

객체지향언어



5장 함수

학습목표

- 함수를 선언하고 사용할 수 있다.
- 인라인 함수를 설명하고 사용할 수 있다.
- 함수의 인수 전달방법을 이해하고 여러 가지 종류의 인수를 사용할 수 있다.
- 함수의 다중 정의를 설명하고 함수를 다중 정의할 수 있다.

함수의 사용

■ 함수(function)를 사용하면

- 프로그램이 이해하기 쉬움
- 프로그램을 간결하게 만들 수 있음
- 복잡한 문제를 세부적인 작은 단위로 쪼개어 처리하는 모듈화(modulation) 가능

■ 함수

- 표준 라이브러리 함수
 - 표준에서 많이 사용하는 기능을 함수로 제공하고 있음
 - 함수를 포함하는 헤더 파일을 `#include` 명령으로 포함해야 함
- 사용자 정의 함수
 - 사용자가 원할 때 언제든지 만들어 사용할 수 있음
 - 함수명과 인수를 알 수 있도록 함수의 원형을 미리 정의해야 함

함수의 사용

■ 함수의 원형(prototype) 선언

```
반환_자료형 함수_이름(인수_자료형, 인수_자료형 ...);
```

■ 함수의 정의

```
반환_자료형 함수_이름(인수_자료형 가인수, 인수_자료형 가인수 ... )  
{  
    함수의 동작이 구현되는 부분  
    ...  
    return 반환하는_값;  
}
```

■ 함수의 호출

```
함수_이름(실인수, 실인수);
```

함수의 사용

■ 함수의 사용 예

// add 함수의 원형 선언

```
int add(int, int);
```

// 함수의 호출

```
add(20, 10);
```

// 20, 10은 실인수

// add 함수의 정의

```
int add(int a, int b)
```

```
{
```

```
    int sum;
```

```
    sum = a + b;
```

```
    return sum;
```

```
}
```

// 함수의 머리 부분, a, b는 가인수

// '{'과 '}' 사이는 함수의 몸체 부분

// 반환하는 값

함수의 인수 및 반환 값

■ 함수의 인수

- 자료를 함수에게 전달하는 것으로 인수 혹은 매개변수(parameter or argument)라고 함
- 실인수(real parameter)
 - 함수를 호출할 때의 인수
- 가인수(formal parameter)
 - 함수를 정의하는 부분의 인수

■ 함수의 반환 값

- 함수를 호출했을 때 처리 결과로 돌아오는 값
- 함수 정의에서 반환 값은 return 다음에 써 명시함
- 반환 값이 없다면 함수 정의에서 반환 자료형에 void 라고 명시함

함수의 인수 및 반환 값

■ [예]

```
void main()                // 반환 값이 없다(void).
{
    int a = 15, b = 10, sum;
    sum = add(a, b);        // sum과 add 함수의 자료형이 일치해야 한다.
    cout << a << " + " << b << " = " << sum << endl;
    multiply(a, b);
    return;                // 반환 값이 없다.
}

int add(int x, int y)      // 반환 값이 정수형(int)이다.
{
    int sum;
    sum = x + y;
    return sum;            // 반환 값이 정수형이다.
}

void multiply(int x, int y) // 반환 값이 없다.
{
    cout << x << " * " << y << " = " << x * y << endl;
}
```

함수의 인수 및 반환 값

■ ex5_1.cpp (1) (함수의 사용 예)

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

int add(int, int);           // 함수의 원형 선언
void multiply(int, int);     // 함수의 정의가 호출 뒤에 있으므로 선언이 필요

void main(void)
{
    int a = 15, b = 10, sum;
    sum = add(a, b);         // 함수의 호출 - a, b는 실인수
    cout << a << " + " << b << " = " << sum << endl;
    multiply(a, b);          // 함수의 호출 - a, b는 실인수

    return;
}
```


■ ex5_1.cpp (2) (함수의 사용 예)

