C++ 프로그래밍 기초

(1주차)

학습개요

- 학습 목표
  - C++ 콘솔 입출력을 구현할 수 있다.
  - 네임스페이스의 특징을 이해할 수 있다.
  - C/C++ 표준 라이브러리 사용을 위한 표준헤더 사용법을 익힐 수 있다.
  - C++의 참조에 의한 함수 호출, 함수 오버로딩, 인라인 함수 활용법을 익힐 수 있다.
  - C++의 동적 메모리 활용법을 익힐 수 있다.

- 학습 내용
  - 콘솔 입출력
  - 변수 선언, 네임스페이스, 표준헤더 파일
  - 함수
  - 동적 메모리 할당과 해제
  - 실습

C++의 콘솔입출력 (1)

- C++ 파일의 확장자는 .cpp여야 함.
- iostream.h 헤더 파일을 포함해야 함
- iostream.h 파일에는 입출력을 위한 iostream 클래스가 정의됨
- 표준 콘솔 입출력을 위해서 iostream 클래스 객체인 cin과 cout을 사용함
  - cin : 표준 입력 장치인 키보드로부터 바이트 스트림을 추출하기 위하여 스트림 삽입 연산자(stream insertion operator, >>)를 사용함
  - cout : 표준 출력 장치인 모니터로 바이트 스트림을 출력하기 위하여 스트림 추출 연산자(stream extraction operator, <<)를 사용함

```
01  #include <iostream>
02  int main()
03  {
04      char name[20] = {0};
05      int age;
06      std::cin >> name >> age;
07      std::cout << "Name is " << name << "\n" << "Age is " << age << "\n";
08      return 0;
09  }
```

```
홍길동 20
Name is 홍길동
Age is 20
```

C++의 콘솔입출력 (2)

1. iostream 클래스 객체인 cin과 cout을 사용할 때, C 언어의 printf( ), scanf( ) 함수와는 달리 << 또는 >>를 사용하여 입출력 스트림에 넣으면 된다.

2. 정수를 진법 변환하여 출력할 때는 dec, hex, oct를 사용하여 출력한다. 기본 값으로 10진수로 출력한다. 진법 변환 출력은 새로 지정할 때까지 계속 유효하다.

```
int a = 10;
std::cout << " a = " << dec << a << "\n";
std::cout << " a = " << hex << a << "\n";
std::cout << " a = " << oct << a << "\n";
```

3. 출력 자리 지정은 cout.width(n)으로 한다. 여기서 n은 전체 출력 자리를 나타내는 숫자이다. 출력 자리 지정은 바로 다음에 오는 출력에 한 번만 적용된다.

```
int a = 12;
std::cout.width(4);
std::cout << a <<"\n";
```

4. 실수 출력에서 유효 자리수 지정 출력은 cout의 cout.precision(n)으로 한다. 여기서 n은 유효 자리를 나타내는 숫자로, 기본 값은 6이다. 유효 자리수 지정 출력은 새로 지정하지 않으면 계속 유효하다.

```
float b=3.141592;
std::cout.precision(2);
std::cout << a <<"\n"; // 3.1 출력
```

변수 선언

- C++의 변수 선언은 C 언어와 달리 프로그램의 어느 곳에서나 가능함
- 변수를 사용하는 근처에서 선언할 수 있음
- 선언된 변수의 유효 범위(scope)는 C에서와 같이 블록 구조에 의한 정적 유효 범위 규칙(static scope rule)에 따라 정의됨
- 변수 선언이 블록의 어느 위치에서나 가능하기 때문에, C++ 변수의 유효 범위는 선언 위치에서 시작하여 블록의 끝까지임

```
01  #include <iostream>
02  int main()
03  {
04      int a, b;
05      a = 10;
06      b = 20;
07
08      int c; // C 언어에서는 오류
09      c = a + b;
10      std::cout << a << " + " << b << " = " << c << std::endl;
11      return 0;
12  }
```

```
10 + 20 = 30
```

네임스페이스와 이름 충돌 방지

- C 언어의 식별자 구분
  - 변수 이름, 함수 이름, 구조체 이름 등의 식별자(identifier)를 사용하여 서로 다른 이름을 구분
  - 블록 기반 정적 유효 범위 규칙에 따라 유효 범위가 결정되며, 변수인 경우 지역(local) 변수와 전역 변수(global variable)로 구분
  - 같은 범위에 이름이 하나 이상 중복되어 있으면 오류가 발생

