C/C++ Day 2

윤광진, 광주과학기술원 02/06/2018

1 Function

C++에서 함수(function)는 구문(statement)들의 집합으로 이루어져있고 프로그램에의해 호출되었을때 가지고있는 일련의 구문들을 실행하게 된다. 일반적으로 함수를 정의 할때는

Listing 1: function

```
1 type name (parameter1, parameter2, ...) { statements }
```

와 같이 한다.

여기서 **type**는 함수에의해 반환되는 데이터의 자료형이다. **name**은 함수의 이름이다. 함수의 이름은 특별한 경우(상속 및 overloading 등)를 제외하면 고유해야한다. **parameter**를 통해 데이터를 넘겨 받는 것이 가능하다. 일반적으로 파라미터의 수는 제한이 없다 (아예 없어도 된다). 함수를 호출 하는 곳에서 데이터를 넘겨주기 위해 이곳에 넣는 변수를 **argument**라 한다 (Listing 3 하단의 설명 참고) **statements**는 함수의 함수의 몸체이며 중괄호로 감싸져있다. 함수가 호출되었을때 이 부분의 구문들이 실행된다. 1일 차에서 배웠던 **printf**도 함수다.

Listing 2: printf

```
1 int printf( const char* format, ... );
```

다음 예제를 통해 **addition** 함수가 어떤 동작을 하는지 보자.

Listing 3: function example

```
1  #include <iostream>
2  using namespace std;
3  int addition (int a, int b)
4  {
5    int r;
6    r=a+b;
7    return r;
8  }
9  void main ()
10 {
11   int z;
12   z = addition (5,3);
13   cout << "The result is " << z;
14 }
```

위 Listing 3에서 12번째 줄에 **addition** 함수를 호출하면서 넘겨준 5와 3을 **argument**라고 하고, 3번째 줄에서 넘겨받는 **a**와 **b**를 **parameter**라고 한다. 그리고 두번째 예제를 보자.

Listing 4: function example

```
#include <iostream>
using namespace std;
void swap(int _a, int _b)
{
    int tmp = _a;
    _a = _b;
    _b = _a;
}
void main()
{
    int a10 = 10;
    int b20 = 20;
    swap(a10, b20);
    cout << "Result of swap fucntion:" << endl;
    cout << "a10: " << a10 << endl;
    cout << "b20: " << b20 << endl << endl;
}
```

위 Listing 4 예제에서 **swap** 함수는 두 정수형 변수 **a10**과 **b20**를 입력으로 받아 두 변수의 내용을 바꾸는 작업을 하고 있다. 그것의 결과를 **main** 함수에서 출력을 하고 있는데 어떤 결과가 나올지 예측해 보자.

1.1 Call by Value, Pointer, Reference

함수의 **parameter**는 함수 내부에서 지역변수처럼 사용 할 수 있다. 이 지역 변수는 함수가 호출되는 과정에서 생성되고 함수가 종료될때 사라진다. 함수를 호출할때 **parameter**에 데이터를 넘겨주는 방법은 다음과 같이 세 가지가 있다.

- Call by Value: 함수를 호출할때 데이터의 값들을 복사하여 넘겨받는다. 이 경우 함수 내부에서 데이터를 변경하여도 **argument**에는 아무런 영향이 없다.
- Call by Pointer: 함수를 호출할때 데이터가 저장되어있는 위치(주소=Pointer)를 넘겨준다. 이 주소를 이용해 **argument**의 데이터를 변경할 수 있다.
- Call by Reference: 함수를 호출할때 데이터의 참조(Reference)를 넘겨준다. 이 참조를 이용해 **argument**의 데이터를 변경할 수 있다.

우선 Call by Value, Pointer, Reference의 차이를 확인하기 위해 두 변수의 데이터를 **main** 함수 안에서 swap하는 프로그램을 작성해보자.

