제1장 객체지향 기본 문법

1. C++의 객체를 이용한 입출력

- 입출력 객체(cout, cin)와 오버로드된 연산자(<<, >>)를 이용한 입출력.
- 데이터형 자동 인식 - C언어에서 사용되던 형식변환문자(%d, %c, %lf...) 불필요
- C보다 편리하고 형식변환문자를 잘못된 사용하는 것과 같은 불필요한 에러를 줄일 수 있다.
- cout과 cin객체는 iostream 헤더파일에 선언되어 있음.

► 객체 : 사용자정의 데이터타입인 클래스(data + function 이 캡슐화 되어있는 데이터 타입)가 구체화된 것. 즉, 클래스의 실제 메모리 블록

* 클래스(class) == 객체형(object type)

* 객체(object)==객체 인스턴스(object instance)==인스턴스 변수(instance variable)

► 연산자 Overloading : 동일한 연산자가 서로 다른 연산에 사용되는 기능

► cout 객체를 이용한 출력

- ostream 클래스로 생성된 객체
- 문자열, 숫자, 개별적인 문자들을 포함한 여러 가지 정보를 출력하는 방법이 정의되어 있는 객체

```
cout        <<        "C++ 프로그래밍"
 ↑          ↑          ↑
객체     inserter   전달인자
      (연산자 오버로딩)
```

► cin 객체를 이용한 입력

```
int var;
cin         >>        var
 ↑          ↑          ↑
객체     extractor  전달인자
      (연산자 오버로딩)
```

► cin의 멤버 함수

- getline() : 여백이 포함된 문장 입력, 지정 길이 또는 지정 문자까지 입력 가능

  ex) cin.getline(buff, size, '₩n');

- get() : 모든 종류의 문자 입력

  ex) cin.get(ch); 또는 ch=cin.get();

- clear() : cin 입력 시 실패에 의해 set된 flag멤버 clear 기능

- fail() : cin 입력 실패 시 set되는 flag멤버 값을 검사(입력 실패 시 true 리턴)

- ignore() : 버퍼 지우기

  ex) cin.ignore(); // 1 바이트 지우기

  ex) cin.ignore(size, '₩n'); // size범위 내에서 '₩n'까지의 문자까지 모두 삭제

[예제1-1] C++에서의 입출력

```
#include<iostream>
using   namespace std;
void my_flush();
int main()
{
        int intNumber;
        double doubleNumber;
        char ch;
        char str[100];

        cout <<"정수값입력: ";
        cin >> intNumber;

        cout <<"실수값입력: ";
        cin >> doubleNumber;

        cout <<"문자입력: ";
        cin >> ch;

        cout <<"문자열입력: ";
        cin >> str;

        cout <<"intNumber = "<< intNumber << endl;
        cout <<"doubleNumber = "<< doubleNumber << endl;
        cout <<"ch = "<< ch << endl;
        cout <<"str = "<< str << endl;

        my_flush();

        cout <<"중간에 여백있는 문자열을 입력하세요: " ;

        cin.getline(str, sizeof(str));
        cout <<"입력 받은문자열: "<< str << endl;

        cout <<"여백문자를 입력하세요(space, tab, enter) : " ;
        ch=cin.get();           //get()은 오버로딩된 멤버함수로 어떤 문자든 한 문자 읽어들임
        cout <<"입력한 여백문자의 아스키코드 값: "<< (int)ch << endl;
        return 0;               // ← 생략가능

}
void my_flush() // 사용자가 만든 cin 입력 버퍼 지우기 함수
{
        while(cin.get()!='₩n') ;
}
```

[예제1-2] C++에서의 입력 예외처리

```
#include<iostream>
using   namespace std;
void my_flush();
int main()
{
      int num;
      cin >> num;
      while(cin.fail())         // cin입력이 실패했는지 검사 -
      {                         // cin자체 flag를 검사해서 입력이 실패하면 true값을 리턴
            my_flush();       // cin 입력버퍼에 들어 있는 불필요한 데이터를 지워줌
                              // C언어에서 사용되던 fflush(stdin)과 유사한 기능임.
            cin >>num;      // 다시 입력을 시도 함.
      }
      cout <<"num="<< num << endl;
      return 0;
}
//---------------------------------------------------------------------
void my_flush() // 사용자가 만든 cin입력버퍼 지우기함수
{
      cin.clear();        // cin의 자체flag값을 0으로 만듬- 이 부분이 빠지면 cin 입력 시도를
                          // 하지 않고 무조건 실패로 인식되어 무한loop에 빠짐
      while(cin.get()!='₩n') ; // cin입력 버퍼내의 모든 데이터 삭제
}
```

[예제1-3] C++에서의 주소 출력

```
#include<iostream>
using   namespace std;
int main()
{
            int intArray[5] = {1,3,5,7,9};
            char charArray[10] = "apple";

            cout << intArray << endl;               // 주소출력
            cout << charArray << endl;              // 문자열출력
            cout << (void *)charArray << endl;   // 주소출력

            return 0;
}
```

