

---

# C/C++ Programming

---

Chungbuk National University, Korea  
Intelligent Robots Lab. (IRL)

Prof. Gon-Woo Kim

# C++ 프로그램 구성

basic.c

c 컴파일

```
#include <stdio.h>

int g=20; /* 전역 변수 */

int add(int x, int y) { /* 전역 함수 */
    return x + y;
}

int main() {
    int a, b, sum; /* 지역 변수 */
    scanf("%d", &a, &b); /* 입력 */
    sum = a + b;
    printf("%d", sum); /* 출력 */
    return 0;
}
```

2 5  
7

키 입력

basic.cpp

C++ 컴파일

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

int g=20; /* 전역 변수 */

int add(int x, int y) { // 전역 함수
    return x + y;
}

int main() {
    int a, b, sum; // 지역 변수
    cin >> a >> b; // 입력
    sum = a + b;
    cout << sum; // 출력
    return 0;
}
```

2 5  
7

키 입력

# 기본적인 구성에 있어서 C/C++ 비교

## □ 비교

- ▣ #include 사용 - 헤더 파일 첨부
- ▣ 변수와 변수 선언 - 변수 타입과 선언 방법 동일
- ▣ 함수 구성 및 함수 호출 - 함수 작성과 호출 방법 동일
- ▣ main() 함수 - 프로그램 실행 시작. main() 함수의 원형 동일
- ▣ 연산자 - C++는 C 언어의 연산자를 그대로 수용
- ▣ 전역 변수와 지역 변수 - C/C++ 동일

## □ 변경 추가

- ▣ 주석문 - /\* \*/에 한 줄짜리 주석(//) 추가
- ▣ 표준 입출력 헤더 파일 - <stdio.h>에 <iostream> 추가.
  - C++에서는 2003년부터 헤더 파일에 .h 사용하기 않음
- ▣ 표준 입출력 방법 - C 표준입출력 함수 scanf/printf에 cin/cout 객체 추가

## □ C++ 데이터 타입 - bool 추가

- ▣ void, char, int, short int, long, float, double, bool
- ▣ bool 타입 추가
  - bool 타입의 상수 true, false 사용 가능

# 예제 기본 C++ 프로그램

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

int g=20; /* 전역 변수 */

int add(int x, int y) { // 전역 함수
    return x + y; // x와 y의 합 리턴
}

int main() {
    int a, b, sum; // 지역 변수
    cout << "두 정수를 입력하세요 >>"; // 프롬프트 출력
    cin >> a >> b; // 두 정수를 읽어 a와 b에 입력
    sum = a + b;
    cout << "합은 " << sum << "\n"; // sum 값 출력
    cout << "합은 " << add(a, b) << "\n"; // add() 함수 호출 결과 출력
    cout << "전역 변수 g 값은 " << g; // g 값 출력

    return 0; // return 문을 생략하면 자동으로 return 0;이 삽입된다.
}
```

두 정수를 입력하세요 >>33 55  
합은 88  
합은 88  
전역 변수 g 값은 20

키 입력

# 조건문

- C++는 C 언어의 다음 2가지 유형의 조건문 그대로 사용
  - ▣ if, if-else, if-else if-else
  - ▣ switch

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

int main() {
    int score;
    cout << "점수를 입력하세요>>";
    cin >> score;
    if(score > 100 || score < 0) {
        cout << "잘못된 점수 입니다";
        return 0;
    }
    if(score >= 90) // 90이상 100이하
        cout << "A 입니다";
    else if(score >= 80) // 80이상 89이하
        cout << "B 입니다";
    else if(score >= 70) // 70이상 79이하
        cout << "C 입니다";
    else if(score >= 60) // 60이상 69이하
        cout << "D 입니다";
    else // 0이상 59이하
        cout << "F 입니다";
}
```

# 반복문

---

- C++는 C 언어의 다음 3가지 반복문을 그대로 사용
  - ▣ for, while, do-while
  - ▣ 추가된 것 없음
- break
  - ▣ 반복문을 벗어남
- continue
  - ▣ 다음 반복 실행

# 예제 for 문 예제

두 정수 a, b를 입력 받아 a에서 b까지의 정수 합을 출력하라. 반드시 작은 수, 큰 수 순서로 입력하라.

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

int main() {
    int i, a, b, sum=0;
    cout << "두 개의 정수 입력>>";
    cin >> a >> b;

    for(i=a; i<=b; i++) { // a에서 b까지 합 계산
        sum += i;
    }

    cout << a << "에서 " << b << "까지 합은 " << sum;
}
```

두 개의 정수 입력>>3 6  
3에서 6까지 합은 18

# 예제 while 문 예제

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

int main() {
    int i, a, b, sum=0;
    cout << "두 개의 정수 입력>>";
    cin >> a >> b;

    i=a;
    while(i<=b) { // i가 b보다 작거나 같은 동안 반복
        sum += i;
        i++;
    }

    cout << a << "에서 " << b << "까지 합은 " << sum;
}
```

두 개의 정수 입력>>3 6  
3에서 6까지 합은 18



# 예제 continue와 break 문

while 문과 continue, break 문을 이용하여, 정수를 입력 받고 3의 배수이면 "Yes"를, 아니면 "No"를 출력하라. 0이 입력되면 프로그램을 종료한다.

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

int main() {
    int a;
    while(true) {
        cout << "정수 입력>>";
        cin >> a;
        if(a == 0)
            break; // 0이 입력되면 while 문을 벗어남
        if(a%3 != 0) {
            cout << "No" << "\n";
            continue; // 다음 반복. while 문으로 분기
        }
        cout << "Yes" << "\n"; // 입력된 3의 배수 출력
    }
}
```

```
정수 입력>>381
Yes
정수 입력>>297
Yes
정수 입력>>235
No
정수 입력>>0
```

# 배열

## □ 배열이란

- 동일 타입의 데이터를 하나의 단위로 다루기 위해 연결된 메모리
- C++는 C 언어의 배열 그대로 사용

## □ 배열 선언

```
int n[10]; // 정수 10개짜리 빈 메모리 공간
double d[] = {0.1, 0.2, 0.5, 3.9}; // d의 크기는 자동으로 3으로 설정.
// 배열 d에 순서대로 0.1, 0.2, 0.5, 3.9로 초기화
```

n[0] n[1] n[2] n[3] n[4] n[5] n[6] n[7] n[8] n[9]

배열 n

--	--	--	--	--	--	--	--	--	--

d[0] d[1] d[2] d[3]

배열 d

0.1	0.2	0.5	3.9
-----	-----	-----	-----

```
n[10] = 20; // 인덱스 10 사용 오류. 인덱스는 0~9까지만 사용 가능
d[-1] = 9.9; // 인덱스 -1 사용 오류. 인덱스로 음수 사용 불가
```

# 2차원 배열

```
int m[2][5]; // 2행 5열의 2차원 배열 선언
```

배열 m

m[0][0]	m[0][1]	m[0][2]	m[0][3]	m[0][4]
m[1][0]	m[1][1]	m[1][2]	m[1][3]	m[1][4]

m[2][0] = 5; // 오류. 인덱스 2가 잘못 사용되었음. 0~1만 가능  
m[0][6] = 2; // 오류. 인덱스 6이 잘못 사용되었음. 0~4만 가능

# 예제 배열 선언 및 활용

배열을 선언하고 배열을 활용하는 코드를 사례를 보인다.

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

int main() {
    int n[10]; // 정수 10개짜리 빈 메모리 공간
    double d[] = {0.1, 0.2, 0.5, 3.9}; // 배열 d에 0.1, 0.2, 0.5, 3.9로 초기화

    int i;
    for(i=0; i<10; i++) n[i] = i*2; // 2의 배수로 n에 값을 채움
    for(i=0; i<10; i++) cout << n[i] << ' '; // 배열 n 출력
    cout << "\n"; // 한 줄 띄다.

    double sum = 0; // C++에서는 필요할 때 변수를 아무 곳이나 선언 가능
    for(i=0; i<4; i++) { // 배열 d의 합 계산
        sum += d[i];
    }

    cout << "배열 d의 합은 " << sum; // 배열 d의 합 출력
}
```

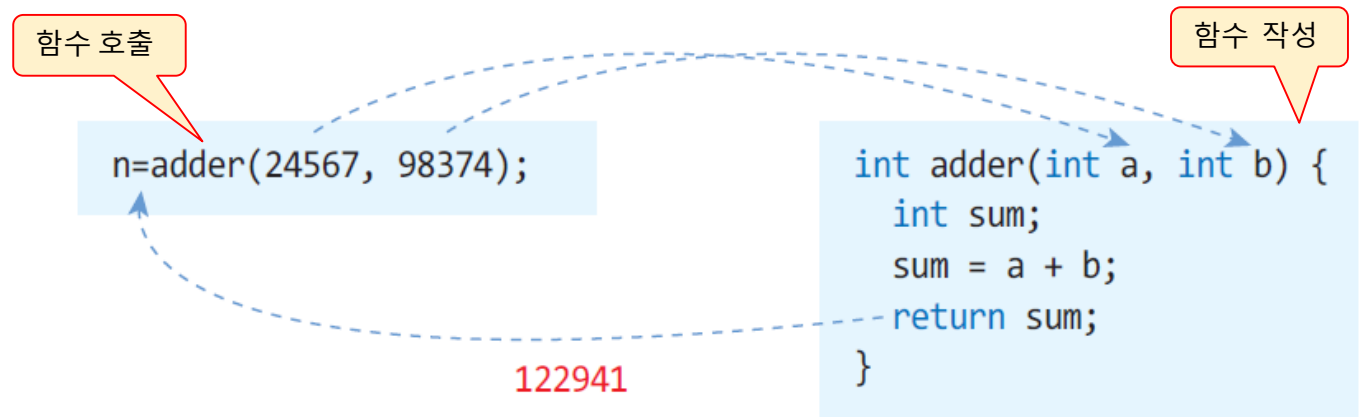
0 2 4 6 8 10 12 14 16 18  
배열 d의 합은 4.7

# 함수의 구성과 함수 호출

## □ 함수란

- ▣ 매개변수를 통해 데이터를 전달받아 처리한 후 결과를 리턴하는 코드 블록
- ▣ C++는 C 언어의 함수 기법 그대로 계승