Listing 5: swap program

```
#include <iostream>
using namespace std;
void main()
```

```
4  {
5      int a10 = 10;
6      int b20 = 20;
7      {
8          int tmp = a10;
9          a10 = b20;
10         b20 = tmp;
11     }
12     cout << "changed in main func" << endl;
13     cout << "a10: " << a10 << endl;
14     cout << "b20: " << b20 << endl << endl;
15 }
```

Listing 5는 **main** 함수 내부에서 두 **a10**과 **b20** 변수의 내용을 바꾸는 코드이다. 실행의 결과로 두 변수의 값이 바뀌게 된다. 이제부터 Listing 5의 7-11번줄의 swapping 코드를 Call by Value, Call by Pointer, Call by Reference로 각각 작성해서 어떤 차이가 있는지 보자.

우선 Listing 6 은 Call by Value 이다. Listing 4와 함수명을 제외하고는 똑같다.

Listing 6: Call by Value

```
1  #include <iostream>
2  using namespace std;
3  //call by value
4  void callbyvalue(int _a, int _b)
5  {
6      int tmp = _a;
7      _a = _b;
8      _b = tmp;
9  }
10 void main()
11 {
12     int a10 = 10;
13     int b20 = 20;
14     callbyvalue(a10, b20);    // call by value
15     cout << "call by value:" << endl;
16     cout << "a10: " << a10 << endl;
17     cout << "b20: " << b20 << endl << endl;
18 }
```

함수의 실행 결과로 **a10**과 **b20**의 값이 변경되지 않는다. 왜냐면 4번줄에 위치한 **callbyvalue** 함수의 **parameter**들인 **_a**와 **_b**는 함수의 지역변수로만 작동하기 때문이며 **int**형 데이터만 담을 수 있기 때문이다.

다음은 Call by Pointer 이다. Listing 7를 실행하여 결과를 확인해 보자.

Listing 7: Call by Pointer

```
1  #include <iostream>
2  using namespace std;
3  //call by pointer
4  void callbypointer(int* _a, int* _b)
5  {
6      int tmp = *_a;
7      *_a = *_b;
8      *_b = tmp;
9  }
10 void main()
11 {
12     int a10 = 10;
13     int b20 = 20;
14     callbypointer(&a10, &b20);    // call by pointer
15     cout << "call by pointer:" << endl;
16     cout << "a10: " << a10 << endl;
17     cout << "b20: " << b20 << endl << endl;
18 }
```

실행의 결과로 두 변수의 값이 바뀌어 있는것을 확인 할 수 있다. 또한 4번줄과 6-8번줄에 포인터 연산자인 **asterisk** (*) 와 주소값 연산자인 **ampersand** (&)가 눈에 띈다. 4번줄의 **parameter** **_a**와 **_b**는 14번 줄에서 함수가 호출 될때 **main** 함수의 지역변수인 **a10**과 **b20**의 주소값을 **argument**로써 넘겨 받는다. **_a**와 **_b**도 마찬가지로 **callbypointer** 함수 내부에서 지역변수로 작동하지만 포인터 변수인 관계로 **main** 함수의 **a10**과 **b20**의 주소값에 직접 접근하여 데이터를 변경 할 수 있는 것이다.

다음은 Call by Reference 다. Listing 8를 실행해서 결과를 확인하자.

Listing 8: Call by Reference

```
1  #include <iostream>
2  using namespace std;
3  //call by reference
4  void callbyreference(int& _a, int& _b)
5  {
6      int tmp = _a;
7      _a = _b;
8      _b = tmp;
9  }
10 void main()
11 {
12     int a10 = 10;
13     int b20 = 20;
```

```
14      callbyreference(a10, b20);    // call by reference
15      cout << "call by reference:" << endl;
16      cout << "a10: " << a10 << endl;
17      cout << "b20: " << b20 << endl << endl;
18  }
```

Listing 8의 **callbyreference** 함수는 Listing 6의 **callbyvalue** 함수와 크게 다르지 않아보인다. 다만 4번줄의 **parameter** 선언부에 참조 선언(&)이 포함된 것이 다를 뿐이다[1]. 함수가 호출 될때 참조로 **argument**를 넘겨받으면 해당 **argument**를 부르는 또 다른 이름을 부여하는 것이라고 생각하면 편하다[2]. 따라서 Call by Reference를 통해서 넘겨받은 **parameter**는 **argument**와 완전히 같은 변수이고 **callbyreference** 함수에서 **parameter** 조작을 통해 데이터를 변경하면 **main** 함수의 **argument**에도 똑같이 적용된다.