```
int add(int x, int y)      // 함수의 정의 - x, y는 가인수
{
    int sum;
    sum = x + y;

    return sum;
}
```

```
void multiply(int x, int y) // 함수의 정의 - x, y는 가인수
{
    cout << x << " * " << y << " = " << x * y << endl;
}
```

```
15 + 10 = 25
15 * 10 = 150
계속하려면 아무 키나 누르십시오 . . .
```

인라인 함수

■ 인라인(inline) 함수

- 인라인 함수는 컴파일 할 때 목적 코드 수준에서 대체
- 함수 호출에 의한 오버헤드를 없앨 수 있어 실행 속도가 개선

```
inline 반환_자료형 함수명(자료형 인수, ...) { 함수_본문; }
```

■ 매크로 함수

- 인수를 갖는 매크로
- 선행처리자에 의해 소스 코드 수준에서 치환
- 매크로 함수와 인라인 함수는 함수를 찾아가서 수행하고 돌아오는 오버헤드를 줄일 수 있으므로 효율적임
- 함수의 내용이 길어지면 코드 양이 많아지므로 일반 함수로 작성하는 것이 좋음

인라인 함수

■ ex5_2.cpp (1) (매크로 함수, 인라인 함수, 일반 함수의 사용)

```
#include <iostream>
```

```
using namespace std;
```

```
#define abs1(x) (x<0) ? -(x) : (x)           // 매크로 함수
```

```
inline int abs2(int y) {                     // 인라인 함수
```

```
    int m;
```

```
    m = y < 0 ? -y : y;
```

```
    return m;
```

```
}
```

```
int abs3(int z) {                             // 일반 함수
```

```
    int m;
```

```
    m = z < 0 ? -z : z;
```

```
    return m;
```

```
}
```

인라인 함수

■ ex5_2.cpp (2) (매크로 함수, 인라인 함수, 일반 함수의 사용)