◈ 조정자를 이용한 입출력

dec, hex, oct	정수값을 10진수, 8진수, 16진수로 출력한다.
showbase,noshowbase	16진수, 8진수 진법접두어(0x, 0) 출력 설정, 취소
fixed	소수점 형태로 출력
scientific	지수 형태로 출력
setprecision(n)	일반 출력 모드에서는 유효숫자 n 자리 출력 fixed와 scientific모드에서는 소수점 이하 n자리 출력
setw(n)	field 폭 지정 (** 1회만 유효 함)
left, right	왼쪽, 오른쪽 정렬

[예제1-4] 조정자를 이용한 다양한 출력

```
#include<iostream>
#include<iomanip> // setprecision, setw 사용을 위해 인클루드
using namespace std;
int main()
{
        int number = 12;
        cout << showbase;
        cout <<"10진수로출력: "<< number << endl;
        cout <<"16진수로출력: "<< hex << number << endl;
        cout <<" 8진수로출력: "<< oct << number << endl;

        double dnumber = 7.12345;
        cout << setprecision(3);
        cout << dnumber << endl; // 유효숫자 3자리 출력
        cout << fixed;
        cout << dnumber << endl; // fixed 모드 - 소숫점 이하3자리 출력
        cout << scientific;
        cout << dnumber << endl; // scientific 모드 - 소수점이하 3자리 출력

        cout <<"|"<< setw(10) << number <<"|"<< setw(15) << dnumber <<"|"<< endl;

        cout << left;
        cout <<"|"<< setw(10) << number <<"|"<< setw(15) << dnumber <<"|"<< endl;

        cout << right;
        cout <<"|"<< setw(10) << number <<"|"<< setw(15) << dnumber <<"|"<< endl;
        return 0;
}
```

```
cout.unsetf(ios_base::scientific); // 실수값 출력 형태를 기본 형태로 되돌림
cout.unsetf(ios_base::showbase);   // 진법접두어 출력 취소
cout << setprecision(-1);      // 유효숫자 및 소수점 이하 자리수 지정 취소
```

◈ cout의 setf(long) 멤버함수를 이용한 출력양식 조정

상 수 명	의 미
ios::showbase	출력에서 C++의 기준 접두어(0,0x)를 붙인다.
ios::showpoint	소수점과 소수점 이하의 0을 생략하지 않는다. 일반 출력 모드에서는 유효숫자 6자리로 출력하고 fixed나 scientific 모드에서는 소수점 이하 6자리까지 출력
ios::uppercase	16진수 출력을 위해 대문자를 사용한다.
ios::showpos	양수 앞에 +부호를 붙인다.
* setf()멤버함수를 통해 지정된 각각의 기능들은 unsetf()함수에 의해 기능이 해제된다.	

[예제1-5] cout의 setf(long)멤버함수의 활용 예

```
#include<iostream>
using namespace std;

int main(void)
{
        int a=10, b=126;
        float f=3.14, g=5.0;
        cout.setf(ios::showpos);
        cout << a << endl;
        cout << f << endl;

        cout << hex << a << endl;
        cout.setf(ios::showbase);
        cout << a << endl;
        cout << b << endl;

        cout.setf(ios::uppercase);
        cout << a << endl;
        cout << b<< endl;

        cout.setf(ios::showpoint);
        cout << f << endl;
        cout << g << endl;
        return 0;
}
```

(결과)

```
  +10
  +3.14
  a
  0xa
  0x7e
  0XA
  0X7E
  +3.14000
  +5.00000
```

◈ cout의 setf(long, long) 멤버함수를 이용한 출력양식 조정

첫번째 전달인자	두번째 전달인자	의 미
ios::fixed	ios::floatfield	고정 소수점 표기 사용
ios::scientific		과학적 표기(지수표기) 사용
ios::right	ios::adjustfield	우측정렬 사용
ios::left		좌측정렬 사용
ios::internal		부호는 좌측정렬, 값은 우측정렬

[예제1-6] cout의 setf()멤버함수의 활용(2)

```
#include<iostream>
using namespace std;
int   main(void)
{
      float f=3.14, g=150.0;

      cout.setf(ios::showpos);
      cout.setf(ios::showpoint);
      cout.precision(3);
      cout << f << "₩n";
      cout << g << "₩n₩n";

      cout.setf(ios::scientific ,ios::floatfield);

      cout << f << "₩n";
      cout << g << "₩n₩n";
      return 0;
}
```

(결과)

```
+3.14
+150.

+3.140e+000
+1.500e+002
```

2. C++에 추가된 데이터 타입 및 향상된 for문

◈ bool

참 또는 거짓을 나타내는 논리 타입

```
bool b = true;
cout << b << endl;    // 1 출력
b = false;
cout << boolalpha << b << endl;   // false 출력
cout << noboolalpha << b << endl;   // 0 출력
```

◈ auto

컴파일러에 의해 자동으로 타입이 정해지는 변수 선언

```
auto num = 7;   // 자동으로 int형으로 할당된 후 7로 초기화 됨
```