```
리턴타입 함수이름(매개변수 리스트) {  
    계산하는 프로그램 코드들  
    결과를 리턴하는 return 문  
}
```



# 예제 adder() 함수와 호출

두 개의 정수를 전달받아 합을 리턴하는 함수 adder()를 작성하라.

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

// 두 개의 정수를 받아 합을 구하고 결과를 리턴하는 함수 adder
int adder(int a, int b) {
    int sum;
    sum = a + b;
    return sum;
}

int main() {
    int n = adder(24567, 98374); // 함수 adder() 호출
    cout << "24567과 98374의 합은 " << n << "입니다\n";

    int a, b;
    cout << "두 개의 정수를 입력하세요>>";
    cin >> a >> b;
    n = adder(a, b); // 함수 adder() 호출
    cout << a << "와 " << b << "의 합은 " << n << "입니다\n";
}
```

24567과 98374의 합은 122941입니다  
두 개의 정수를 입력하세요>>2342 158619  
2342와 158619의 합은 160961입니다

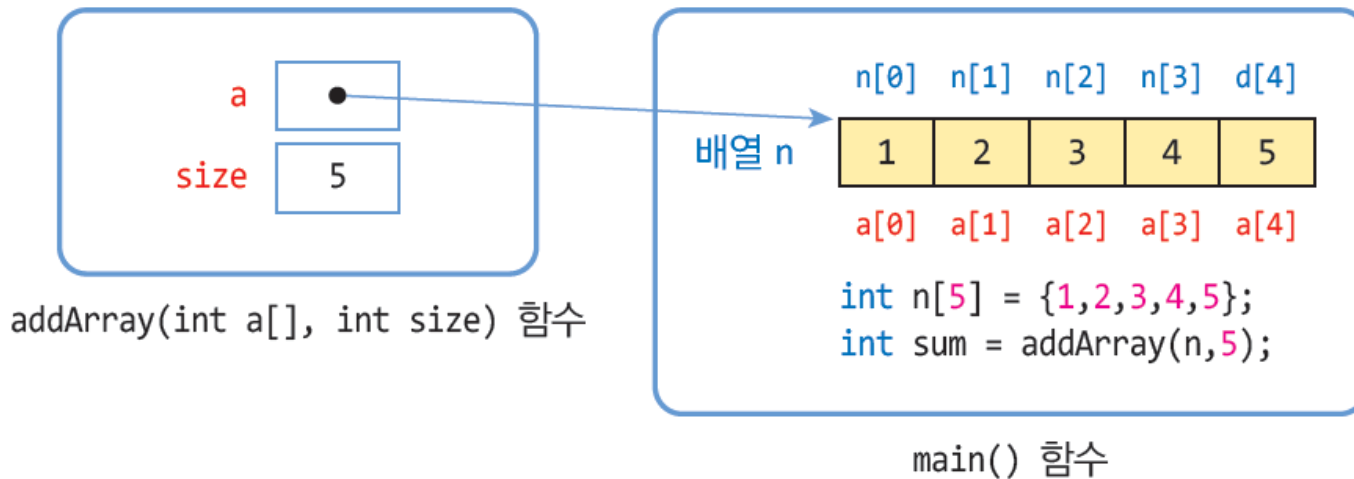
# 매개 변수로 배열 전달

## □ 함수의 매개 변수로 배열 전달

```
int addArray(int a[], int size) { // 배열을 매개 변수로 가진 함수
...
}
```

```
int n[5] = { 1,2,3,4,5 };
int s = addArray(n, 5); // 배열 n을 매개 변수로 전달
int m[3] = { 1,2,3 };
int t = addArray(m, 3); // 배열 m을 매개 변수로 전달
```

addArray() 호출



# 포인터

## □ 포인터란?

- 포인터(pointer)는 실행 중 메모리의 주소 값
- 주소(포인터)를 이용하여 메모리에 직접 값을 쓰거나 메모리로부터 값을 읽어올 수 있음

## □ 변수와 메모리 주소

```
int n;  
n = 3;
```

- 변수 n은 정수를 저장할 메모리 공간에 대한 이름, 이곳에 3 기록
- 값 3이 메모리 몇 번지에 기록되는지 알 수 없음
  - 프로그램이 실행을 시작할 때, 변수 n의 절대 메모리 주소가 정해짐
- 주소를 사용하는 것보다 이름 n을 사용하는 것이 용이함



# 포인터 변수 선언

## □ 포인터 변수

### ▣ 포인터 즉 주소를 저장하는 변수

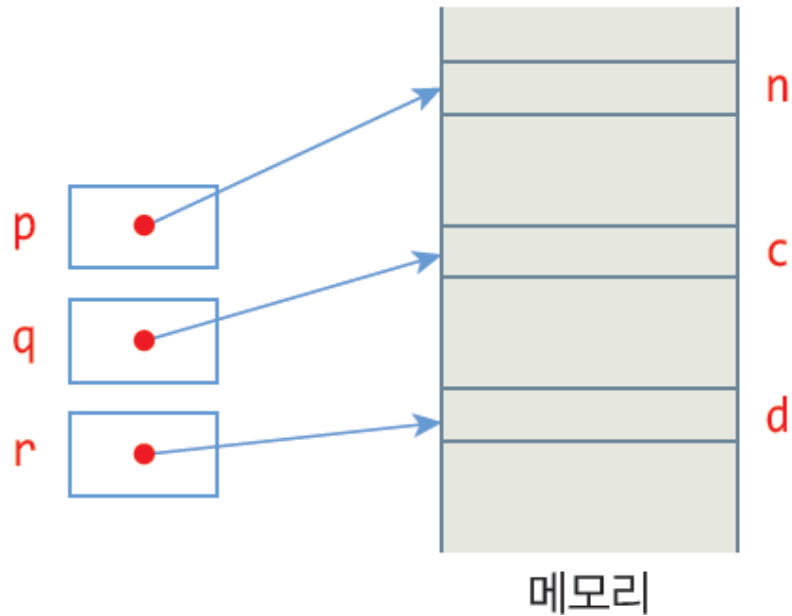
```
int *p; // 정수를 저장하는 메모리에 대한 포인터 변수 p 선언  
p = &n; // p에 n의 주소를 저장
```

```
int n;  
char c;  
double d;
```

```
int *p = &n;
```

```
char *q = &c;
```

```
double *r = &d;
```



# 예제 포인터 선언 및 활용

포인터를 이용하여 변수에 들어 있는 값을 출력하는 코드를 보인다.

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

```
int main() {
    int n=10, m;
    char c='A';
    double d;
```

```
    int *p= &n; // p는 n의 주소값을 가짐
    char *q = &c; // q는 c의 주소값을 가짐
    double *r = &d; // r은 d의 주소값을 가짐
```

```
    *p = 25; // n에 25가 저장됨
    *q = 'A'; // c에 문자 'A'가 저장됨
    *r = 3.14; // d에 3.14가 저장됨
    m = *p + 10; // p가 가리키는 값(n 변수값)+10을 m에 저장
```

```
    cout << n << ' ' << *p << "\n"; // 둘 다 25가 출력됨
    cout << c << ' ' << *q << "\n"; // 둘 다 'A'가 출력됨
    cout << d << ' ' << *r << "\n"; // 둘 다 3.14가 출력됨
    cout << m << "\n"; // m 값 35 출력
```

```
}
```

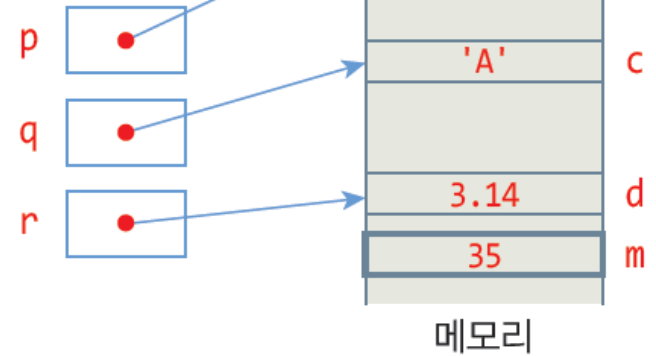
```
int m;
```

```
*p = 25;
```

```
*q = 'A';
```

```
*r = 3.14;
```

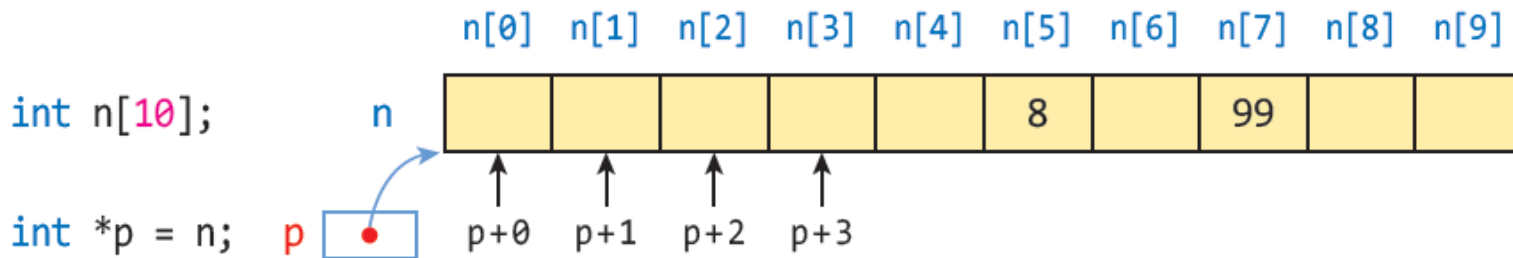
```
m = *p + 10;
```



```
25 25
A A
3.14 3.14
35
```

# 배열과 포인터

- 배열 이름은 배열 메모리의 시작 주소로 다름



<code>n[5] = 8</code>	→ 배열 <code>n</code> 의 시작 위치에서 5만큼 떨어진 주소에 8 기록
<code>n + 5</code>	→ <code>n[5]</code> 의 주소
<code>*(n + 5) = 8;</code>	→ <code>n[5]</code> 에 8기록
<code>p = p + 7;</code>	→ <code>p</code> 는 <code>n[7]</code> 의 주소
<code>*p = 99;</code>	→ <code>n[7]</code> 에 99 기록

# 예제 포인터로 배열 접근

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

int main() {
    int n[10];
    int i;
    int *p;

    for(i=0; i<10; i++)
        *(n+i) = i*3; // 배열의 이름 n을 주소처럼 사용 가능. 배열 n을 3의 배수로 채움

    p = n; // 포인터 p에 배열 n의 시작 주소를 설정한다.
    for(i=0; i<10; i++) {
        cout << *(p+i) << ' '; // 포인터 p를 이용하여 배열 n의 원소 접근
    }
    cout << "\n";

    for(i=0; i<10; i++) {
        *p = *p + 2; // 포인터 p를 이용하여 배열의 원소 값을 2 증가
        p++; // p는 다음 원소의 주소로 증가
    }

    for(i=0; i<10; i++)
        cout << n[i] << ' ';
    cout << "\n";
}
```

배열 n을  
3의 배수로 초기화

포인터 p를 이용하여  
배열 n 출력

포인터 p를 이용하여  
배열 n의 원소  
값 2 증가

배열 n 출력