```
int main(int argc, char* argv[])
```

```
{
```

```
    int a = -15, m1, m2, m3;
```

```
    m1 = abs1(a);
```

```
    cout << "(매크로 함수) a = " << a << ", m1 = " << m1 << endl;
```

```
    m2 = abs2(a);
```

```
    cout << "(인라인 함수) a = " << a << ", m2 = " << m2 << endl;
```

```
    m3 = abs3(a);
```

```
    cout << "(일반 함수) a = " << a << ", m3 = " << m3 << endl;
```

```
    return 0;
```

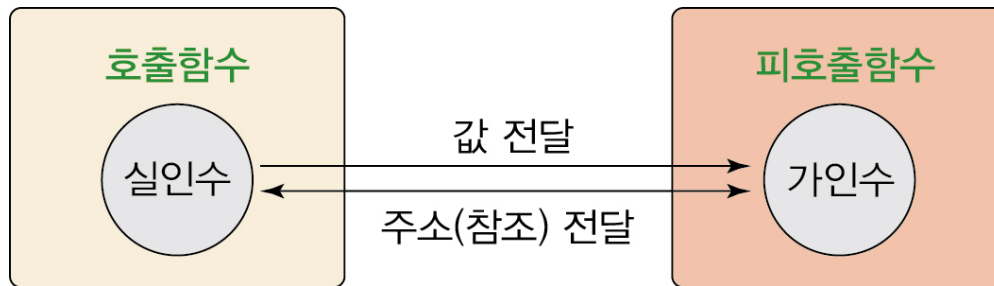
```
}
```

```
(매크로 함수) a = -15, m1 = 15  
(인라인 함수) a = -15, m2 = 15  
(일반 함수) a = -15, m3 = 15  
계속하려면 아무 키나 누르십시오 . . .
```

함수에 인수 전달 방식

■ C++에서 함수에게 사용될 데이터를 보내는 방법

- (1) 값에 의한 전달 방식(call by value)
- (2) 주소에 의한 전달 방식(call by address)
- (3) 참조에 의한 전달 방식(call by reference)



- 값 전달에서는 실인수의 값이 가인수로 전달되고 실인수는 영향 받지 않음
- 주소(참조) 전달에서는 실인수의 주소가 전달되고 실인수도 영향을 받음

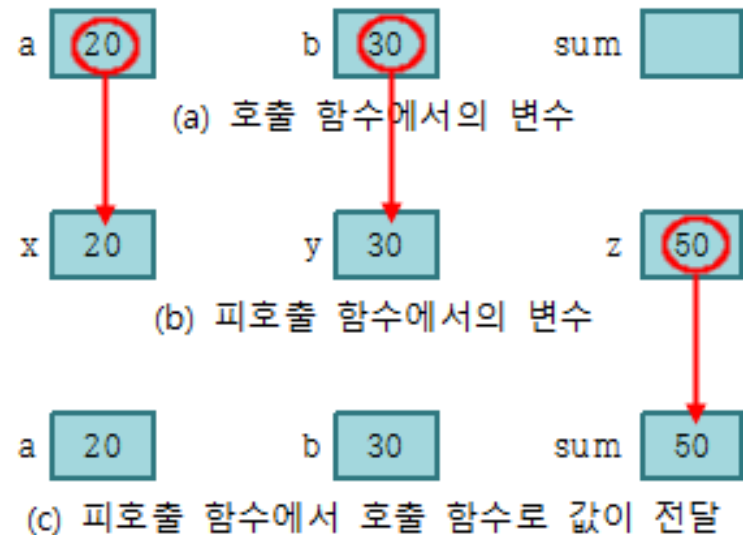
자료 전달 방식	함수의 선언	함수의 호출	함수 간의 독립성
값에 의한 전달	<code>swap(int x, int y)</code>	<code>swap(a, b)</code>	독립성 보장
주소에 의한 전달	<code>swap(int *x, int *y)</code>	<code>swap(&a, &b)</code>	독립성 보장 안됨
참조에 의한 전달	<code>swap(int &x, int &y)</code>	<code>swap(a, b)</code>	독립성 보장 안됨

함수에 인수 전달 방식

■ 값에 의한 전달(call by value)

- 일반적인 인수 전달 방식
- 실인수는 변수가 아닌 실제 값을 전달하고,
- 가인수는 실인수와 별개의 기억장소를 할당 받아 실인수의 값을 복사
- 피호출 함수에서 인수 값이 변경되어도 호출 함수에는 영향을 받지 않아 호출 함수와 피호출 함수가 독립성을 유지함

```
void main()  
{  
    int a = 20, b = 30, sum;  
    sum = add(a, b); // 호출 함수  
}  
  
int add(int x, int y) // 피호출 함수  
{  
    int z;  
    z = x + y;  
    return z;  
}
```



함수에 인수 전달 방식 (3)

■ ex5_3.cpp (값에 의한 전달의 예)