Reference 선언이 어떻게 작동하는지 확인해 보기위해 다음의 Listing 9을 실행하여 결과를 확인해 보자.

Listing 9: Reference declaration

```
1  #include <iostream>
2  using namespace std;
3  void main()
4  {
5      int a = 2;
6      int& b = a;
7      cout << "b: " << b << endl;
8      b = -1;
9      cout << "a: " << a << endl;
10     cout << "b: " << b << endl;
11 }
```

2 Class

클래스는 객체 지향 프로그래밍 언어의 중요한 개념이다. C++도 객체 지향 언어로써 **Class**를 사용할 수 있다. **class**는 C의 **struct**와 자주 비교되는데 우선 **struct**(구조체)부터 알아보자.

Listing 10: Struct, 구조체

```
1  #include <stdio.h>
2
3  struct Matrix
4  {
5      int rows;
6      int cols;
```

[1] ampersand가 쓰이는 위치에 따라 주소 연산자가 될 수도 있고 참조 선언이 될 수도 있는것에 유의하라
[2] 사실은 참조도 **argument**의 주소값을 가르키는 또 다른 변수를 만드는 것이다.

```
7  };
8  void main()
9  {
10     Matrix mat; // struct Matrix mat;
11     mat.rows = 2;
12     mat.cols = 3;
13     printf("Number of rows: %d\n", mat.rows);
14     printf("Number of columns: %d\n", mat.cols);
15 }
```

Listing 10에 **Matrix** 구조체의 정의가 3-7줄에 되어 있다. 구조체 정의 부분을 보면 두개의 정수형 변수 **rows**와 **cols**가 정의 되어있고 이 둘이 하나의 구조체 **Matrix**에 묶여있다. 이때 **rows**와 **cols**를 이 구조체의 멤버 변수라고 부른다. **main** 함수의 10번째 줄에서 **Matrix** 구조체를 **mat**라는 이름으로 하나 생성하는 부분과 11-12번 줄에서 생성된 **mat**의 열과 행의 갯수를 설정해 주는 부분을 눈여겨 보자. 정의된 구조체의 멤버 변수에 접근할때 마침표(.) 연산자를 통해 접근하는 것을 볼수 있다.

이제 **class**에 대해서 알아보자. 다음 Listing 11를 보자.

Listing 11: Class

```
1  #include <stdio.h>
2
3  class Matrix
4  {
5  public:
6      int rows;
7      int cols;
8  };
9  void main()
10 {
11     Matrix mat; // struct Matrix mat;
12     mat.rows = 2;
13     mat.cols = 3;
14     printf("Number of rows: %d\n", mat.rows);
15     printf("Number of columns: %d\n", mat.cols);
16 }
```

Matrix 선언부의 키워드가 **struct**에서 **class**로 바뀐것을 보자. 이제 **Matrix**는 구조체가 아니라 **클래스**다. 그리고 5번줄에 **public:** 키워드가 보인다. 이것은 5번줄 밑에 선언된 멤버들이 **public** 접근성을 가진다는 의미이고 클래스의 내부외부에서 모두 접근 가능하다는 뜻이다. **class**의 접근에는 **public, private, proteced** 세 가지 키워드가 있다. **public**으로 선언된 멤버는 어디서든 접근이 가능하며, **private**로 선언된 멤버는 해당 **클래스**의 내부에서만 접근이 가능하다. **protected**는 **클래스**의 상속과 관련이 있는데 추후에 설명이 필요할때 하겠다. **struct**(구조체)의 멤버는 일반적으로 **public**이다. 따라서 Listing 10의 **main** 함수에서 **Matrix**

의 멤버 변수에 접근 할 수 있었다. 반면에 **class**의 멤버는 디폴트로 **private**이다. 따라서 5번 줄에 **public:** 선언을 해주지 않으면 **main**함수에서 접근이 불가능하다.