```
0 3 6 9 12 15 18 21 24 27
2 5 8 11 14 17 20 23 26 29
```

# 예제 포인터를 매개 변수로 전달받는 함수

포인터로 정수 2개를 전달받아 비교하는 함수 `equal()`을 작성하라.

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

bool equal(int* p, int* q); // 함수의 원형 선언

int main() {
    int a=5, b=6;
    if(equal(&a, &b)) cout << "equal" << "\n";
    else cout << "not equal" << "\n";
}

bool equal(int* p, int* q) { // 포인터 매개 변수
    if(*p == *q) return true;
    else return false;
}
```

not equal

# C++: C 언어에 추가한 기능

- 함수 중복(function overloading)
  - 매개 변수의 개수나 타입이 다른 동일한 이름의 함수들 선언
- 디폴트 매개 변수(default parameter)
  - 매개 변수에 디폴트 값이 전달되도록 함수 선언
- 참조와 참조 변수(reference)
  - 하나의 변수에 별명을 사용하는 참조 변수 도입
- 참조에 의한 호출(call-by-reference)
  - 함수 호출 시 참조 전달
- new/delete 연산자
  - 동적 메모리 할당/해제를 위해 new와 delete 연산자 도입
- 연산자 재정의
  - 기존 C++ 연산자에 새로운 연산 정의
- 제네릭 함수와 클래스
  - 데이터 타입에 의존하지 않고 일반화시킨 함수나 클래스 작성 가능

# C++ 객체 지향 특성 - 캡슐화

- 캡슐화(Encapsulation)
  - ▣ 데이터를 캡슐로 싸서 외부의 접근으로부터 보호
  - ▣ C++에서 클래스(class 키워드)로 캡슐 표현
- 클래스와 객체
  - ▣ 클래스 – 객체를 만드는 틀
  - ▣ 객체 – 클래스라는 틀에서 생겨난 실체
  - ▣ 객체(object), 실체(instance)는 같은 뜻

멤버들

```
class Circle {  
    private:  
        int radius; // 반지름 값  
    public:  
        Circle(int r) { radius = r; }  
        double getArea() { return 3.14*radius*radius; }  
};
```

원을 추상화한 Circle 클래스

# 표준 입출력

## □ cout과 << 연산자 이용

```
std::cout << "Hello\n"; // 화면에 Hello를 출력하고 다음 줄로 넘어감  
std::cout << "첫 번째 맛보기입니다.";
```

## □ cout 객체

- 스크린 출력 장치에 연결된 **표준** C++ 출력 스트림 객체
- <iostream> 헤더 파일에 선언
- std 이름 공간에 선언: **std::cout**으로 사용

## □ << 연산자

- 스트림 삽입 연산자(stream insertion operator)
  - C++ 기본 산술 시프트 연산자(<<)&gt;가 스트림 삽입 연산자로 재정의됨
  - ostream 클래스에 구현됨
  - 오른쪽 피연산자를 왼쪽 스트림 객체에 삽입
  - cout 객체에 연결된 화면에 출력
- 여러 개의 << 연산자로 여러 값 출력

```
std::cout << "Hello\n" << "첫 번째 맛보기입니다.";
```



# 예제 cout과 <<를 이용한 화면 출력

```
#include <iostream>
```

```
double area(int r); // 함수의 원형 선언
```

```
double area(int r) { // 함수 구현  
    return 3.14*r*r; // 반지름 r의 원면적 리턴  
}
```

```
int main() {  
    int n=3;  
    char c='#';  
    std::cout << c << 5.5 << '-' << n << "hello" << true << std::endl;  
    std::cout << "n + 5 = " << n + 5 << '\n';  
    std::cout << "면적은 " << area(n); // 함수 area()의 리턴 값 출력  
}
```

true는  
1로  
출력됨

```
#5.5-3hello1  
n + 5 = 8  
면적은 28.26
```

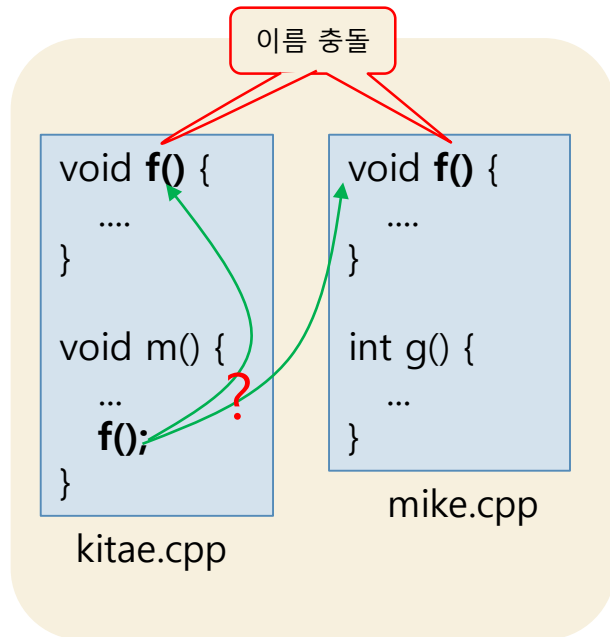
# namespace 개념

- 이름(identifier) 충돌이 발생하는 경우
  - 여러 명이 서로 나누어 프로젝트를 개발하는 경우
  - 오픈 소스 혹은 다른 사람이 작성한 소스나 목적 파일을 가져와서 컴파일 하거나 링크하는 경우
  - 해결하는데 많은 시간과 노력이 필요
- namespace 키워드
  - 이름 충돌 해결
    - 2003년 새로운 C++ 표준에서 도입
  - 개발자가 자신만의 이름 공간을 생성할 수 있도록 함
    - 이름 공간 안에 선언된 이름은 다른 이름공간과 별도 구분
- 이름 공간 생성 및 사용

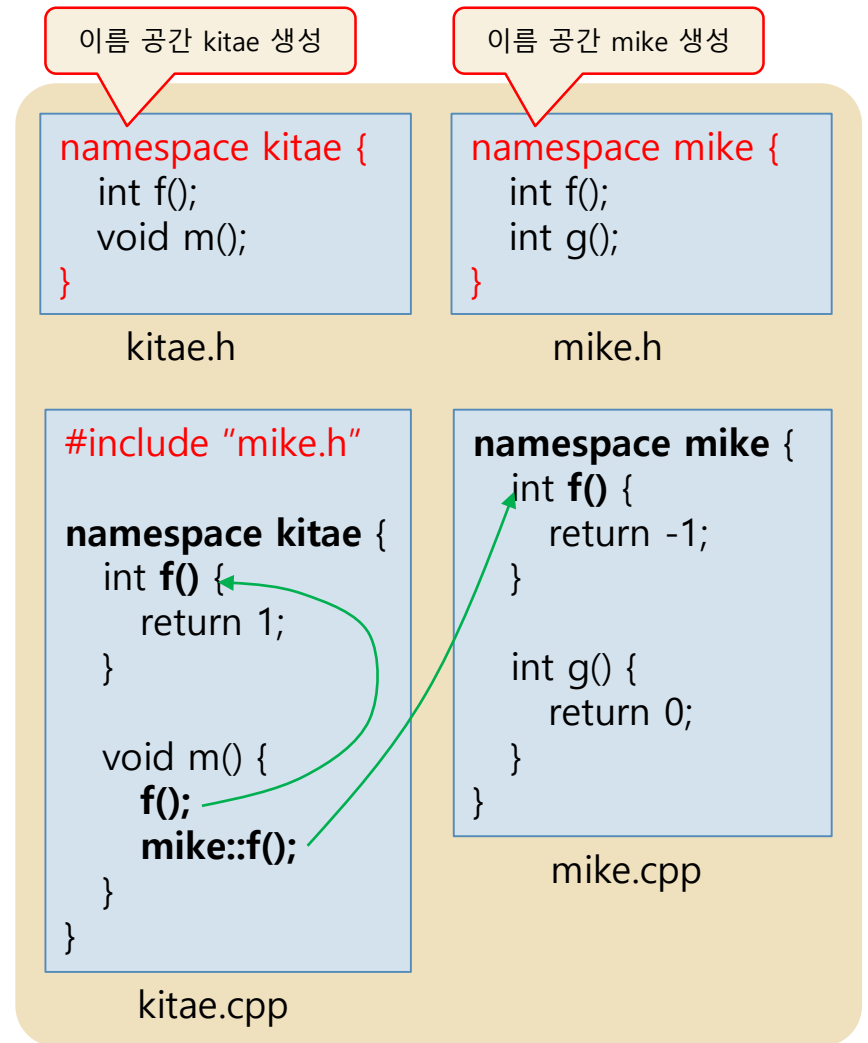
```
namespace kitae { // kitae 라는 이름 공간 생성
    ..... // 이 곳에 선언된 모든 이름은 kitae 이름 공간에 생성된 이름
}
```

- 이름 공간 사용
  - 이름 공간 :: 이름

# namespace 개념



(a) kitae와 mike에 의해 작성된 소스를 합치면  
f() 함수의 이름 충돌. 컴파일 오류 발생



(b) 이름 공간을 사용하여 f() 함수 이름의 충돌 문제 해결

# std:: 란?

## □ std

- C++ 표준에서 정의한 **이름 공간(namespace)** 중 하나
  - <iostream> 헤더 파일에 선언된 모든 이름: std 이름 공간 안에 있음
  - cout, cin, endl 등
- std 이름 공간에 선언된 이름을 접근하기 위해 std:: 접두어 사용
  - std::cout, std::cin, std::endl

## □ std:: 생략

- using 지시어 사용

```
using std::cout; // cout에 대해서만 std:: 생략
```

std:: 생략

```
.....  
cout << "Hello" << std::endl; // std::cout에서 std:: 생략
```

```
using namespace std; // std 이름 공간에 선언된 모든 이름에 std:: 생략
```

```
.....  
cout << "Hello" << endl; // std:: 생략
```

std:: 생략

std:: 생략

# 예제 C++ 프로그램에서 키 입력 받기

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

int main() {
    cout << "너비를 입력하세요>>";

    int width;
    cin >> width; // 키보드로부터 너비를 읽어 width 변수에 저장

    cout << "높이를 입력하세요>>";

    int height;
    cin >> height; // 키보드로부터 높이를 읽어 height 변수에 저장

    int area = width*height; // 사각형의 면적 계산
    cout << "면적은 " << area << "\n"; // 면적을 출력하고 다음 줄로 넘어감
}
```