```
#include <iostream>
using namespace std;
int add(int, int);           // add() 함수 선언
void main()
{
    int a = 20, b = 30, sum;
    sum = add(a, b);         // a와 b의 값 20과 30이 전달된다.
                             // add() 함수에서 반환된 값이 sum에 대입된다.
    cout << a << " + " << b << " = " << sum << endl;
}
int add(int x, int y)       // 20과 30이 x와 y에 복사된다.
{
    int z;
    z = x + y;
    return z;               // z의 값을 반환한다.
}
```

20 + 30 = 50

계속하려면 아무 키나 누르십시오 . . .

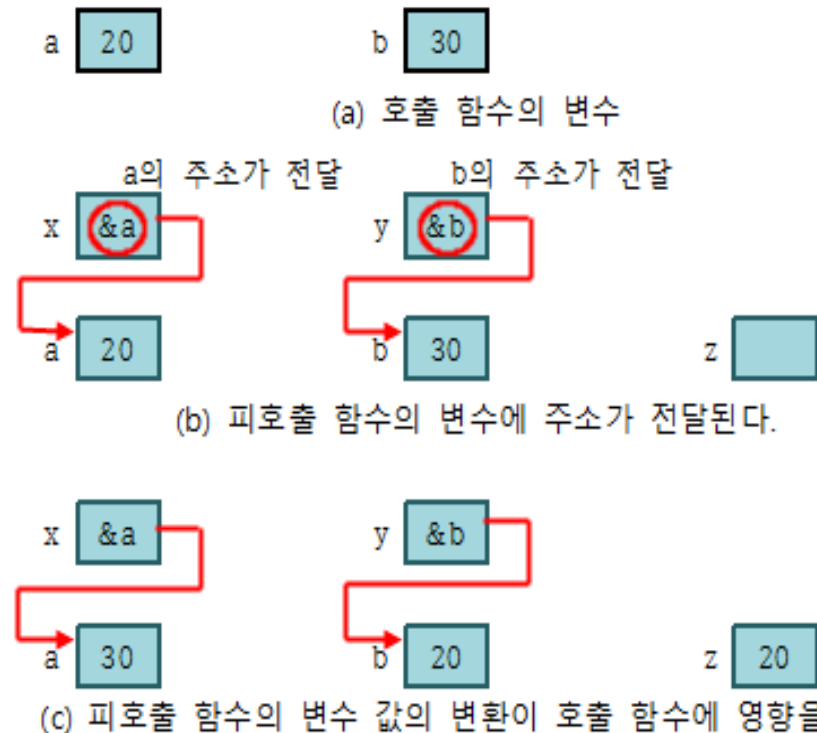
함수에 인수 전달 방식

■ 주소에 의한 전달 (call by address)

- 실인수는 변수의 주소를 전달하고, 가인수(포인터 변수)에는 전달된 주소가 복사
- 포인터는 실인수의 값을 가리키고 값을 변경하면 호출한 함수에서도 값이 바뀜
- 두 함수 사이에 독립성이 유지되지 않으므로 주소에 의한 전달 방식을 사용할 때는 값의 변경에 주의하여야 함

```
void main()
{
    int a = 20, b = 30;
    swap(&a, &b);    // 호출 함수
}

void swap(int *x, int *y) // 피호출 함수
{
    int z;
    z = *x;
    *x = *y;
    *y = z;
}
```



함수에 인수 전달 방식 (6)

- ex5_4.cpp (주소에 의한 전달 방식으로 swap() 함수)

```
#include <iostream>
using namespace std;
void swap(int *, int*);      //swap() 함수 선언
void main()
{
    int a = 20, b = 30;
    cout << "(swap 이전) a = " << a << ", b = " << b << endl;
    swap(&a, &b);            //swap() 함수 호출
    cout << "(swap 이후) a = " << a << ", b = " << b << endl;
}
void swap(int *x, int *y)    //swap() 함수 정의
{
    int z;
    z = *x;
    *x = *y;
    *y = z;
}
```

```
<swap 이전> a = 20, b = 30
<swap 이후> a = 30, b = 20
계속하려면 아무 키나 누르십시오 . . .
```

함수에 인수 전달 방식

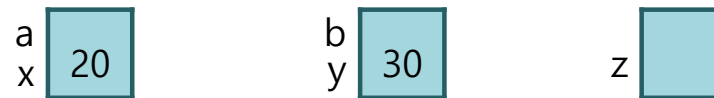
■ 참조에 의한 전달 방식(call by reference)

- 참조는 변수를 참조하는 것으로 변수의 별명
- 참조에 의한 전달 방식은 주소에 의한 전달 방식과 같은 효과를 내지만, 사용방법은 값에 의한 전달 방식과 비슷하여 사용하기가 편리함
- 인수를 직접 다루어야 하는 경우에 참조를 이용할 수 있음

```
void main()
{
    int a = 20, b = 30;
    swap(a, b);    // 호출 함수
}
```

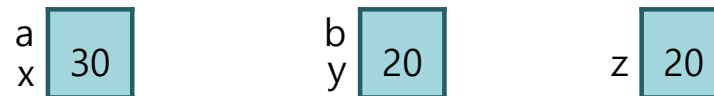


(a) 호출 함수에서 변수



(b) 피호출 함수에서 호출 함수의 변수를 참조한다.