클래스는 멤버 변수뿐만 아니라 멤버 함수도 가질 수 있다[3]. 그리고 이 둘을 합해 클래스의 멤버라고 부른다.

Listing 12: Class

```
1  #include <stdio.h>
2
3  class Matrix
4  {
5      int rows;
6      int cols;
7  public:
8      void setRows(int _r)
9      {
10         rows = _r;
11     };
12     void setCols(int _c)
13     {
14         cols = _c;
15     };
16     int getRows()
17     {
18         return rows;
19     };
20     int getCols()
21     {
22         return cols;
23     };
24 };
25 void main()
26 {
27     Matrix mat; // struct MyMatrix mat;
28     mat.setRows(2);
29     mat.setCols(3);
30     printf("Number of rows: %d\n", mat.getRows());
31     printf("Number of columns: %d\n", mat.getCols());
32 }
```

Listing 12의 **Matrix**클래스 멤버 변수는 이제 **private** 접근 권한을 가진다. 그리고 7번줄 밑의 멤버 함수들 **setRows, setCols, getRows, getCols**이 **public** 접근 권한을 가지게

[3]정확히는 구조체도 멤버 함수를 가질수 있긴 하다.

된다. 멤버 변수를 **private**로 감춰 놓는 이유는 실수 혹은 고의로 (보통 실수일 확률이 많음) 멤버 변수의 값을 외부에서 변경 하지 못하도록 하기 위함이다. 대신 **public**에 있는 정해진 멤버를 통해서 멤버 변수를 변경 할수 있게 해서 코드의 안전성을 높일 수 있다. 또 눈여겨 볼것은 클래스 내부에 함수의 정의가 있는 점이다. 클래스 내부에 함수의 선언뿐아니라 정의도 같이 넣을 수 있다.

2.1 생성자와 소멸자

클래스의 생성자(Constructor)와 소멸자(Destructor)에 대해서 알아보자. Listing 12의 예제에서 **Matrix** 클래스를 **main** 함수에서 생성할때 생성(27번줄)과 데이터 입력(28-29번줄)을 따로 했다. 이번에는 클래스를 생성과 동시에 원하는 데이터로 초기화 할수 있는데 생성자가 그 기능을 수행할 수 있다. 생성자는 클래스가 생성되는 시기에 호출되고 소멸자는 클래스가 사라질때 호출된다. Listing 13를 보자.

Listing 13: Class

```
1  #include <iostream>
2  using namespace std;
3  class Matrix
4  {
5  public:
6      int rows;
7      int cols;
8      Matrix() { cout << "create\n"; };
9      ~Matrix() { cout << "delete\n"; };
10 };
11 void main()
12 {
13     Matrix mat; // struct MyMatrix mat;
14     mat.rows = 2;
15     mat.cols = 3;
16     printf("Number of rows: %d\n", mat.rows);
17     printf("Number of columns: %d\n", mat.cols);
18 }
```

Listing 13는 Listing 11와 똑같지만 생성자 **Matrix()**와 소멸자 ~**Matrix()**를 따로 정의 해준것만 다르다. 8번과 9번줄에 생성자와 소멸자가 정의되어있다. 특이한 점으로는 생성자와 소멸자 모두 반환 타입(return type)이 없고, 이름이 클래스 명과 같다는 점이다 (또 소멸자의 경우 이름 앞에 ~기호가 있다). 8번줄에 새로 정의된 생성자는 **cout**을 통해 'create'라는 출력을 내놓는다. 9번줄의 소멸자는 **cout**을 통해 'delete'를 출력한다. Listing 13에서 이 생성자와 소멸자가 호출되는 시점은 각각 13번 줄과 18번줄에서다. 왜냐면 **main** 함수의 13번 줄에서 **Matrix** 클래스를 **mat**이라는 이름으로 생성할때 생성자가 한번 호출되고 **main** 함수가 끝나는 시점에 **mat**도 같이 사라지기 때문이다.

그러면 이번에 생성과 동시에 멤버 변수를 초기화하는 생성자를 만들어보자.