```
너비를 입력하세요>>3
높이를 입력하세요>>5
면적은 15
```

# 클래스와 객체



# C++클래스와 C++객체

---

## □ 클래스

- ▣ 객체를 만들어내기 위해 정의된 설계도, 틀
- ▣ 클래스는 객체가 아님. 실체도 아님
- ▣ 멤버 변수와 멤버 함수 선언

## □ 객체

- ▣ 객체는 생성될 때 클래스의 모양을 그대로 가지고 탄생
- ▣ 멤버 변수와 멤버 함수로 구성
- ▣ 메모리에 생성, 실체(instance)라고도 부름
- ▣ 하나의 클래스 틀에서 찍어낸 여러 개의 객체 생성 가능
- ▣ 객체들은 상호 별도의 공간에 생성

# C++ 클래스

---

- 클래스 작성
  - ▣ 멤버 변수와 멤버 함수로 구성
  - ▣ 클래스 선언부와 클래스 구현부로 구성
- 클래스 선언부(class declaration)
  - ▣ class 키워드를 이용하여 클래스 선언
  - ▣ 멤버 변수와 멤버 함수 선언
    - 멤버 변수는 클래스 선언 내에서 초기화할 수 없음
    - 멤버 함수는 원형(prototype) 형태로 선언
  - ▣ 멤버에 대한 접근 권한 지정
    - private, public, protected 중의 하나
    - 디폴트는 private
    - public : 다른 모든 클래스나 객체에서 멤버의 접근이 가능함을 표시
- 클래스 구현부(class implementation)
  - ▣ 클래스에 정의된 모든 멤버 함수 구현



# C++ 클래스

클래스의 선언은  
class 키워드 이용

클래스  
이름

멤버에 대한 접근 지정자

세미콜론으로 끝남

```
class Circle {  
public:  
    int radius; // 멤버 변수  
    double getArea(); // 멤버 함수  
};
```

클래스  
선언부

함수의  
리턴 타입

클래스  
이름

범위지정  
연산자

멤버 함수명과  
매개변수

```
double Circle :: getArea() {  
    return 3.14*radius*radius;  
}
```

클래스  
구현부

클래스 선언과 클래스  
구현으로 분리하는 이유는  
클래스를 다른 파일에서  
활용하기 위함

# 예제 Circle 클래스의 객체 생성 및 활용

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

class Circle {
public:
    int radius;
    double getArea();
};

double Circle::getArea() {
    return 3.14*radius*radius;
}

int main() {
    Circle donut;
    donut.radius = 1; // donut 객체의 반지름을 1로 설정
    double area = donut.getArea(); // donut 객체의 면적 알아내기
    cout << "donut 면적은 " << area << endl;

    Circle pizza;
    pizza.radius = 30; // pizza 객체의 반지름을 30으로 설정
    area = pizza.getArea(); // pizza 객체의 면적 알아내기
    cout << "pizza 면적은 " << area << endl;
}
```

Circle 선언부

Circle 구현부

객체 donut 생성

donut의 멤버  
변수 접근

donut의 멤버  
함수 호출

donut 면적은 3.14  
pizza 면적은 2826

# 실습 – Rectangle 클래스 만들기

다음 main() 함수가 잘 작동하도록 너비(width)와 높이(height)를 가지고 면적 계산 기능을 가진 Rectangle 클래스를 작성하고 전체 프로그램을 완성하라.

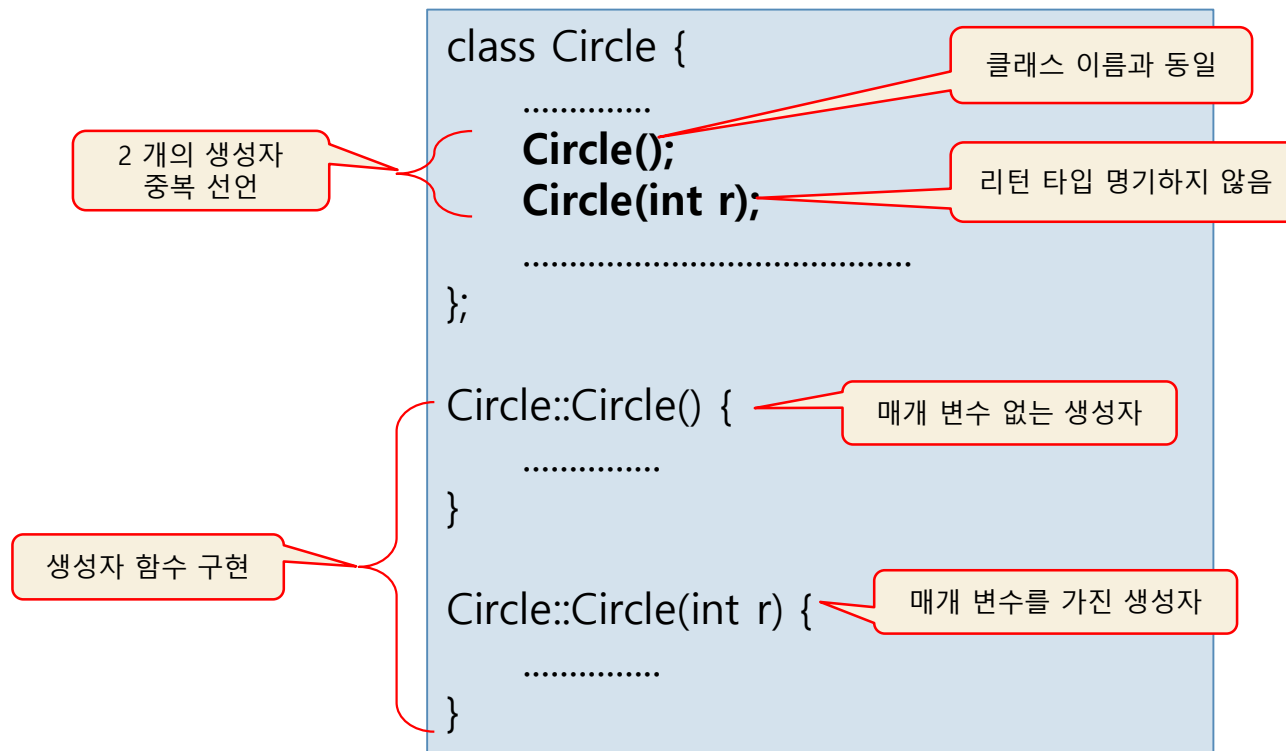
```
int main() {  
    Rectangle rect;  
    rect.width = 3;  
    rect.height = 5;  
    cout << "사각형의 면적은 " << rect.getArea() << endl;  
}
```

사각형의 면적은 15

# 생성자

## □ 생성자(constructor)

- ▣ 객체가 **생성**되는 시점에서 **자동**으로 호출되는 **멤버 함수**
- ▣ 클래스 이름과 동일한 멤버 함수



## 예제 2 개의 생성자를 가진 Circle 클래스

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

```
class Circle {
public:
    int radius;
    Circle(); // 매개 변수 없는 생성자
    Circle(int r); // 매개 변수 있는 생성자
    double getArea();
};
```

```
Circle::Circle() {
    radius = 1;
    cout << "반지름 " << radius << " 원 생성" << endl;
}
```

```
Circle::Circle(int r) {
    radius = r;
    cout << "반지름 " << radius << " 원 생성" << endl;
}
```

```
double Circle::getArea() {
    return 3.14*radius*radius;
}
```

Circle(); 자동 호출

Circle(30); 자동 호출

```
int main() {
```

```
    Circle donut; // 매개 변수 없는 생성자 호출
    double area = donut.getArea();
    cout << "donut 면적은 " << area << endl;
```

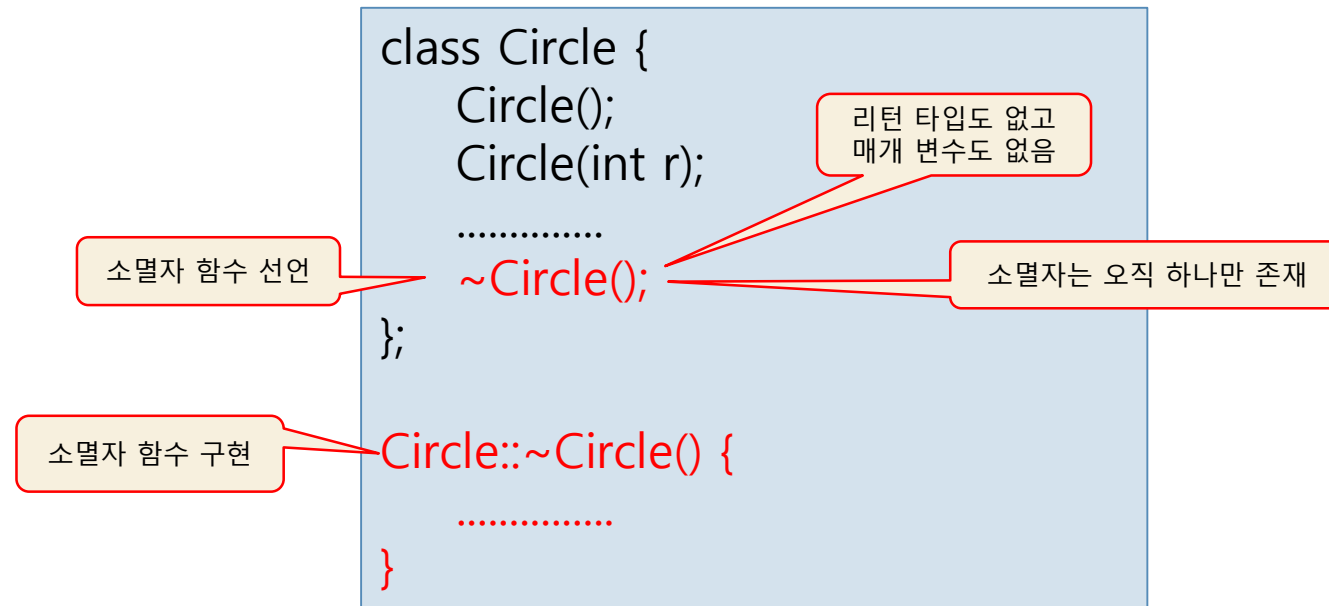
```
    Circle pizza(30); // 매개 변수 있는 생성자 호출
    area = pizza.getArea();
    cout << "pizza 면적은 " << area << endl;
}
```

반지름 1 원 생성  
donut 면적은 3.14  
반지름 30 원 생성  
pizza 면적은 2826

# 소멸자

## □ 소멸자(destructor)

- ▣ 객체가 **소멸**되는 시점에서 **자동**으로 호출되는 **함수**
  - 오직 한번만 자동 호출, 임의로 호출할 수 없음
  - 객체 메모리 소멸 직전 호출됨