◈ decltype(expr)

expr 이 나타내는 타입과 같은 타입을 따르는 변수 선언

```
int num = 7;
decltype(num) number = 8; // number는 num의 타입을 따라 int형으로 할당됨
```

◈ 향상된 for문

배열에 대한 간략화된 구문 제공. auto와 함께 사용하면 더욱 간략화 됨

```
int data[10];
// 기존 for 루프
for(int i = 0; i < sizeof(data) / sizeof(data[0]); ++i) {
      data[i] = rand() % 100;      //data 배열에 0 ~ 99 사이 난수 대입
}
// 향상된 for 루프
for(auto t : data) {                //data[0] ~ data[9]를 차례로 t에 대입. t는 int 타입으로 추론
        cout <<   t << ' ';         //표준 출력으로 data 배열 값 전달
}
```

3. 네임스페이스 (namespace)

◈ namespace의 개념

- 여러 가지 식별자들을 이름이 있는 범위로 그룹화하는 기법
  - 식별자들의 종류 : 변수, 함수, 구조체, 클래스, 공용체, typedef 등
- 동일한 이름을 갖는 함수나 변수라도 서로 다른 namespace안에 만들 수 있다.
- 코드 내에서 이름이 같은 식별자나 라이브러리 사용하는 과정에서 생기는 식별자명 충돌 (name clash) 문제를 해결할 수 있음
- scope resolution operator ('::') 를 사용하여 네임스페이스를 구분함

  ex) 네임스페이스명 :: 식별자명

[예제1-7] namespace 예제

(lectopia.h)
```
namespace lectopia
{
	extern int number;
	void print();
}
```

(lectopia.cpp)
```
#include <iostream>
using namespace std;
namespace lectopia
{
	int number = 3;
	void print()
	{
		cout<< "lectopia print()내에서의 number = " << number  <<endl;
	}
}
```

(ioacademy.h)
```
namespace ioacademy
{
	extern int number;
	void print();
}
```

(ioacademy.cpp)
```
#include <iostream>
using namespace std;
namespace ioacademy
{
	int number = 7;
	void print()
	{
		cout<< "ioacademy print() 내에서의 number = " << number <<endl;
	}
}
```

(main.cpp)
```
#include<iostream>
using namespace std;
#include "ioacademy.h"
#include "lectopia.h"
```

```
int main()
{
	cout<< "main()에서 출력하는 lectopia 네임스페이스의 number = "
		<<lectopia::number <<endl;
	lectopia::print();
	cout<< "main()에서 출력하는 ioacademy 네임스페이스의 number = "
		<<ioacademy::number <<endl;
	ioacademy::print();
	return 0;
}
```

◈ using 지시자를 이용한 namespace의 활용

• using 지시자 : 네임스페이스내의 식별자를 네임스페이스명 없이 사용하기 위한 명령어

① using 네임스페이스명::식별자명; ← 네임스페이스 내의 하나의 식별자명만 using 하기

② using namespace 네임스페이스명; ← 네임스페이스 내의 모든 식별자를 바로 사용 하기

** using 지시어를 사용하면 네임스페이스 내의 특정 식별자를 바로 사용할 수 있다.

[예제1-8] namespace와 using 지시어

(main.cpp)

```
#include<iostream>
using namespace std;
#include "ioacademy.h"
#include "lectopia.h"
using lectopia::number;
using lectopia::print;

int main()
{
	cout<< "main()에서 출력하는 lectopia 네임스페이스의 number = "
		<< number <<endl;
	print();
	cout<< "main()에서출력하는 ioacademy 네임스페이스의 number = "
		<<ioacademy::number <<endl;
	ioacademy::print();
	return 0;
}
```

[예제1-9] namespace와 using namespace 지시어

(main.cpp)

```
#include<iostream>
using namespace std;
#include "ioacademy.h"
#include "lectopia.h"
using namespace lectopia; // using namespace ioacademy; 도 함께 하면
                          // 아래에서 namepace명을 생략할 수 없다.
int main()
{
	cout<< "main()에서 출력하는 lectopia 네임스페이스의 number = "
		<< number <<endl;
	print();

	cout<< "main()에서 출력하는 ioacademy 네임스페이스의 number = "
		<<ioacademy::number <<endl;
	ioacademy::print();
	return 0;
}
```

♣ iostream header file내의 입출력 객체의 권장 사용법

```
#include<iostream>
//using namespace std;
using std::cout;
using std::cin;
using std::endl;
int main()
{
    cout << " 편하게 출력" << endl;
}
```

그러나 일일이 모두 using지시자를 사용하는 것은 또 다른 불편을 초래한다.

```
using std::hex;
using std::oct;
using std::setprecision;
using std::fixed;
using std::scientific;
using std::setw;
        :
```

◈ namespace의 중첩 사용

• namespace 내에 다른 namespace를 선언하여 사용될 수 있다.
이때 범위결정연산자(::)를 두 번 연속 사용하게 됨.