Listing 14: Class

```
1  #include <iostream>
2  using namespace std;
3  class Matrix
4  {
5      int rows;
6      int cols;
7  public:
8      Matrix() { cout << "create\n"; };
9      ~Matrix() { cout << "delete\n"; };
10     Matrix(int _r, int _c)
11     {
12         cout << "create and init\n";
13         rows = _r; cols = _c;
14     }
15     void setRows(int _r)
16     {
17         rows = _r;
18     };
19     void setCols(int _c)
20     {
21         cols = _c;
22     };
23     int getRows()
24     {
25         return rows;
26     };
27     int getCols()
28     {
29         return cols;
30     };
31 };
32 void main()
33 {
34     Matrix mat = Matrix(2,3);
35     printf("Number of rows: %d\n", mat.getRows());
36     printf("Number of columns: %d\n", mat.getCols());
37 }
```

Listing 14의 10-14번 줄에 생성과 동시에 초기화를 수행하는 생성자가 정의되어 있다. 'create and init'을 출력하고 멤버변수 **rows**와 **cols**에 각각 **_r**과 **_c**를 대입한다. 이 생성자를 호출하는 법은 34번줄처럼 할 수 있다. 또는 **Matrix mat(2,3)** 으로 해도 된다. 전자를 C 스타일 초기

화 후자를 C++ 스타일 초기화라고도 한다. 그리고 **Matrix** 클래스 내의 생성자가 **Matrix**라는 이름으로 두개 존재하는데 (**Matrix()**와 **Matrix(int _r, int _c)**, ~**Matrix()**는 다른 이름이다) 두 함수의 **parameter**가 다르기 때문에 어떤 함수를 호출 하는건지 구분이 가능하므로 이런 형태가 가능하다. 이것을 함수 **overloading**이라고 부른다.

2.2 모듈화

지금까지는 하나의 **.cpp**파일에 클래스의 선언과 정의, **main** 함수의 정의까지도 다 포함 시켰다. 프로그램의 크기가 커지거나 프로젝트의 크기가 커지면 하나의 파일로 프로그램을 관리하는데 많은 어려움을 준다. 이번 장에서는 모듈화를 통해서 프로그램을 더 보기 좋게 만드는 법을 알아볼것이다.

방법은 **Matrix** 클래스의 선언은 헤더파일에 하고, 클래스 멤버 함수의 정의는 **.cpp** 파일에 하는 것이다. 그리고 나서 **Matrix** 클래스가 필요로 하는 파일**main.cpp**에서는 클래스의 헤더파일만 포함하면 되도록 하는 것이다. Listing 15이 **Matrix** 클래스의 헤더파일이고, 멤버 함수의 정의는 Listing 16에 있다. 그리고 이 클래스를 사용하는 **main** 함수의 정의는 Listing 17에 있다.

Listing 15: Matrix.h

```
1  #include <iostream>
2  using namespace std;
3  class Matrix
4  {
5      int rows;
6      int cols;
7  public:
8      Matrix();
9      ~Matrix();
10     Matrix(int _r, int _c);
11     void setRows(int _r);
12     void setCols(int _c);
13     int getRows();
14     int getCols();
15 };
```

Listing 16: Matrix.cpp

```
1  #include "Matrix.h"
2
3  Matrix::Matrix()
4  {
5      cout << "create\n";
6  };
7  Matrix::~Matrix()
8  {
9      cout << "delete\n";
```

```
10 };
11 Matrix::Matrix(int _r, int _c)
12 {
13     cout << "create and init\n";
14     rows = _r; cols = _c;
15 }
16 void Matrix::setRows(int _r)
17 {
18     rows = _r;
19 };
20 void Matrix::setCols(int _c)
21 {
22     cols = _c;
23 };
24 int Matrix::getRows()
25 {
26     return rows;
27 };
28 int Matrix::getCols()
29 {
30     return cols;
31 };
```

Listing 17: main.cpp

```
1 #include <iostream>
2 #include "Matrix.h"
3 using namespace std;
4
5 void main()
6 {
7     Matrix mat(2,3);
8     printf("Number of rows: %d\n", mat.getRows());
9     printf("Number of columns: %d\n", mat.getCols());
10 }
```