```
// 피호출 함수
void swap(int &x, int &y) {
    int z;
    z = x;
    x = y;
    y = z;
}
```



(c) 피호출함수에서 변수 값의 변환이 호출 함수에 영향을 준다.

함수에 인수 전달 방식 (6)

- ex5_5.cpp (참조에 의한 전달 방식으로 swap() 함수)

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

void swap(int &, int&);

void main()
{
    int a = 20, b = 30;

    cout << "(swap 이전) a = " << a << ", b = " << b << endl;
    swap(a, b);
    cout << "(swap 이후) a = " << a << ", b = " << b << endl;
}

void swap(int &x, int &y)
{
    int z;
    z = x;
    x = y;
    y = z;
}
```

```
<swap 이전> a = 20, b = 30
<swap 이후> a = 30, b = 20
계속하려면 아무 키나 누르십시오 . . .
```

함수에 인수 전달 방식

- 상수(constant)에 이름을 주는 방법
 - 매크로 상수
 - #define PI 3.141592 와 같이 전처리 지시어를 사용
 - const 상수
 - const 키워드로 상수를 선언
 - const 상수를 다른 곳에서 변경하려고 하면 컴파일 오류를 보낸다.

```
const 자료형 변수명 = 상수_값;
```

```
#define PI 3.14159          // 매크로 상수 선언
const double pi = 3.14159;  // const 상수 선언
```

함수에 인수 전달 방식 (10)

- ex5_6.cpp (상수에 이름을 주는 방법의 예)

```
#include <iostream>
using namespace std;
#define PI 3.14159          // 매크로 상수 선언
const double pi = 3.14159;  // const 상수 선언

void main()
{
    int r = 10;
    double s;

    s = pi * r * r;
    cout << "(const 상수 사용) area = " << s << endl;

    s = PI * r2 * r2;
    cout << "(매크로 상수 사용) area = " << s << endl;
}
```

```
<const 상수 사용> area = 314.159
<매크로 상수 사용> area = 314.159
계속하려면 아무 키나 누르십시오 . . .
```

함수에 인수 전달 방식

■ 참조에 의한 전달 방식의 문제

- 참조에 의한 전달 방식은 값에 의한 전달보다 전달 속도가 빠르고,
- 주소에 의한 전달보다 간단하고 안정적임
- 그러나 함수 간의 독립성 보장 차원에서 문제가 있을 수 있음
- 함수의 인수로 사용한 변수 값이 변경되는 것에 주의해야 함
- 이러한 문제를 해결하기 위해 참조를 `const` 선언하여 상수 형태로 사용하는 것이 좋음 ➔ 상수형 참조

■ 참조의 반환

- 함수가 참조를 반환할 수도 있음
- 참조의 반환은 함수를 대입문의 왼쪽에 사용할 때 사용할 수 있음

함수에 인수 전달 방식

- ex5_7.cpp (const 인수를 사용한 참조에 의한 전달)

```
#include <iostream>
using namespace std;
int add(const int &, const int &);
void main()
{
    int a = 20, b = 30, sum;

    sum = add(a, b);
    cout << "a + b = " << sum << endl;
}
int add(const int &x, const int &y)
{
    int s;

    s = x + y;
    return s;
}
```

a + b = 50
계속하려면 아무 키나 누르십시오 . . .

함수에 인수 전달 방식


■ ex5_8.cpp (참조를 반환)

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

```
int x;
```

```
int &Set_x()
{
    return x;
}
```

```
void main()
{
    cout << "x = " << x << endl;
    Set_x() = 1000;
    cout << "x = " << x << endl;
}
```



```
x = 0
x = 1000
계속하려면 아무 키나 누르십시오 . . .
```


함수에 인수 전달 방식

■ 세 가지 전달 방식의 비교

자료 전달 방식	함수의 선언	함수의 호출	함수 간의 독립성
값에 의한 전달	<code>swap(int x, int y)</code>	<code>swap(a, b)</code>	독립성 보장
주소에 의한 전달	<code>swap(int *x, int *y)</code>	<code>swap(&a, &b)</code>	독립성 보장 안됨
참조에 의한 전달	<code>swap(int &x, int &y)</code>	<code>swap(a, b)</code>	독립성 보장 안됨