# 예제 Circle 클래스에 소멸자 작성 및 실행

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

class Circle {
public:
    int radius;

    Circle();
    Circle(int r);
    ~Circle(); // 소멸자
    double getArea();
};

Circle::Circle() {
    radius = 1;
    cout << "반지름 " << radius << " 원 생성" << endl;
}

Circle::Circle(int r) {
    radius = r;
    cout << "반지름 " << radius << " 원 생성" << endl;
}

Circle::~Circle() {
    cout << "반지름 " << radius << " 원 소멸" << endl;
}
```

```
double Circle::getArea() {
    return
    3.14*radius*radius;
}

int main() {
    Circle donut;
    Circle pizza(30);

    return 0;
}
```

main() 함수가 종료하면 main() 함수의 스택에 생성된 pizza, donut 객체가 소멸된다.

반지름 1 원 생성  
반지름 30 원 생성  
반지름 30 원 소멸  
반지름 1 원 소멸

객체는 생성의  
반대순으로  
소멸된다.

# 접근 지정자

- 캡슐화의 목적
  - ▣ 객체 보호, 보안
  - ▣ C++에서 객체의 캡슐화 전략
    - 객체의 상태를 나타내는 데이터 멤버(멤버 변수)에 대한 보호
    - 중요한 멤버는 다른 클래스나 객체에서 접근할 수 없도록 보호
    - 외부와의 인터페이스를 위해서 일부 멤버는 외부에 접근 허용
- 멤버에 대한 3 가지 접근 지정자
  - ▣ private
    - 동일한 클래스의 멤버 함수에만 제한함
  - ▣ public
    - 모든 다른 클래스에 허용
  - ▣ protected
    - 클래스 자신과 상속받은 자식 클래스에만 허용

```
class Sample {  
    private:  
        // private 멤버 선언  
    public:  
        // public 멤버 선언  
    protected:  
        // protected 멤버 선언  
};
```



# 객체 포인터

- 객체에 대한 포인터
  - ▣ C 언어의 포인터와 동일
  - ▣ 객체의 주소 값을 가지는 변수
- 포인터로 멤버를 접근할 때
  - ▣ 객체포인터->멤버

```
Circle donut;  
double d = donut.getArea();
```

객체에 대한 포인터 선언

```
Circle *p; // (1)  
p = &donut; // (2)  
d = p->getArea(); // (3)
```

포인터에 객체 주소 저장

멤버 함수 호출

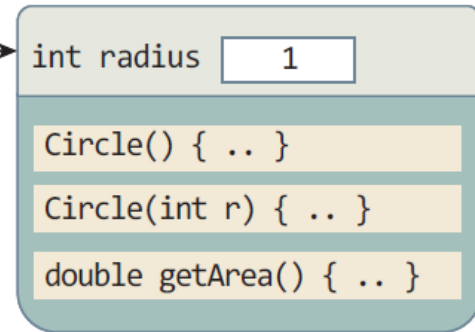
(1) Circle \*p;



(2) p=&donut;



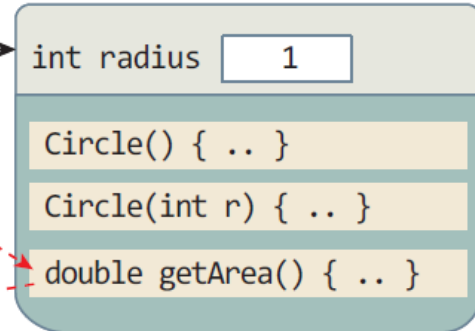
donut 객체



(3) d=p->getArea();



donut 객체



호출



# 예제 객체 포인터 선언 및 활용

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

class Circle {
    int radius;
public:
    Circle() { radius = 1; }
    Circle(int r) { radius = r; }
    double getArea();
};

double Circle::getArea() {
    return 3.14*radius*radius;
}
```

```
int main() {
    Circle donut;
    Circle pizza(30);

    // 객체 이름으로 멤버 접근
    cout << donut.getArea() << endl;

    // 객체 포인터로 멤버 접근
    Circle *p;
    p = &donut;
    cout << p->getArea() << endl; // donut의 getArea() 호출
    cout << (*p).getArea() << endl; // donut의 getArea() 호출

    p = &pizza;
    cout << p->getArea() << endl; // pizza의 getArea() 호출
    cout << (*p).getArea() << endl; // pizza의 getArea() 호출
}
```

```
3.14
3.14
3.14
2826
2826
```

# 동적 메모리 할당 및 반환

## □ C++의 동적 메모리 할당/반환

### ▣ new 연산자

- 기본 타입 메모리 할당, 배열 할당, 객체 할당, 객체 배열 할당
- 객체의 동적 생성 - 힙 메모리로부터 객체를 위한 메모리 할당 요청
- 객체 할당 시 생성자 호출

### ▣ delete 연산자

- new로 할당 받은 메모리 반환
- 객체의 동적 소멸 - 소멸자 호출 뒤 객체를 힙에 반환

```
int *pInt = new int; // int 타입의 메모리 동적 할당  
char *pChar = new char; // char 타입의 메모리 동적 할당  
Circle *pCircle = new Circle(); // Circle 클래스 타입의 메모리 동적 할당
```

```
delete pInt; // 할당 받은 정수 공간 반환  
delete pChar; // 할당 받은 문자 공간 반환  
delete pCircle; // 할당 받은 객체 공간 반환
```

# this 포인터

## □ this

- 포인터, 객체 자신 포인터
- 클래스의 멤버 함수 내에서만 사용
- 개발자가 선언하는 변수가 아니고, 컴파일러가 선언한 변수
  - 멤버 함수에 컴파일러에 의해 묵시적으로 삽입 선언되는 매개 변수

```
class Circle {  
    int radius;  
public:  
    Circle() { this->radius=1; }  
    Circle(int radius) { this->radius = radius; }  
    void setRadius(int radius) { this->radius = radius; }  
    ....  
};
```

# 함수와 참조



# 참조 변수

## □ 참조 변수 선언

- ▣ 참조자 &의 도입
- ▣ 이미 존재하는 변수에 대한 다른 이름(별명)을 선언
  - 참조 변수는 이름만 생기며
  - 참조 변수에 새로운 공간을 할당하지 않는다.
  - 초기화로 지정된 기존 변수를 공유한다.

```
int n=2;  
int &refn = n; // 참조 변수 refn 선언. refn은 n에 대한 별명
```

```
Circle circle;  
Circle &refc = circle; // 참조 변수 refc 선언. refc는 circle에 대한 별명
```

# 예제 기본 타입 변수에 대한 참조

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

int main() {
    cout << "i" << 'Wt' << "n" << 'Wt' << "refn" << endl;
    int i = 1;
    int n = 2;
    int &refn = n; // 참조 변수 refn 선언. refn은 n에 대한 별명
    n = 4;
    refn++; // refn=5, n=5
    cout << i << 'Wt' << n << 'Wt' << refn << endl;

    refn = i; // refn=1, n=1
    refn++; // refn=2, n=2
    cout << i << 'Wt' << n << 'Wt' << refn << endl;

    int *p = &refn; // p는 n의 주소를 가짐
    *p = 20; // refn=20, n=20
    cout << i << 'Wt' << n << 'Wt' << refn << endl;
}
```

참조 변수 refn 선언

참조에 대한  
포인터 변수 선언

i	n	refn
1	5	5
1	2	2
1	20	20

# 참조에 의한 호출

---

- 참조를 가장 많이 활용하는 사례
- call by reference라고 부름
- 함수 형식
  - ▣ 함수의 매개 변수를 참조 타입으로 선언
    - 참조 매개 변수(reference parameter)라고 부름
      - 참조 매개 변수는 실인자 변수를 참조함
    - 참조매개 변수의 이름만 생기고 공간이 생기지 않음
    - 참조 매개 변수는 실인자 변수 공간 공유
    - 참조 매개 변수에 대한 조작은 실인자 변수 조작 효과



# 예제 참조 매개 변수로 평균 리턴하기

참조 매개 변수를 통해 평균을 리턴하고 리턴문을 통해서 함수의 성공 여부를 리턴하도록 `average()` 함수를 작성하라

avg에 평균이 넘어오고,  
`average()`는 true 리턴

avg의 값은 의미없고,  
`average()`는 false 리턴

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

bool average(int a[], int size, int& avg) {
    if(size <= 0)
        return false;
    int sum = 0;
    for(int i=0; i<size; i++)
        sum += a[i];
    avg = sum/size;
    return true;
}

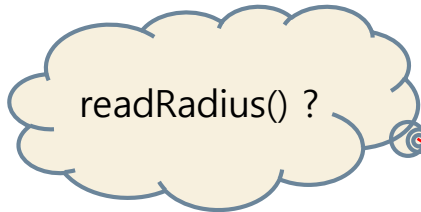
int main() {
    int x[] = {0,1,2,3,4,5};
    int avg;
    if(average(x, 6, avg)) cout << "평균은 " << avg << endl;
    else cout << "매개 변수 오류" << endl;

    if(average(x, -2, avg)) cout << "평균은 " << avg << endl;
    else cout << "매개 변수 오류 " << endl;
}
```

평균은 2  
매개 변수 오류

# 실습 참조 매개 변수를 가진 함수 만들기

키보드로부터 반지름 값을 읽어  
Circle 객체에 반지름을 설정하는  
readRadius() 함수를 작성하라.