- C++ 언어의 식별자 구분
  - 네임스페이스를 추가하여 같은 유효 범위를 네임스페이스로 구분 => 이름 충돌 방지
  - 특정 네임스페이스에 정의된 멤버를 지정하려면 범위 연산자(scope resolution operator, ::)를 사용
  - using 지시어(directive)를 사용하면, 해당 네임스페이스의 명칭을 현재 명칭으로 가져옴 => using 지시어가 속한 범위에서 네임스페이스에 지정된 명칭을 사용할 수 있음
  - 전역 범위에서 using 지시어가 선언되어 있으면, 프로그램의 모든 부분에서 네임스페이스가 유효하고, using 지시어가 함수 안에 있으면 해당 함수 안에서만 유효

네임스페이스 및 using 지시어 (1)

- 네임스페이스 및 using 지시어의 형식

```
namespace [identifier] { namespace-body }
...
using namespace identifier;
```

  - identifier : 네임스페이스 이름. 한 유효 범위 안에서 네임스페이스 이름은 유일해야 함
  - namespace-body : 네임스페이스가 동일한 멤버를 나타내는 것으로, 또 다른 네임스페이스도 가능함

네임스페이스 및 using 지시어 (2)

```
01  #include <iostream>
02  namespace NA
03  {
04      int a;
05  }
06  namespace NB
07  {
08      int a;
09  }
10
11  int main()
12  {
13      NA::a = 10;
14      NB::a = 20;
15      std::cout << "NA::a = " << NA::a << std::endl;
16      std::cout << "NB::a = " << NB::a << std::endl;
17
18      using namespace NA;
19      std::cout << "a = " << a << std::endl; // NA::a
20      return 0;
21  }
```

```
NA::a = 10
NB::a = 20
a = 10
```

네임스페이스 및 using 지시어 (3)

```
01  #include <iostream>
02  namespace NA
03  {
04      int a;
05  }
06  namespace NB
07  {
08      int a;
09  }
10  using namespace NA; // 전역 범위에서 유효
11  void func();
12
13  int main()
14  {
15      a = 10; // NA::a = 10;
16      func();
17      return 0;
18  }
19
20  void func()
21  {
22      std::cout << "a = " << a << std::endl;
23  }
```

```
a = 10
```

네임스페이스 및 using 지시어 (4)

```
01  #include <iostream>
02  namespace NA
03  {
04      int a;
05  }
06  namespace NB
07  {
08      int a;
09      int b;
10  }
11  using namespace NA; // 전역 범위에서 유효
12  void func();
13
14  int main()
15  {
16      int a;
17
18      a = 10; // 지역 변수 a = 10;
19      func();
20
```

네임스페이스 및 using 지시어 (5)

```
21      using namespace NB; // 지역 범위에서 유효
22      std::cout << "main, a = " << a << std::endl; // 지역 변수 a = 10;
23
24      b = 30; // NB::b
25      std::cout << "main, b = " << b << std::endl; // NB::b;
26
27      return 0;
28  }
29
30  void func()
31  {
32      a = 20; // NA::a
33      std::cout << "func, a = " << a << std::endl;
34  }
```

```
func, a = 20
main, a = 10
main, b = 30
```

표준헤더 파일 사용 (1)

- 모든 C++ 라이브러리 구성 요소(자료형, 함수, 클래스 등)는 표준 헤더 파일 한 개 이상에 정의 또는 선언됨
- C++ 표준 라이브러리는 네임스페이스 std 안에 포함됨
- C++에서는 #include <iostream>과 같이 헤더 파일 확장자 .h를 사용하지 않기를 권장함
- using namespace std를 통해 표준 라이브러리 네임스페이스를 지원하고 있으며, C 표준 라이브러리를 지원하기 위하여 파일 확장자 h는 제거하고 앞에 c를 추가한 헤더를 사용함

  예) C 표준 헤더 파일 time.h는 ctime, math.h는 cmath로 사용함
- 전통적인 헤더 파일(*.h)과의 차이

  표준 헤더는 std 네임스페이스에서 표준 라이브러리 함수를 지원함

```
01  #include <iostream>
02  using namespace std;
03  int main()
04  {
05    int a = 10;
06    cout << "a = " << a << endl; // using namespace std 구문이 없으면 오류 발생
07    return 0;
08  }
```

```
a = 10
```