[예제1-10] namespace의 중첩 사용 예

(other.h)

```
namespace other
{
        extern int number;
        int func();
}
```

(other.cpp)

```
namespace other
{
        int number=7;
        int func()
        {
                return number;
        }
}
```

(inner_outer.h)

```
namespace outer
{
        extern int number;
        namespace inner
        {
                extern int number;
                int func();
        }
}
```

(inner_outer.cpp)

```
namespace outer
{
        int number=5;
        namespace inner
        {
              int number=3;  // ← 이 줄을 생략하면 아래 함수에서 5를 리턴
```

```
                int func()
                {
                        return number;
                }
        }
}
```

(main.cpp)

```
#include<iostream>
using std::cout;
using std::endl;

#include "inner_outer.h"
#include "other.h"

int main()
{
    cout << "outer::inner 네임스페이스내의number 값출력: " << outer::inner::func() << endl;
    cout << "other 네임스페이스내의number 값출력: "<< other::func() <<endl;
    cout << "outer 안쪽, inner 밖의number 값출력: "<< outer::number<< endl;
    return 0;
}
```

◈ 이름 없는 namespace

- 이름 없는 namespace를 지정하면 그 내부에 바로 접근 가능하다.
- 이름 없는 namespaces내의 변수나 함수는 해당파일에서만 접근 가능(Internal Linkage) 한데 이것은 C언어에서 사용되는 static 전역변수, static 함수와 같은 역할을 한다.

[예제1-11] 이름 없는 namespace

```
#include<iostream>
using std::cout;
using std::endl;
namespace  // 해당 파일에서만 접근가능(Internal Linkage)
{
        int number=10;
        void func()
        {
                cout << "func" << endl;
        }
}
int main()
{
        cout << number << endl;
        func();
        return 0;
}
```

[예제1-12] c언어에서의 static 함수의 예

(main.c)

```
#include<stdio.h>
static void abc();
void sub();

int main()
{
        abc();
        sub();
        return 0;
}

static void abc()
{
        printf("main 함수에서 호출한 abc 함수₩n");
}
```

(sub.c)

```
#include<stdio.h>
static void abc();
void sub()
{
        abc();
        return;
}

static void abc()
{
        printf("sub()함수에서 호출한 abc 함수₩n");
}
```

4. C++에서 확장된 함수의 기능

◈ 함수 오버로딩(overloading)

- C++에서는 같은 이름의 함수를 두 개 이상 정의(구현)하여 사용하는 것
- 함수명은 같고 parameter list 부분이 달라야 함 (함수type은 상관없음)
  : parameter의 갯수, parameter 의 type, 배치 순서
  ** const 포인터와 일반 포인터는 다른 type으로 인식 됨
- 하나의 함수명으로 다양한 type에 대한 처리를 수행할 때 유용하다.
  intInput(), charInput(), dblInput()대신 input() 하나로 함수명을 통일하여 사용 함

  ex) void input(int *);
  void input(char *);
  void input(double *);

[예제1-13] 함수 오버로딩과 호출

```
1:  #include <iostream>
2:  using namespace std;
3:  void prn(char *);
4:  void prn(char *, int);
5:  void prn(char, int);
6:
7:  int main()
8:  {
9:      char str[10]="Star";
10:
11:     prn(str);   // prn(char *) 호출
12:     prn("Moon");   // prn(char *) 호출
13:     prn("Gogumi", 3);   // prn(char *, int) 호출
14:     prn('#', 7);   // prn(char, int) 호출
15:
16:     return 0;
17: }
18:
19: void prn(char *p)
20: {
21:     cout << p << endl;
22: }
23:
24: void prn(char *p, int rc)
25: {
26:     int i;
27:     for(i=0; i<rc; i++){
28:         cout << p << endl;
29:     }
30: }
31:
32: void prn(char ch, int rc)
33: {
34:     int i;
35:     for(i=0; i<rc; i++){
36:         cout << ch;
37:     }
38:     cout << endl;
39: }
```

◈ 함수 오버로딩(overloading)시의 주의 사항

• 오버로딩을 하게 되면 함수를 호출할 때 전달인자로 항상 정확한 자료형을 사용해야 할 필요가 있으나 전달인자를 각 자료형의 대표type(int, double)으로 지정할 경우 해당 대표 type으로 자동형변환 한다.

[예제1-14] 자동형변환에 의한 오버로딩 함수의 호출

```
1:    #include <iostream>
2:    using namespace std;
3:    void square(int);   // unsigned int형이나 long형이면 에러
4:    void square(double);
5:
6:    int main()
7:    {
8:          char ch='A';
9:          short sh=10;
10:         float ft=3.4f;
11:         double db=3.4;
12:
13:         square(ch);   // square(int) 호출
14:         square(sh);   // square(int) 호출
15:         square(ft);   // square(double) 호출
16:         square(db);   // square(double) 호출
17 :
18:         return 0;
19:   }
20:
21:   void square(int a)
22:   {
23:         cout << "int형으로 호출 : " << a*a << endl;
24:   }
25:
26:   void square(double a)
27: {
28:       cout << "double형으로 호출 : " << a*a << endl;
29: }
```

* 위의 square()함수의 전달인자를 short와 float으로 바꾸어 수행해 보자

(연습하기) 함수 오버로딩 기능을 구현해보자

키보드로부터 두 개의 실수값을 입력 받아 큰 값을 출력하고, 두 개의 문자열을 입력 받아 길이가 긴 문자열을 출력한다. 큰 값과 긴 문자열을 구하는 과정은 오버로딩된 함수를 호출하여 수행한다.
함수의 이름은 max로 하며 각각 큰 값과 긴 문자열을 리턴하도록 구현한다.
문자열의 경우 길이가 같으면 첫 번째 문자열을 리턴한다.