함수의 다중 정의

■ C에서는 함수 다중정의(function overloading)를 할 수 없음

- 같은 이름을 사용할 수 없고 다른 이름을 사용해야 함

```
int intAdd(int x, int y);           // int형의 덧셈  
double doubleAdd(double x, double y); // double형의 덧셈
```

■ C++에서는 함수의 다중정의가 가능

- 다형성(polymorphism)을 제공
- 함수의 인수 자료형이나 인수 개수가 다르면 함수의 이름을 같이 할 수 있음
- 컴파일러가 인수의 자료형을 보고 판단하여 해당 함수를 호출함

```
int Add(int x, int y);           // int형의 덧셈  
double Add(double x, double y); // double형의 덧셈
```

함수의 다중 정의

■ ex5_9.c (1) (C 프로그램에서 정수의 더하기와 실수의 더하기)

```
#include <stdio.h>
```

```
int intAdd(int x, int y) {  
    return (x + y);  
}
```

```
double doubleAdd(double x, double y) {  
    return (x + y);  
}
```

```
void main()  
{
```

```
    int a = 8, b = 24, c;  
    double e = 5.8, f = 28.7, g;
```

```
    c = intAdd(a, b);  
    printf("%d + %d = %d \n", a, b, c);
```

```
    g = doubleAdd(e, f);  
    printf("%f + %f = %f \n", e, f, g);
```

```
}
```



```
8 + 24 = 32  
5.800000 + 28.700000 = 34.500000  
계속하려면 아무 키나 누르십시오 . . .
```

함수의 다중 정의

■ ex5_10.cpp (1) (C++에서 함수의 다중 정의의 예)

```
#include <iostream>
using namespace std;
int Add(int x, int y) {
    cout << "int Add(int x, int y) 호출 ==> " ;
    return (x + y) ;
}
double Add(double x, double y) {
    cout << "double Add(double x, double y) 호출 ==> " ;
    return (x + y) ;
}
void main() {
    int a = 8, b = 24, c;
    double e = 5.8, f = 28.7, g;

    c = Add(a, b);
    cout << a << " + " << b << " = " << c << endl;

    g = Add(e, f);
    cout << e << " + " << f << " = " << g << endl;
}
```

```
int Add<int x, int y> 호출 ==> 8 + 24 = 32
double Add<double x, double y> 호출 ==> 5.8 + 28.7 = 34.5
계속하려면 아무 키나 누르십시오 . . .
```

함수의 다중 정의

■ 함수의 용법(signature)

- 함수의 다중정의에서 함수를 구분할 수 있는 조건
- 함수를 다중 정의할 때는 각각의 함수의 용법이 서로 달라야 함
- 함수의 용법을 구분하는 기준
 - 인수의 개수가 다르거나,
 - 인수의 자료형이 달라야 함. 인수의 개수가 같더라도 자료형이 다르면 용법이 다른 것임
 - 반환 자료형이나 인수의 이름은 용법 구분에 사용하지 않음
 - const나 참조(&)도 용법을 구분하는데 사용되지 않음

함수의 다중 정의

■ 함수의 용법(signature)의 예 (1)

- (1) 반환 자료형이 다르더라도 인수의 개수와 자료형이 같으면 다중 정의될 수 없다.

```
int Func(int x, int y);
```

```
void Func(int x, int y);
```

- (2) 인수의 이름만 다를 경우 다중 정의될 수 없다.

```
void Func(int x, int y);
```

```
int Func(int a, int b);
```

함수의 다중 정의

■ 함수의 용법(signature)의 예 (2)

(3) const나 참조(&)도 용법을 구분하는데 사용되지 않는다.