표준헤더 파일 사용 (2)

```
01  #include <iostream>
02  #include <cmath>
03  using namespace std;
04  int main()
05  {
06    cout << "sqrt(2.0) = " << std::sqrt(2.0) << endl;
07    return 0;
08  }
```

```
sqrt(2.0) = 1.41421
```

참조연산자와 참조에 의한 함수 호출

- C++은 형식 인수를 참조 연산자(&, reference operator)로 정의하여 실 인수와 형식 인수를 다른 이름으로 사용하는 방법

```
01  #include <iostream>
02  void swap(int &i, int &j);
03  int main()
04  {
05      int a = 1, b = 2;
06      std::cout << &a << std::endl;
07      swap(a, b); // a, b 값 교환
08      std::cout << "a = " << a << "\n" << "b = " << b << std::endl; // a = 2, b = 1
09      return 0;
10  }
11
12  void swap(int &i, int &j)
13  {
14      std::cout << &i << std::endl; // main의 &a와 동일한 주소
15      int temp;
16      temp = i;
17      i = j;
18      j = temp;
19  }
```

```
0023FEA8
0023FEA8
a = 2
b = 1
```

함수 오버로딩 (1)

- C++에서는 같은 유효 범위(scope) 안에 있는 인수의 수 또는 인수의 자료형이 다르면 이름이 같은 함수를 한 개 이상 정의할 수 있음
- 함수 오버로딩(function overloading)은 객체지향 프로그래밍 언어의 특징인 다형성(polymorphism) 중 하나임
  - 동일한 함수 이름이라도 인수의 수 또는 자료형이 다르면 함수 오버로딩 사용이 가능함
  - 인수의 수와 자료형이 같고, 함수의 반환형(return type)만 다른 경우에는 함수 오버로딩을 할 수 없음

```
01  #include <iostream>
02  int sum(int n1, int n2);
03  int sum(int n1, int n2, int n3);
04  double sum(double f1, double f2);
05
06  int main()
07  {
08      int s1 = sum(1, 2);
09      int s2 = sum(1, 2, 3);
10      double s3 = sum(1.0, 2.0);
11      std::cout << "s1 = " << s1 << "\n";
12      std::cout << "s2 = " << s2 << "\n";
```

함수 오버로딩 (2)

```
13      std::cout << "s3 = " << s3 << "\n";
14      return 0;
15  }
16
17  int sum(int n1, int n2)
18  {
19      return n1 + n2;
20  }
21
22  int sum(int n1, int n2, int n3)
23  {
24      return n1 + n2 + n3;
25  }
26
27  double sum(double f1, double f2)
28  {
29      return f1 + f2;
30  }
```

```
s1 = 3
s2 = 6
s3 = 3
```

인수의 기본 값 (1)

- C++에서는 함수의 인수에 기본 값을 정의할 수 있음
- 함수를 호출할 때 실 인수에 값을 주지 않으면 형식 인수는 기본 값을 저장함
- 함수 원형 선언에 기본 값을 설정함
- 기본 인수를 사용할 때 주의할 점은 오른쪽에 있는 인수에 기본 값을 지정하지 않으면, 왼쪽에 있는 인수에 기본 값을 지정할 수 없음

  int func(int a = 0, int b=0, int c); // 오류

```
01  #include <iostream>
02  int func(int a, int b = 0, int c = 1);
03  // int func( int , int = 0, int = 1 );
04
05  int main()
06  {
07      int a = func(1);
08      int b = func(1, 2);
09      int c = func(1, 2, 3);
```

인수의 기본 값(2)

```
10      std::cout << "a = " << a << "\n";
11      std::cout << "b = " << b << "\n";
12      std::cout << "c = " << c << "\n";
13      return 0;
14  }
15
16  int func(int a, int b, int c)
17  {
18      return (a + b + c);
19  }
```

```
a = 2
b = 4
c = 6
```

인라인 함수 (1)

