fin >> col;
16
17     double *raw = new double[row*col];
18     int idx = 0;
19     while (fin >> raw[idx])
20         ++idx;
21
22     _ASSERT(idx == row * col);
23
24     Matrix mat(row, col, raw);
25     for (int i = 0; i < mat.rows(); ++i)
26     {
27         for (int j = 0; j < mat.cols(); ++j)
28         {
29             cout << mat.elem(i, j) << ", ";
30         }
31         cout << endl;
32     }
33
34     Matrix mat2(2, 2);
```

```
35      for (int i = 0; i < mat2.rows(); ++i)
36          for (int j = 0; j < mat2.cols(); ++j)
37              mat2.elem(i, j) = i + j;
38      for (int i = 0; i < mat2.rows(); ++i)
39      {
40          for (int j = 0; j < mat2.cols(); ++j)
41          {
42              cout << mat2.elem(i, j) << ", ";
43          }
44          cout << endl;
45      }
46
47      delete[] raw;
48  }
```