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

class Circle {
    int radius;
public:
    Circle() { radius = 1; }
    Circle(int radius) { this->radius = radius; }
    void setRadius(int radius) { this->radius = radius; }
    double getArea() { return 3.14*radius*radius; }
};

int main() {
    Circle donut;
    readRadius(donut);
    cout << "donut의 면적 = " << donut.getArea() << endl;
}
```

정수 값으로 반지름을 입력하세요>>3  
donut의 면적 = 28.26

# 함수 중복과 디폴트 매개변수



# 함수 중복

```
int sum(int a, int b, int c) {  
    return a + b + c;  
}
```

```
double sum(double a, double b) {  
    return a + b;  
}
```

```
int sum(int a, int b) {  
    return a + b;  
}
```

성공적으로 중복된 sum() 함수들

```
int main(){  
    cout << sum(2, 5, 33);  
  
    cout << sum(12.5, 33.6);  
  
    cout << sum(2, 6);  
}
```

중복된 sum() 함수 호출.  
컴파일러가 구분

# 디폴트 매개 변수

- 디폴트 매개 변수(default parameter)
  - ▣ 매개 변수에 값이 넘어오지 않는 경우, 디폴트 값을 받도록 선언된 매개 변수
    - ‘매개 변수 = 디폴트값’ 형태로 선언

- 디폴트 매개 변수 선언 사례

```
void star(int a=5); // a의 디폴트 값은 5
```

- 디폴트 매개 변수를 가진 함수 호출

```
star(); // 매개 변수 a에 디폴트 값 5가 전달됨. star(5);와 동일  
star(10); // 매개 변수 a에 10을 넘겨줌
```

# 예제 디폴트 매개 변수를 가진 함수 선언 및 호출

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

```
// 원형 선언
```

```
void star(int a=5);
void msg(int id, string text="");
```

디폴트  
매개 변수 선언

```
// 함수 구현
```

```
void star(int a) {
    for(int i=0; i<a; i++)
        cout << '*';
    cout << endl;
}
```

```
void msg(int id, string text) {
    cout << id << ' ' << text << endl;
}
```

동일한  
코드

```
void star(int a=5) {
    for(int i=0; i<a; i++)
        cout << '*';
    cout << endl;
}
```

```
void msg(int id, string text="") {
    cout << id << ' ' << text << endl;
}
```

```
int main() {
    // star() 호출
    star();
    star(10);
```

star(5);

```
// msg() 호출
```

```
msg(10);
msg(10, "Hello");
}
```

msg(10, "");

\*\*\*\*\*

\*\*\*\*\*

10

10 Hello

## □ 프렌드 함수

- 클래스의 멤버 함수가 아닌 외부 함수
  - 전역 함수
  - 다른 클래스의 멤버 함수
- friend 키워드로 클래스 내에 선언된 함수
  - 클래스의 모든 멤버를 접근할 수 있는 권한 부여
  - 프렌드 함수라고 부름
- 프렌드 선언의 필요성
  - 클래스의 멤버로 선언하기에는 무리가 있고, 클래스의 모든 멤버를 자유롭게 접근할 수 있는 일부 외부 함수 작성 시
- 프렌드 함수가 되는 3 가지
  - 전역 함수 : 클래스 외부에 선언된 전역 함수
  - 다른 클래스의 멤버 함수 : 다른 클래스의 특정 멤버 함수
  - 다른 클래스 전체 : 다른 클래스의 모든 멤버 함수

# C++ 프렌드

---

## 1. 외부 함수 equals()를 Rect 클래스에 프렌드로 선언

```
class Rect { // Rect 클래스 선언
    ...
    friend bool equals(Rect r, Rect s);
};
```

## 2. RectManager 클래스의 equals() 멤버 함수를 Rect 클래스에 프렌드로 선언

```
class Rect {
    .....
    friend bool RectManager::equals(Rect r, Rect s);
};
```

## 3. RectManager 클래스의 모든 멤버 함수를 Rect 클래스에 프렌드로 선언

```
class Rect {
    .....
    friend RectManager;
};
```



# 예제 프랜드 함수 만들기

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

```
class Rect;
bool equals(Rect r, Rect s); // equals() 함수 선언
```

Rect 클래스가 선언되기 전에 먼저 참조되는  
컴파일 오류(forward reference)를 막기 위한  
선언문(forward declaration)

```
class Rect { // Rect 클래스 선언
    int width, height;
public:
    Rect(int width, int height) { this->width = width; this->height = height; }
    friend bool equals(Rect r, Rect s);
};
```

equals() 함수를  
프렌드로 선언

```
bool equals(Rect r, Rect s) { // 외부 함수
    if(r.width == s.width && r.height == s.height) return true;
    else return false;
}
```

equals() 함수는 private 속성을 가진  
width, height에 접근할 수 있다.

```
int main() {
    Rect a(3,4), b(4,5);
    if(equals(a, b)) cout << "equal" << endl;
    else cout << "not equal" << endl;
}
```

객체 a와 b는 동일한 크기의  
사각형이므로 "not equal" 출력

not equal

상속



# C++에서의 상속(Inheritance)

- C++에서의 상속이란?
  - ▣ 클래스 사이에서 상속관계 정의
    - 객체 사이에는 상속 관계 없음
  - ▣ 기본 클래스의 속성과 기능을 파생 클래스에 물려주는 것
    - 기본 클래스(base class) - 상속해주는 클래스. 부모 클래스
    - 파생 클래스(derived class) - 상속받는 클래스. 자식 클래스
      - 기본 클래스의 속성과 기능을 물려받고 자신 만의 속성과 기능을 추가하여 작성
  - ▣ 기본 클래스에서 파생 클래스로 갈수록 클래스의 개념이 구체화
  - ▣ 다중 상속을 통한 클래스의 재활용성 높임

# 상속의 표현



```
class Phone {  
    void call();  
    void receive();  
};
```

Phone을 상속받는다.

```
class MobilePhone : public Phone {  
    void connectWireless();  
    void recharge();  
};
```

MobilePhone을 상속받는다.

```
class MusicPhone : public MobilePhone {  
    void downloadMusic();  
    void play();  
};
```

C++로 상속 선언



전화기



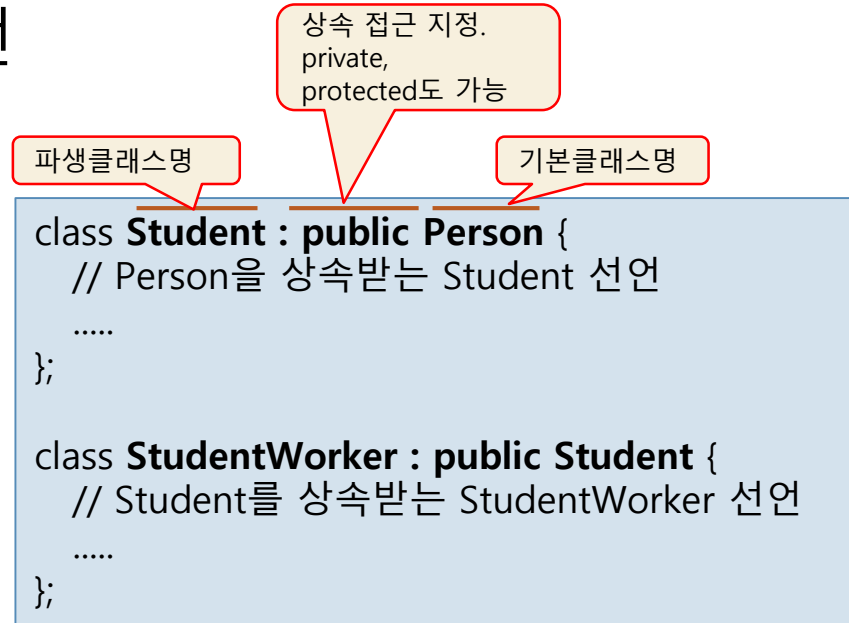
휴대 전화기



음악 기능  
전화기

# 상속 선언

## □ 상속 선언



- Student 클래스는 Person 클래스의 멤버를 물려받는다.
- StudentWorker 클래스는 Student의 멤버를 물려받는다.
  - Student가 물려받은 Person의 멤버도 함께 물려받는다.

# 예제 Point 클래스를 상속받는 ColorPoint 클래스 만들기

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

// 2차원 평면에서 한 점을 표현하는 클래스 Point 선언
class Point {
    int x, y; //한 점 (x,y) 좌표값
public:
    void set(int x, int y) { this->x = x; this->y = y; }
    void showPoint() {
        cout << "(" << x << "," << y << ")" << endl;
    }
};
```

```
class ColorPoint : public Point { // 2차원 평면에서 컬러
    점을 표현하는 클래스 ColorPoint. Point를 상속받음
    string color; // 점의 색 표현
public:
    void setColor(string color) {this->color = color; }
    void showColorPoint();
};

void ColorPoint::showColorPoint() {
    cout << color << ":";
    showPoint(); // Point의 showPoint() 호출
}

int main() {
    Point p; // 기본 클래스의 객체 생성
    ColorPoint cp; // 파생 클래스의 객체 생성
    cp.set(3,4); // 기본 클래스의 멤버 호출
    cp.setColor("Red"); // 파생 클래스의 멤버 호출
    cp.showColorPoint(); // 파생 클래스의 멤버 호출
}
```

Red:(3,4)