** 결과 출력은 main()함수에서 해준다.

실행결과

```
두 실수 입력 : 2.7 3.5(엔터)
큰 값 : 3.5
두 문자열 입력 : banana apple(엔터)
긴 문자열 : banana
```

◈ 기본 전달인자(default parameter)

- 함수 prototype에서 전달인자의 기본값을 정의
- 전달 인자 없이 함수를 호출하는 경우 기본값이 사용됨
- 모든 전달인자에 기본 전달인자를 준다면 전달인자 없이도 함수 호출이 가능하다.
  (기본 전달인자를 n개 사용하면 n+1가지 형태의 함수 호출 가능)
- 기본 전달인자는 함수호출부 보다 이전에 표기되는 함수선언부에서 지정해주는 것이 일반적이나 함수정의부만 사용하는 경우에는 함수정의부에서 지정한다.
  (선언부와 정의부 양쪽에 표기하면 에러발생)

[예제1-15] 기본 전달인자의 사용

```
1:    #include <iostream>
2:    using namespace std;
3:    void char_prn(char ch='#', int cnt=5);   // 기본 전달인자 사용
4:    // void char_prn(char='#', int=5);          //   매개변수 이름은 생략가능
5:
6:    int main()
7:    {
8:        char_prn();
9:        char_prn('@');
10:       char_prn('@', 10);
11:
12:       return 0;
13:   }
14:
15:   void char_prn(char ch, int cnt)   // 정의에는 기본 전달인자가 없다!
16:   {
17:       int i;
18:       for(i=0; i<cnt; i++)
19:       {
20:           cout << ch;
21:       }
22:       cout << endl;
23:   }
```

◈ 기본 전달인자(default parameter) 사용 시의 주의 사항

• 기본 전달인자를 지정하면 그 이후의 모든 매개변수는 기본 전달인자로 지정되어야 한다.

(기본 전달인자를 잘못 지정한 경우의 예)

```
void func(int a=10, int b, int c);
void func(int a=10, int b=20, int c);
void func(int a, int b=20, int c);
void func(int a=10, int b, int c=30);
```

(연습하기) 기본전달인자 구현하기

실행결과에 맞도록 str_prn함수를 선언하고 정의하여 다음 프로그램을 완성하자. str_prn함수는 하나만 작성하며 문자를 처리하는 두 번째 매개변수는 기본 전달인자를 사용한다.

```
#include <iostream>
using namespace std;
int main()
{
      char str[80];
      cout << "문장 입력 : ";
      cin.getline(str, sizeof(str));
      char ch;
      cout << "문자 입력 : ";
      cin >> ch;
      str_prn(str);
      str_prn(str, ch);
      return 0;
}
```

실행결과

```
문장 입력 : Thanks God It's Friday!(엔터)
문자 입력 : G(엔터)
Thanks God It's Friday!
Thanks
```

◈ 함수오버로딩(overloading)과 기본전달인자(default parameter) 동시 사용 시 주의 사항

• 아래와 같이 함수 오버로딩되어 있는 함수에게 default parameter를 지정하면 오류가 발생한다.

```
void prn(char *);
void prn(char *, int=10);

int   main()
{
      char str[10]="Star";
      prn(str);          ← 모호함
      prn("Moon");       ← 모호함
      prn("Gogumi",3);
      return 0;
}

void prn(char *p)
{
      cout << p << endl;
}

void prn(char *p, int rc)
{
      int i;
      for(i=0; i<rc; i++)
      {
            cout << p << endl;
      }
}
```

◈ 인라인(inline) 함수

• 인라인 함수는 함수호출로 인한 부담을 줄이고 실행속도를 개선하기 위해 C++에 추가된 문법이다.
• 인라인 함수는 함수 호출부에 함수의 코드가 직접 삽입되는 형태로 컴파일 된다.
• 함수의 크기만큼 실행파일이 커지므로 메모리에 부담이 될 수 있으므로 주로 반복문 안에서 호출되는 작은 함수에 사용하는 것이 효과적이다.
• 함수의 머리(head)부분에 inline 예약어를 붙여서 만든다.
• 인라인 함수는 호출하기 전에 반드시 먼저 정의되어 있어야 함.
(header파일 내에 정의하는 것이 좋음)
• 클래스의 선언부에서 멤버함수를 정의하면 인라인 함수로 정의됨.