참조형인 경우 함수를 호출할 때 같은 모양이므로 용법을 구분할 수가 없으므로 다중 정의될 수 없다.

```
int Func(int x, int y);
```

```
int Func(const int x, const int y);
```

```
int Func(int& x, int& y);
```

(4) 인수의 자료형이 int형과 int *, double형과 double *로 자료형이 다르면, 다중 정의될 수 있다.

```
int Func(int x, int y);
```

```
int Func(int* x, int* y);
```

```
double Func(double x, double y);
```

```
double Func(double* x, double* y);
```

함수의 다중 정의 (7)

■ ex5_11.cpp (1) (함수의 용법의 예)

```
#include <iostream>
using namespace std;
int Add(int x, int y) {
    cout << "int Add(int x, int y) 호출==> " ;
    return (x + y);
}
int Add(int *x, int *y) {
    cout << "int Add(int *x, int *y) 호출==> " ;
    return (*x + *y);
}
double Add(double x, double y) {
    cout << "double Add(double x, double y) 호출==> " ;
    return (x + y);
}
double Add(double* x, double* y) {
    cout << "double Add(double* x, double* y) 호출==> " ;
    return (*x + *y);
}
```


함수의 다중 정의 (9)

■ ex5_11.cpp (3) (함수의 용법의 예)

```
void main()
{
    int a = 8, b = 24, c;
    double r = 20.5, s = 12.8, t;

    c = Add(a, b);
    cout << a << " + " << b << " = " << c << endl;
    c = Add(&a, &b);
    cout << a << " + " << b << " = " << c << endl;
    t = Add(r, s);
    cout << r << " + " << s << " = " << t << endl;
    t = Add(&r, &s);
    cout << r << " + " << s << " = " << t << endl;
}
```

```
int Add<int x, int y> 호출==> 8 + 24 = 32
int Add<int *x, int *y> 호출==> 8 + 24 = 32
double Add<double x, double y> 호출==> 20.5 + 12.8 = 33.3
double Add<double* x, double* y> 호출==> 20.5 + 12.8 = 33.3
계속하려면 아무 키나 누르십시오 . . .
```

함수의 다중 정의

■ 인수의 기본 값

- 함수의 가인수에 지정한 초기 값은 그 인수의 기본 값(default value)이 됨
- 함수를 호출할 때 초기화 된 인수가 생략되면 기본 값으로 지정됨
- 함수 호출에서 같은 이름을 사용하면서도 인수의 개수가 다르게 보이므로 함수를 다중 정의한 것처럼 보임
- 함수의 가인수를 초기화하여 기본 값을 지정할 때 초기화된 인수는 다른 인수들보다 뒤에 와야 함
- 함수의 원형 선언과 그 함수의 정의 중에 한번만 초기화하여야 함
- 함수의 호출 이전에 기본 값이 지정되어 있어야 함

[예]

```
int Add(int x=1, int y) { ... } // 초기화된 인수가 앞에 있으므로 에러  
int Add(int x, int y=1) { ... } // 인수의 초기화가 올바르게 사용되었다.
```

```
Add(10, 20);           // x = 10, y = 20으로 호출  
Add(10);                // x = 10, y = 1 로 호출
```

함수의 다중 정의 (11)

■ ex5_12.cpp (1) (함수의 가인수에 기본 값을 준 예)

```
#include <iostream>
using namespace std;
int Add(int x, int y = 1); // 함수의 원형 선언
void main()
{
    int sum;

    cout << "인수가 10, 20 으로 전달한 경우" << endl;
    sum = Add(10, 20);
    cout << "sum = " << sum << endl << endl;

    cout << "인수가 10 으로 전달한 경우" << endl;
    sum = Add(10);
    cout << "sum = " << sum << endl << endl;
}
int Add(int x, int y)    // 함수의 정의
{
    cout << "x = " << x << ", y = " << y << endl;
    return (x + y);
}
```

```
인수가 10, 20 으로 전달한 경우
x = 10, y = 20
sum = 30
```

```
인수가 10 으로 전달한 경우
x = 10, y = 1
sum = 11
```

```
계속하려면 아무 키나 누르십시오 . . . _
```



Thank You