# protected 접근 지정

---

## □ 접근 지정자

### ▣ private 멤버

- 선언된 클래스 내에서만 접근 가능
- 파생 클래스에서도 기본 클래스의 private 멤버 직접 접근 불가

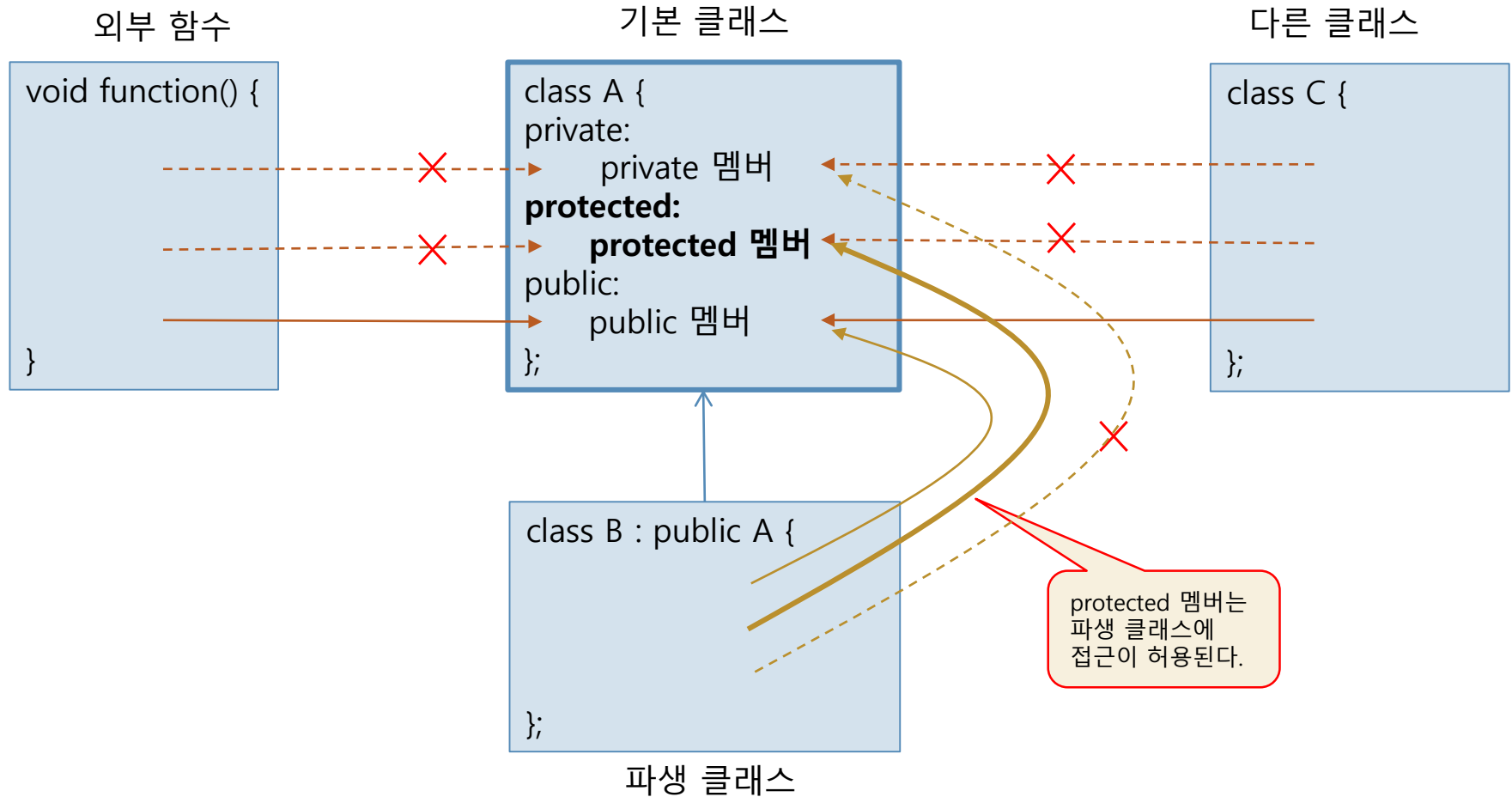
### ▣ public 멤버

- 선언된 클래스나 외부 어떤 클래스, 모든 외부 함수에 접근 허용
- 파생 클래스에서 기본 클래스의 public 멤버 접근 가능

### ▣ protected 멤버

- 선언된 클래스에서 접근 가능
- 파생 클래스에서만 접근 허용
  - 파생 클래스가 아닌 다른 클래스나 외부 함수에서는 *protected* 멤버를 접근할 수 없다.

# 멤버의 접근 지정에 따른 접근성





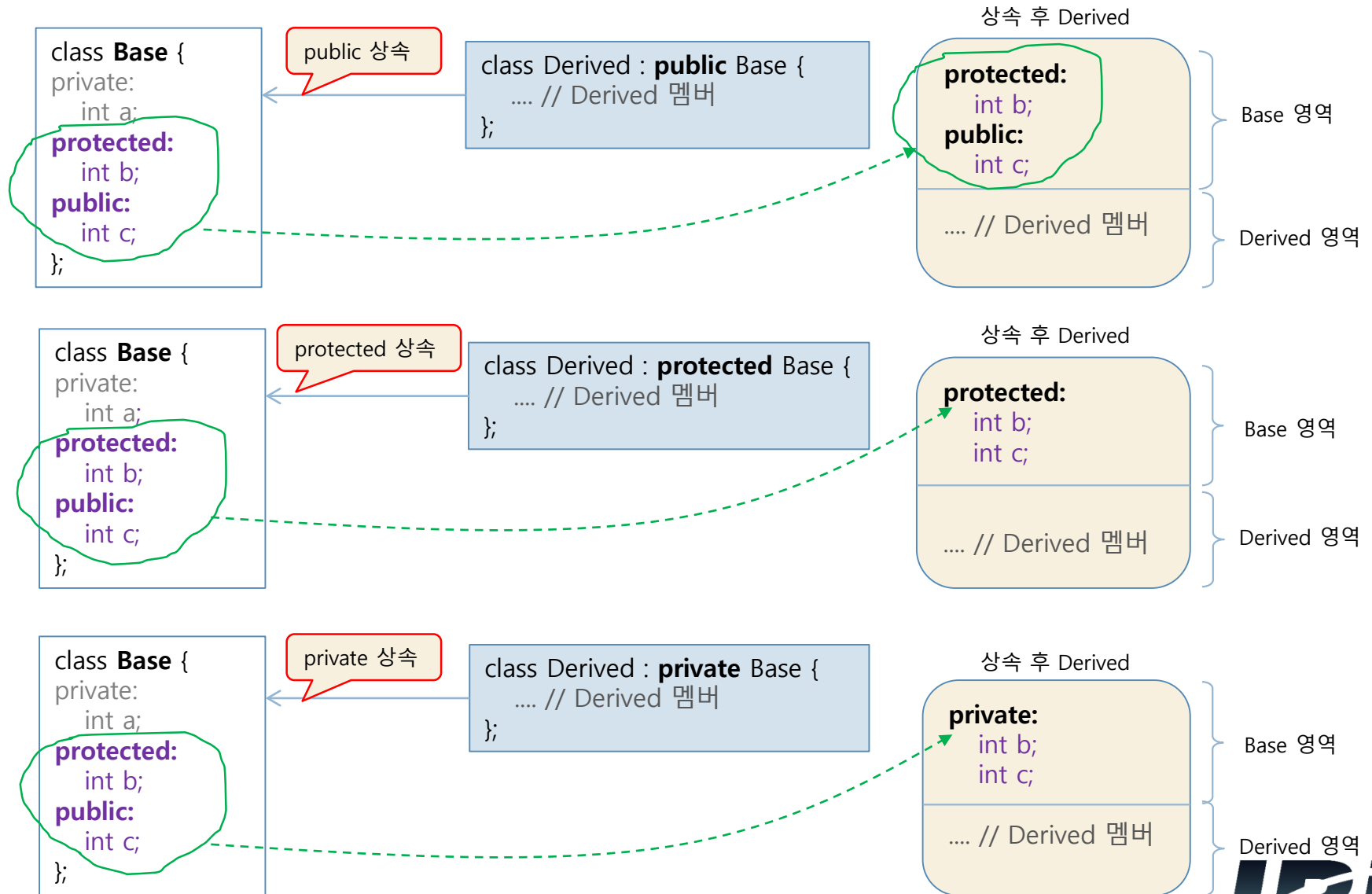
# 상속 지정

---

## ■ 상속 지정

- 상속 선언 시 `public`, `private`, `protected`의 3가지 중 하나 지정
- 기본 클래스의 멤버의 접근 속성을 어떻게 계승할지 지정
  - `public` – 기본 클래스의 `protected`, `public` 멤버 속성을 그대로 계승
  - `private` – 기본 클래스의 `protected`, `public` 멤버를 `private`으로 계승
  - `protected` – 기본 클래스의 `protected`, `public` 멤버를 `protected`로 계승

# 상속 시 접근 지정에 따른 멤버의 접근 지정 속성 변화



# 템플릿과 표준 템플릿 라이브러리(STL)



# 함수 중복의 약점 - 중복 함수의 코드 중복

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

```
void myswap(int& a, int& b) {
```

```
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```

두 함수는 매개 변수만  
다르고 나머지 코드는  
동일함

```
void myswap(double &a, double &b) {
```

```
    double tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```

동일한 코드  
중복 작성

```
int main() {
    int a=4, b=5;
    myswap(a, b); // myswap(int& a, int& b) 호출
    cout << a << '\t' << b << endl;
```

```
    double c=0.3, d=12.5;
    myswap(c, d); // myswap(double& a, double& b) 호출
    cout << c << '\t' << d << endl;
}
```

5	4
12.5	0.3

# 일반화와 템플릿

- 제네릭(generic) 또는 일반화
  - ▣ 함수나 클래스를 일반화시키고, 매개 변수 타입을 지정하여 틀에서 찍어 내듯이 함수나 클래스 코드를 생산하는 기법
- 템플릿
  - ▣ 함수나 클래스를 일반화하는 C++ 도구
  - ▣ template 키워드로 함수나 클래스 선언
    - 변수나 매개 변수의 타입만 다르고, 코드 부분이 동일한 함수를 일반화시킴
  - ▣ 제네릭 타입 - 일반화를 위한 데이터 타입

- 템플릿 선언

```
template <class T> 또는  
template <typename T>
```

```
3 개의 제네릭 타입을 가진 템플릿 선언  
template <class T1, class T2, class T3>
```

템플릿을  
선언하는 키워드

제네릭 타입을  
선언하는 키워드

제네릭 타입 T 선언

```
template <class T>  
void myswap (T & a, T & b) {  
    T tmp;  
    tmp = a;  
    a = b;  
    b = tmp;  
}
```

템플릿을 이용한 제네릭 함수 myswap

# 예제 제네릭 myswap() 함수 만들기

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

class Circle {
    int radius;
public:
    Circle(int radius=1) { this->radius = radius; }
    int getRadius() { return radius; }
};

template <class T>
void myswap(T & a, T & b) {
    T tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```

```
int main() {
    int a=4, b=5;
    myswap(a, b);
    cout << "a=" << a << ", " << "b=" << b << endl;

    double c=0.3, d=12.5;
    myswap(c, d);
    cout << "c=" << c << ", " << "d=" << d << endl;

    Circle donut(5), pizza(20);
    myswap(donut, pizza);
    cout << "donut반지름=" << donut.getRadius() << ", ";
    cout << "pizza반지름=" << pizza.getRadius() << endl;
}
```

myswap(int& a, int& b)  
함수 구체화 및 호출

myswap(double& a, double& b)  
함수 구체화 및 호출

myswap(Circle& a, Circle& b)  
함수 구체화 및 호출

a=5, b=4  
c=12.5, d=0.3  
donut반지름=20, pizza반지름=5

# 제네릭 클래스 만들기

## ■ 제네릭 클래스 선언

```
template <class T>
class MyStack {
    int tos;
    T data [100]; // T 타입의 배열
public:
    MyStack();
    void push(T element);
    T pop();
};
```

## ■ 제네릭 클래스 구현

```
template <class T>
void MyStack<T>::push(T element) {
    ...
}
template <class T> T MyStack<T>::pop() {
    ...
}
```

## ■ 클래스 구체화 및 객체 활용

```
MyStack<int> iStack; // int 타입을 다루는 스택 객체 생성
MyStack<double> dStack; // double 타입을 다루는 스택 객체 생성

iStack.push(3);
int n = iStack.pop();

dStack.push(3.5);
double d = dStack.pop();
```

# 예제 제네릭 스택 클래스 만들기

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

template <class T>
class MyStack {
    int tos; // top of stack
    T data [100]; // T 타입의 배열. 스택의 크기는 100
public:
    MyStack();
    void push(T element); // element를 data [] 배열에 삽입
    T pop(); // 스택의 탑에 있는 데이터를 data[] 배열에서 리턴
};

template <class T>
MyStack<T>::MyStack() { // 생성자
    tos = -1; // 스택은 비어 있음
}

template <class T>
void MyStack<T>::push(T element) {
    if(tos == 99) {
        cout << "stack full";
        return;
    }
    tos++;
    data[tos] = element;
}

template <class T>
T MyStack<T>::pop() {
    T retData;
    if(tos == -1) {
        cout << "stack empty";
        return 0; // 오류 표시
    }
    retData = data[tos--];
    return retData;
}
```