인라인 함수는 매크로함수와 비슷한 기능을 수행하지만 일반적인 함수의 특징을 그대로 가지고 있다. 따라서 매크로함수에서와 같은 부작용이 없고 매개변수의 자료형도 검사할 수 있으므로 보다 정확하고 안전한 코드의 확장이 가능하다.

[예제1-16] 인라인 함수와 매크로 함수의 비교

```
#include <iostream>
using namespace std;

#define SQUARE(x) x*x   // (x)*(x) 로 수정하여 수행해보자
inline int square(int x) { return x*x; } // 인라인 함수의 정의

int main()
{
        int ires;
        ires=square(3+4);
        cout << "inline 함수를 이용한 3+4의 거듭제곱 : " << ires << endl;

        ires=SQUARE(3+4);
        cout << "매크로 함수를 이용한 3+4의 거듭제곱 : " << ires << endl;
        return 0;
}
```

5. const의 이해

◈ const의 의미

변수선언 시 자료형 앞에 const라는 명령어를 붙여서 변수에게 상수의 성격을 부여한다.
이때 const로 선언된 변수는 상수처럼 사용된다. (읽기 전용변수)
<특징>
- const변수는 반드시 선언문에서만 초기화가 가능하다
- 초기화 이후에는 실행문에서 그 값을 변경할 수 없다

◈ const의 사용 목적

• C언어에서 상수값 사용을 위해 사용되었던 #define을 대체

```
const double PI = 3.14159;
```

• 함수나 메서드의 parameter를 const로 지정하여 수정을 불가능하게 함

```
void func(const int value);
void func(const int *ptr);          //  void func(int const *ptr); 과 동일한 표현
void func(const int *const ptr);
```

• const 형태의 리턴

```
const char * sevenPass(const char * ptr)
{
        for(int i=0; i<7; i++)   ptr++;
        return ptr;
}
```

◈ 포인터 변수의 대입 가능여부 type별로 보기

```
int * = int *                   (가능)
const int * = int *             (가능)
int * = const int *             (불가능)
const int * = const int *       (가능)
```

◈ const 포인터변수와 일반 포인터변수는 오버로딩이 가능하다

컴파일러는 일반 포인터변수와 const를 사용한 포인터변수를 서로 다른 자료형으로 인식하므로 const를 사용한 것과 그렇지 않은 함수는 서로 오버로딩이 가능하다.

```
void func(char *cp);
void func(const char *cp);
int main()
{
        char *tp="tiger";
        const char *lp="lion";
        func(tp);
        func(lp);
        return 0;
}
void func(char *cp)
{
        cout << "(char *) 매개변수 : " << cp << endl;
}
void func(const char *cp)
{
```

```
        cout << "(const char *) 매개변수 : " << cp << endl;
}
```

◈ 외부변수를 const 변수로 선언 시 사용 영역이 하나의 파일로 제한된다

외부변수에 const를 사용하면 사용영역이 하나의 파일로 제한된다. 결국 같은 const변수를 여러 파일에서 사용할 수 있으며, 이 경우 보통 헤더파일로 작성하여 파일에 포함시키는 형태로 사용한다.

[예제1-17] const 외부변수의 사용영역

[헤더파일] – myhdr.h

```
1:  const int ArySize=5;
2:  int ary_sum(const int *ap);
```

[소스파일1] – a.cpp

```
1:  #include <iostream>
2:  #include "myhdr.h"
3:
4:  int main()
5:  {
6:      int ary[ArySize]={10,20,30,40,50};
7:      int sum;
8:      sum=ary_sum(ary);
9:      cout << "배열 요소의 합 : " << sum << endl;
10:     return 0;
11: }
```

[소스파일2] – b.cpp

```
1:  #include "myhdr.h"
2:
3:  int ary_sum(const int *ap)
4:  {
5:      int i, sum=0;
6:      for(i=0; i<ArySize; i++){
7:          sum+=ap[i];
8:      }
9:      return sum;
10: }
```

실행결과

```
배열 요소의 합 : 150
```

6. 참 조 (Reference)

◈ 참조변수

- 참조변수는 이미 선언되어 있는 변수에 별명(alias)을 붙이는 것이다.
- 참조변수를 만들 때에는 '&'(reference)기호를 사용한다.
- 참조변수의 자료형은 참조대상이 되는 변수의 자료형과 같아야 한다.
- 참조에 대한 모든 수정사항은 참조형 변수가 가리키고 있는 실제 변수에 반영 됨
- 참조 변수는 선언과 동시에 참조하고자 하는 변수로 초기화 해야 함
- 참조형 변수는 주소값 추출과 역참조 연산이 자동으로 수행되는 특수한 포인터 변수 임

- 일반 포인터와의 차이점
  - 참조 변수의 주소를 출력하면 참조 대상의 주소가 나옴
    (컴파일러에서 참조 변수의 메모리 접근을 차단함)
  - 초기화 이후에는 참조 대상을 변경 할 수 없음
- 일반 함수/클래스의 멤버함수의 parameter나 return type으로도 사용 가능