- 일반적인 함수 선언 및 호출은 함수 호출에 따른 시간 지연을 일으킬 수 있음
- 인라인(inline)으로 정의된 함수는 호출하는 곳마다 해당 함수의 코드 부분이 삽입됨 => 이러한 기능을 일반적으로 매크로 확장(macro expansion)이라고 함
- 인라인 함수는 코드 부분이 적거나 호출 횟수가 적을 때 효과적임
- 함수의 코드 부분이 큰 경우 인라인 함수를 사용하면 실행 파일의 크기가 커지는 단점이 존재함

```
01  #include <iostream>
02  inline int maximum(int a, int b);
03  int main()
04  {
05      int a = maximum(1, 2);
06      int b = maximum(3, 2);
07
08      std::cout << "a = " << a << ", b = " << b << std::endl;
09
10      return 0;
11  }
```

인라인 함수 (2)

```
12  inline int maximum(int a, int b)
13  {
14      return (a > b) ? a : b;
15  }
```

```
a = 2, b = 3
```

new와 delete (1)

- C++에서는 실행 시간에 동적 메모리 할당(dynamic memory allocation)을 위한 new 연산자와 할당된 메모리를 회수하기 위한 delete 연산자를 제공함
- new 연산자를 이용해서 동적으로 할당한 메모리는 힙(heap)에서 할당되므로, 포인터 변수의 유효 범위를 벗어나도 회수하지 않으면 할당된 상태로 계속 남음
- delete 연산자를 이용해서 힙에 할당된 메모리를 할당 해제함
- 동적으로 할당된 배열의 경우 delete []포인터변수명; 구문으로 메모리 할당을 해제함.

```
01  #include <iostream>
02  #include <cstdlib>
03  #include <ctime>
04  int main()
05  {
06      int *pValue = new int[5];
07      std::srand((unsigned) std::time(NULL));
08      for (int i = 0; i < 5; i++)
09      {
10          pValue[i] = std::rand(); // 0~32767 사이의 난수 저장
11          std::cout << "pValue[" << i << "] = " << pValue[i] << std::endl;
12      }
13      delete []pValue; // 메모리 해제
```

new와 delete (2)

```
14      // 정수 10개를 저장할 메모리를 연속으로 할당
15      pValue = new int[10];
16      for (int i = 0; i < 10; i++)
17      {
18          pValue[i] = rand();
19          std::cout << "pValue[" << i << "] = " << pValue[i] << std::endl;
20      }
21      delete []pValue; // 메모리 해제
22      return 0;
23  }
```

```
pValue[0] = 16600
pValue[1] = 10116
pValue[2] = 2632
pValue[3] = 18452
pValue[4] = 7445
pValue[0] = 28878
pValue[1] = 17313
pValue[2] = 20132
pValue[3] = 6876
pValue[4] = 18270
pValue[5] = 17284
pValue[6] = 27258
pValue[7] = 529
pValue[8] = 14805
pValue[9] = 27427
```

학습정리

- C++ 파일의 확장자는 반드시 .cpp이어야 합니다.
- iostream.h 파일에는 입출력을 위한 iostream 클래스가 정의되어 있으며, 표준 콘솔 입출력을 위해 iostream 클래스 객체인 cin과 cout와 <<, >> 연산자를 사용해 입출력을 구현합니다.
- C++의 변수 선언은 C 언어와 달리 프로그램의 어느 곳에서나 선언 가능합니다.
- 네임스페이스의 식별자는 범위 연산자(::)를 사용하며 접근하며, using 지시어를 사용해 해당 네임스페이스의 식별자를 using 지지어가 사용된 범위에서 접근할 수 있게 합니다.
- C++ 표준 라이브러리는 네임스페이스 std 안에 포함되며, 헤더 파일 확장자 .h를 사용하지 않는 것을 권장하며, C 표준 라이브러리를 지원하기 위하여 파일 확장자 h는 제거하고 앞에 c를 추가한 헤더를 사용합니다.
- C++은 형식 인수를 참조 연산자(&)로 정의하여 참조에 의한 함수 호출을 할 수 있습니다.
- C++에서는 인수의 수 또는 자료형이 다르면 이름이 같은 함수를 한 개 이상 정의할 수 있으며, 이를 함수 오버로딩이라 합니다.
- C++에서는 함수 인수에 기본 값을 설정할 수 있으며, 가장 오른쪽에 있는 인수부터 기본 값을 지정해야 합니다.
- 인라인(inline) 함수는 코드 부분이 적거나 호출 횟수가 적을 때 사용하면 효과적입니다.
- C++에서는 실행 시간에 동적 메모리 할당을 위한 new 연산자와 할당된 메모리를 회수하기 위한 delete 연산자를 제공합니다.