```
int main() {
    MyStack<int> iStack; // int 만 저장하는 스택
    iStack.push(3);
    cout << iStack.pop() << endl;

    MyStack<double> dStack; // double 만 저장하는 스택
    dStack.push(3.5);
    cout << dStack.pop() << endl;

    MyStack<char> *p = new MyStack<char>(); // char만 저장하는 스택
    p->push('a');
    cout << p->pop() << endl;
    delete p;
}
```

3  
3.5  
a



# 예제 두 개의 제네릭 타입을 가진 클래스 만들기

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

template <class T1, class T2> // 두 개의 제네릭 타입 선언
class GClass {
    T1 data1;
    T2 data2;
public:
    GClass();
    void set(T1 a, T2 b);
    void get(T1 &a, T2 &b);
};

template <class T1, class T2>
GClass<T1, T2>::GClass() {
    data1 = 0; data2 = 0;
}

template <class T1, class T2>
void GClass<T1, T2>::set(T1 a, T2 b) {
    data1 = a; data2 = b;
}

template <class T1, class T2>
void GClass<T1, T2>::get(T1 &a, T2 &b) {
    a = data1; b = data2;
}
```

```
int main() {
    int a;
    double b;
    GClass<int, double> x;
    x.set(2, 0.5);
    x.get(a, b);
    cout << "a=" << a << '\t' << "b=" << b << endl;

    char c;
    float d;
    GClass<char, float> y;
    y.set('m', 12.5);
    y.get(c, d);
    cout << "c=" << c << '\t' << "d=" << d << endl;
}
```

a=2	b=0.5
c=m	d=12.5

# C++ 표준 템플릿 라이브러리, STL

- STL(Standard Template Library)
  - ▣ 표준 템플릿 라이브러리
    - C++ 표준 라이브러리 중 하나
  - ▣ 많은 제네릭 클래스와 제네릭 함수 포함
    - 개발자는 이들을 이용하여 쉽게 응용 프로그램 작성
- STL의 구성
  - ▣ 컨테이너 – 템플릿 클래스
    - 데이터를 담아두는 자료 구조를 표현한 클래스
    - 리스트, 큐, 스택, 맵, 셋, 벡터
  - ▣ iterator – 컨테이너 원소에 대한 포인터
    - 컨테이너의 원소들을 순회하면서 접근하기 위해 만들어진 컨테이너 원소에 대한 포인터
  - ▣ 알고리즘 – 템플릿 함수
    - 컨테이너 원소에 대한 복사, 검색, 삭제, 정렬 등의 기능을 구현한 템플릿 함수
    - 컨테이너의 멤버 함수 아님

# C++ 표준 템플릿 라이브러리, STL

〈표 10-1〉 STL 컨테이너의 종류

컨테이너 클래스	설명	헤더 파일
vector	가변 크기의 배열을 일반화한 클래스	<vector>
deque	앞뒤 모두 입력 가능한 큐 클래스	<deque>
list	빠른 삽입/삭제 가능한 리스트 클래스	<list>
set	정렬된 순서로 값을 저장하는 집합 클래스. 값은 유일	<set>
map	(key, value) 쌍을 저장하는 맵 클래스	<map>
stack	스택을 일반화한 클래스	<stack>
queue	큐를 일반화한 클래스	<queue>

〈표 10-2〉 STL iterator의 종류

iterator의 종류	iterator에 ++ 연산 후 방향	read/write
iterator	다음 원소로 전진	read/write
const_iterator	다음 원소로 전진	read
reverse_iterator	지난 원소로 후진	read/write
const_reverse_iterator	지난 원소로 후진	read

〈표 10-3〉 STL 알고리즘 함수들

copy	merge	random	rotate
equal	min	remove	search
find	move	replace	sort
max	partition	reverse	swap

# STL과 관련된 헤더 파일과 이름 공간

## □ 헤더파일

### ▣ 컨테이너 클래스를 사용하기 위한 헤더 파일

- 해당 클래스가 선언된 헤더 파일 include

예) vector 클래스를 사용하려면 `#include <vector>`

list 클래스를 사용하려면 `#include <list>`

### ▣ 알고리즘 함수를 사용하기 위한 헤더 파일

- 알고리즘 함수에 상관 없이 `#include <algorithm>`

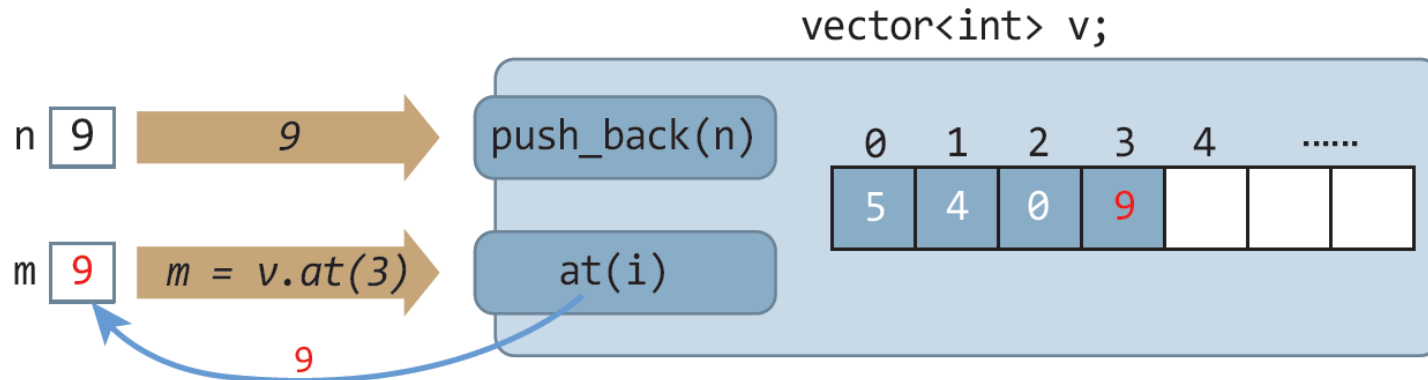
## □ 이름 공간

- ▣ STL이 선언된 이름 공간은 `std`

# vector 컨테이너

## □ 특징

- ▣ 가변 길이 배열을 구현한 제네릭 클래스
  - 개발자가 벡터의 길이에 대한 고민할 필요 없음
- ▣ 원소의 저장, 삭제, 검색 등 다양한 멤버 함수 지원
- ▣ 벡터에 저장된 원소는 인덱스로 접근 가능
  - 인덱스는 0부터 시작



# vector 클래스의 주요 멤버와 연산자

멤버와 연산자 함수	설명
<code>push_back(element)</code>	벡터의 마지막에 <code>element</code> 추가
<code>at(int index)</code>	<code>index</code> 위치의 원소에 대한 참조 리턴
<code>begin()</code>	벡터의 첫 번째 원소에 대한 참조 리턴
<code>end()</code>	벡터의 끝(마지막 원소 다음)을 가리키는 참조 리턴
<code>empty()</code>	벡터가 비어 있으면 <code>true</code> 리턴
<code>erase(iterator it)</code>	벡터에서 <code>it</code> 가 가리키는 원소 삭제. 삭제 후 자동으로 벡터 조절
<code>insert(iterator it, element)</code>	벡터 내 <code>it</code> 위치에 <code>element</code> 삽입
<code>size()</code>	벡터에 들어 있는 원소의 개수 리턴
<code>operator[]()</code>	지정된 원소에 대한 참조 리턴
<code>operator=()</code>	이 벡터를 다른 벡터에 치환(복사)

# 예제 vector 컨테이너 활용하기

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


int main() {
    vector<int> v; // 정수만 삽입 가능한 벡터 생성

    v.push_back(1); // 벡터에 정수 1 삽입
    v.push_back(2); // 벡터에 정수 2 삽입
    v.push_back(3); // 벡터에 정수 3 삽입

    for(int i=0; i<v.size(); i++) // 벡터의 모든 원소 출력
        cout << v[i] << " "; // v[i]는 벡터의 i 번째 원소
    cout << endl;

    v[0] = 10; // 벡터의 첫 번째 원소를 10으로 변경
    int n = v[2]; // n에 3이 저장
    v.at(2) = 5; // 벡터의 3 번째 원소를 5로 변경

    for(int i=0; i<v.size(); i++) // 벡터의 모든 원소 출력
        cout << v[i] << " "; // v[i]는 벡터의 i 번째 원소
    cout << endl;
}
```



1 2 3  
10 2 5

# STL 알고리즘 사용하기

## □ 알고리즘 함수

### ▣ 템플릿 함수

### ▣ 전역 함수

- STL 컨테이너 클래스의 멤버 함수가 아님

### ▣ iterator와 함께 작동

## □ sort() 함수 사례

### ▣ 두 개의 매개 변수

- 첫 번째 매개 변수 : 소팅을 시작한 원소의 주소
- 두 번째 매개 변수 : 소팅 범위의 마지막 원소 다음 주소

```
vector<int> v;
```

```
...
```

```
sort(v.begin(), v.begin()+3); // v.begin()에서 v.begin()+2까지, 처음 3개 원소 정렬
```

```
sort(v.begin()+2, v.begin()+5); // 벡터의 3번째 원소에서 v.begin()+4까지, 3개 원소 정렬
```

```
sort(v.begin(), v.end()); // 벡터 전체 정렬
```



# 예제 sort() 함수를 이용한 vector 정렬

정수 벡터에 5개의 정수를 입력 받아 저장하고, sort()를 이용하여 정렬하는 프로그램을 작성하라.

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

int main() {
    vector<int> v; // 정수 벡터 생성

    cout << "5개의 정수를 입력하세요>> ";
    for(int i=0; i<5; i++) {
        int n;
        cin >> n;
        v.push_back(n); // 키보드에서 읽은 정수를 벡터에 삽입
    }

    // v.begin()에서 v.end() 사이의 값을 오름차순으로 정렬
    // sort() 함수의 실행 결과 벡터 v의 원소 순서가 변경됨
    sort(v.begin(), v.end());

    vector<int>::iterator it; // 벡터 내의 원소를 탐색하는 iterator 변수 선언

    for(it=v.begin(); it != v.end(); it++) // 벡터 v의 모든 원소 출력
        cout << *it << ' ';
    cout << endl;
}
```

주목

5개의 정수를 입력하세요>> 30 -7 250 6 120  
-7 6 30 120 250