▶ 참조 변수 : 일반 변수의 참조형

```
data-type 일반 변수명;
data-type & 참조 변수명 = 일반 변수명;
```

포인터	참조변수

int a = 10;
int *b = &a;
*b += 20;
	int a = 10;
	int &b = a;
	b += 20;

▶ 참조 포인터 변수 : 포인터 변수의 참조형

```
data-type * 포인터 변수명;
data-type * &참조 변수명 = 포인터 변수명;
```

▶ 참조 배열 : 배열의 참조형

```
data-type   배열명[n];
data-type   (&참조배열명)[n]   = 배열명;
```

◈ 포인터변수에 대한 참조변수

• 참조변수는 비록 별명이지만 일반변수 처럼 주소연산자로 주소값을 구할 수 있고 그 값을 포인터 변수에 저장하여 참조변수의 기억공간을 가리키게 할 수 있다.

```
int dream=10;
int &effort=dream;
int *p1=&dream;
int *p2=&effort;

cout << p1 << '₩t' << p2 << endl;
cout << *p1 << '₩t' << *p2 << endl;
```

• 포인터변수도 하나의 기억공간이므로 참조변수를 만들 수 있다.

```
char message[]="Happy Christmas!";
char *santa=message;
char * &papa=santa;   // papa는 santa 포인터 변수를 참조한다.

cout << &santa << '₩t' << &papa << endl;
cout << santa << endl;
cout << papa << endl;

papa="White Christmas!";
cout << santa << endl;
```

◈ 매개변수에 사용되는 참조변수

- 참조변수는 주로 함수의 매개변수로 사용된다.
- 참조변수를 매개변수로 사용하면 C언어에서 사용되었던 포인터변수의 역할을 대신할 수 있다.
- 이렇게 매개변수를 참조변수로 사용하는 함수 호출기법을 call by reference 기법 이라 한다.

[예제1-18] 참조변수를 매개변수로 사용하는 call by reference의 예

```
1:   #include <iostream>
2:   using namespace std;
3:   void exchange(int &x, int &y);
4:
5:   int main()
6:   {
7:        int a=10, b=20;
8:
9:        cout << "바꾸기 전 a, b의 값 : ";
10:       cout << a << ' ' << b << endl;
11:
12:       exchange(a, b);
13:
14:       cout << "바꾼 후 a, b의 값 : ";
15:       cout << a << ' ' << b << endl;
16:
17:       return 0;
18:  }
19:
20:  void exchange(int &x, int &y)   // 매개변수로 참조변수를 사용
21:  {
22:       int temp;
23:
24:       temp=x;
25:       x=y;
26:       y=temp;
27:  }
```

- call by reference 기법에서 실매개변수의 값을 보호하려면 참조변수에 const를 사용한다.

```
char *name="홍길동";
int age=27;
double height=178.5;
profile_prn(name, age, height);   // 함수의 호출
void profile_prn(const char * const &np, const int &a, const double &h)
{
     cout << "이름 : " << np << endl;
     cout << "나이 : " << a << endl;
     cout << "키 : " << h << endl;
}
```

• 함수 오버로딩 시 주의가 필요하다.

```
void func(int x, int y);
void func(int &x, int &y);

int a=10, b=20;
func(a, b);  // 어떤 함수를 호출하는 것인가?
```

◈ 매개변수로 사용되는 참조변수의 const화의 장점

• 기본적으로 참조변수를 매개변수로 사용하게 되면 호출부에서 상수값을 전달하는 것이 불가능하다. 그러나 const화된 참조변수를 매개변수로 사용하면 상수값도 전달인자로 받아서 처리하는 것이 가능해진다.

[예제1-19] 참조변수를 매개변수로 사용하는 call by reference의 예

```
int sub(int &num);
int main()
{
        int res;
        res = sub(10+20); ← 에러 발생
        cout << res << endl;
        return 0;
}
int sub(int &num)
{
        return num * num;
}
```

```
int sub(const int &num);
int main()
{
        int res;
        res = sub(10+20);
        cout << res << endl;
        return 0;
}
int sub(const int &num)
{
        return num * num;
}
```

◈ 참조배열의 예

[예제1-20] 참조배열의 예

```
void prn(char (&chr)[10]);
void prn(int (&aaa)[3][4]);
int main()
{
        char str[10]="Star";
        int ary[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};

        prn(str);
        prn(ary);
        cout << "After : " << ary[2][3] << endl;
        return 0;
}
```

```
void prn(char (&chr)[10])
{
        cout << chr << endl;
        cout << sizeof(chr) << endl;
        strcpy(chr, "Dream");
}
void prn(int (&aaa)[3][4])
{
        cout << aaa[2][3] << endl;
        cout << sizeof(aaa) << endl;
        aaa[2][3]=20;
}
```

◈ 참조를 리턴하는 함수

• 리턴값의 형태를 참조형으로 선언하면 피호출함수에서 사용하던 기억공간을 호출함수에서 공유할 수 있다.

< 특징 >

함수 호출부가 L-Value로서 사용이 가능하다.