3 컴파일

이번 절에서는 작성한 코드를 직접 컴파일 해보면서 C++가 어떻게 실행 가능한 파일로 변환되는지를 경험해본다. 사실 Visual Studio를 사용하면 컴파일 및 링킹 등의 과정에 대해서 걱정을 할 필요가 없다. Visual Studio에서 정해진 버튼을 누르면 알아서 해주기 때문이다. 하지만 프로그램을

만들다보면 Visual Studio를 쓰지 못하는 환경에서 개발할 수도 있고, 또 이런 공부가 C++를 더 깊이 이해하는데 도움을 줄수 있기 때문에 공부해 보기로한다.

우선 위의 Listing 17의 **main** 코드를 다음의 Listing 18로 고쳐보자.

Listing 18: main.cpp

```
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 }
```

이 **main** 함수는 2개의 인자를 입력 받아 프로그램을 실행한다. 두 개의 인자를 입력받지 못했을 경우 9번줄의 에러 메세지를 출력하고 프로그램을 종료한다. 입력받은 두 개의 인자는 **main** 함수의 **in1**, **in2**로 대치되고(14-15번줄) **Matrix** 클래스의 생성자의 인자로 넘겨진다(16번줄). Visual Studio에서 인자를 넘겨주기 위해서는 **Project** → **Properties ...** → **Debugging** → **Command Arguments** 에 원하는 인자를 입력하면된다. 설정을 마친후 Visual Studio에서 프로그램을 실행해 결과를 확인해보자. 그리고 커맨드 창에서 해당 실행파일이 위치한 곳으로 이동해 실행파일을 직접 실행해보자.

이번에는 소스코드를 직접 컴파일 해서 실행파일을 만들어 보자. 우선 **Visual Studio 2017 용 x64 네이티브 도구 명령 프롬프트** 를 실행 시켜 커맨드 창(=명령 프롬프트=명령 창)을 열어야 한다[4]. 이 커맨드 창은 일반 커맨드 창과 크게 다를 것이 없으며 다만, 창 안에서 컴파일을 쉽게 할 수 있도록 환경변수 설정(특히 PATH 설정)이 완료된 창이다.

그럼 소스코드(**main.cpp**, **Matrix.cpp**, **Matrix.h**)가 위치해있는 폴더로 이동한다. 그리고 다음 명령어를 입력한다.

Listing 19: compile

```
1  cl /EHsc main.cpp Matrix.cpp
```

[4]윈도우 시작 화면에서 '명령'을 입력하면 검색 결과에 나타난다

cl은 MS에서 제공하는 C++ 컴파일러이다. **EHsc**는 예외처리 옵션으로써 여기서는 다루지 않는다. (https://msdn.microsoft.com/en-us/library/1deeycx5.aspx) 그 다음 인자는 **main.cpp**와 **Matrix.cpp**로 컴파일 할 소스코드 목록이다. 헤더파일은 컴파일 되지 않으므로 **Matrix.h**는 적어주지 않는다. 컴파일이 완료되면 같은 폴더에 **main.exe** 파일이 생성된다. Listing 20 처럼 이 파일을 실행하보자.

Listing 20: compile

```
1 main.exe 10 12
2 main.exe
```

Listing 20 의 1번줄의 실행결과와 2번줄의 실행결과를 비교해보자.

MS의 **cl.exe** 컴파일러는 생성되는 **exe** 파일의 이름을 첫번째로 입력된 소스코드의 이름으로 설정한다. 만약,

Listing 21: compile

```
1 cl /EHsc Matrix.cpp main.cpp
```

으로 입력하면 **Matrix.exe**로 실행파일이 생성된다.

생성되는 실행 파일의 이름을 임의로 설정 할 수 있는데 이 옵션이 **Fe** 옵션이다. 예를들어

Listing 22: compile

```
1 cl /EHsc /Fe:yoon Matrix.cpp main.cpp
```

로 입력하면 **yoon.exe**로 실행파일이 생성된다.