** 단, 동적메모리 할당된 기억공간을 pointer type으로 리턴하는 형태와는 완전히 다르다.

[예제1-21] 참조를 리턴하는 함수의 예

```
1:  #include <iostream>
2:  using namespace  std;
3:  int &counter();
4:  int cnt=0;
5:
6:  int main()
7:  {
8:          int res;
9:
10:         res=counter();   // 호출결과가 변수가 된다. res=cnt;
11:         cout << res << endl;
12:         counter()=10;   // cnt=10;
13:         cout << cnt << endl;
14:         counter()++;   // cnt++;
15:         cout << cnt << endl;
16:         cout << counter()+5 << endl;   // cnt+5
17:
18:         return 0;
19: }
20:
21: int &counter()
22: {
23:         cnt++;
24:         return cnt;   // 변수 자체를 리턴한다.
25: }
```

• 참조를 리턴하는 함수는 리턴된 후에도 기억공간이 계속 유지되는 변수만 리턴해야 함
(호출함수의 기억공간을 참조해서 사용하던 매개변수, 외부변수, 정적변수)
즉, 함수가 리턴되면 기억공간이 사라지는 자동변수를 리턴하면 컴파일이 가능할지라도 데이터의 안전을 보장받을 수 없다.

[예제1-22] 참조를 리턴하는 함수의 나쁜 예

```
1:  #include<iostream>
2:  using std::cout;
3:  using std::endl;
5:  int &func();
6:  int main()
7:  {
8:          int num = 7;
9:          num = func();
10:         cout << num << endl;
11:         return 0;
12:  }
13:  int &func()  // 참조를 리턴하는 함수
14:  {
15:         int a = 10;  // 자동변수
16:         return a;
17:  }
```

[예제1-23] 참조를 리턴하는 함수로 평균점수 구하기

```
1:  #include <iostream>
2:  using namespace std;
3:  struct Score{
4:          int kor, eng, mat;
5:          double avg;
6:  };
7:
8:  void input(Score &r);
9:  Score &average(Score &r);
10:
11: int main()
12: {
13:         Score s;
14:
15:         input(s);
16:         cout << "평균 점수 : ";
17:         cout << average(s).avg << endl;  // s.avg와 같다.
18:
19:         return 0;
20: }
21:
22: void input(Score &r)
23: {
24:         cout << "세 과목의 점수 입력 : ";
25:         cin >> r.kor >> r.eng >> r.mat;
26: }
27:
28: Score &average(Score &r)
29: {
30:         int tot;
31:         tot=r.kor+r.eng+r.mat;
32:         r.avg=tot/3.0;
33:
34:         return r;
35: }
```

◈ const화된 참조를 리턴하는 함수

const 화된 참조를 리턴하는 함수의 호출부는 L-value 자리에 사용하는 것이 불가능하다.

[예제 1-24] const 화된 reference type 의 함수

```
#include<iostream>
using   namespace   std;
 const int &sub(int);

 int a=5;
 int   main()
 {
          int b=10, res;
          cout << "a=" << a << "   b=" << b << endl;
          res = sub(b);
          //sub(b) = 7;
          cout << "res = " << res << endl;
          return 0;
 }
 const int &sub(int x)
 {
          cout << "x = " << x << endl;
          return a;
 }
```

7. 동적메모리 할당

▶ 동적메모리 할당이란?

프로그래밍 시에 정해진 크기의 기억공간을 미리 선언하여 놓는 것을 정적메모리 할당이라하며, 프로그램 수행도중에 필요한 메모리를 할당 받아 사용하는 것을 동적메모리 할당이라 한다.

- C++에서도 동적메모리 할당이 가능하다.
- 동적메모리 할당 & 해제 연산자 : new, delete, new [], delete []

▶ 변수 1개의 동적메모리 할당

(c++언어 code)

```
double *dp;
dp = new double;  // 동적메모리 할당
*dp = 3.5;  // 동적메모리 할당한 기억공간에 3.5 저장
cout << *dp;
cin >> *dp;  // 동적메모리 할당한 기억공간에 실수값 입력 받음
cout << *dp;
delete dp;  // 동적메모리 할당한 기억공간을 반납(미예약 영역으로 되돌림)
```

▶ 1차원 배열의 동적메모리 할당

```
int *ip;
ip = new int[cnt];
     :
delete[] ip;
```

▶ 2차원 배열의 동적메모리 할당(column 크기 고정 배열)

```
char (*cp)[10];
cp = new char[cnt][10];
          :
delete[] cp;
```

▶ new 실패 검사

```
#include<iostream>
using namespace std;
#include <new>

int main(){
        int *ip;
        int num=0;
        try{
```

```
		while(1){
			ip=new int[1000000];
			++num;
			cout << num << "번째 할당 성공" << endl;
		}
	}
	//catch(bad_alloc &ex){
	//	cout << ex.what() << endl;
	//	cout << "할당불가" << endl;
	//}
	catch(exception &ex){
		cout << ex.what() << endl;
		cout << "할당불가" << endl;
	}

	return 0;
}
```