



# 함수 중복과 static 멤버

# 함수 중복과 static 멤버

- 이번 강의에서는
  - 함수 중복과 함수 중복 간소화
  - 디폴트 매개변수
  - static 멤버에 대하여 알아보려고 합니다.
- 함수 중복과 정적 멤버는 자바에서도 다루었던 내용이라 어렵지 않습니다
- 시작하도록 하겠습니다.

# 학습 목차

- 함수 중복
- 디폴트 매개변수
- 함수 중복 간소화
- 함수 중복 모호성
- static 멤버와 non-static 멤버

# 학습 목표

- 함수 중복을 이해하고 활용할 수 있다
- 디폴트 매개변수 개념을 이해하고 활용할 수 있다
- Static 멤버와 non – static 멤버 차이점을 이해하고 활용 할 수 있다
- 디폴트 매개변수를 사용한 함수 중복을 이해하고 활용할 수 있다

# 함수 중복

- 함수 중복
  - 동일한 이름의 함수가 공존
  - function overloading
  - 함수 중복이 가능한 범위
    - 보통 함수들 사이
    - 클래스의 멤버 함수들 사이
    - 상속 관계에 있는 기본 클래스와 파생 클래스의 멤버 함수들 사이
- 함수 중복 성공 조건
  - 중복된 함수들의 이름 동일
  - 중복된 함수들의 매개 변수 타입이 다르거나 개수가 달라야 함
  - 리턴 타입은 함수 중복과 무관

# 함수 중복 편리함

- 동일한 이름을 사용하면 함수 이름을 구분하여 기억할 필요 없고, 함수 호출을 잘못하는 실수를 줄일 수 있음

```
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;  
}
```

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

중복된 sum() 함수 호출.  
컴파일러가 구분

성공적으로 중복된 sum() 함수들

# 함수 중복 실패 사례

- 리턴 타입이 다르다고 함수 중복이 성공하지 않는다.

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

함수 중복 실패

```
int main() {  
    cout << sum(2, 5);  
}
```

컴파일러는 어떤 sum()  
함수를 호출하는지  
구분할 수 없음

# 생성자 함수 중복과 소멸자 함수

- 생성자 함수 중복 목적
  - 객체 생성시, 매개 변수를 통해 다양한 형태의 초기값 전달
- 소멸자 함수 중복 불가
  - 소멸자는 매개 변수를 가지지 않음
  - 한 클래스 내에서 소멸자는 오직 하나만 존재

```
class Circle {  
    .....  
public:  
    Circle() { radius = 1; }      //생성자 함수 중복  
    Circle(int r) { radius = r; } //생성자 함수 중복  
    ...  
};  
  
int main() {  
    Circle donut;      // Circle() 생성자 호출  
    Circle pizza(30);  // Circle(int r) 생성자 호출  
}
```



# 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 1

```
void msg(int id, string text="Hello" ); // text의 디폴트 값은 "Hello"
```

```
msg(10); // msg(10, "Hello"); 호출과 동일. id에 10, text에 "Hello" 전달
```

```
msg(20, "Good Morning"); //id에 20, text에 "Good Morning" 전달
```

```
msg(); //컴파일 오류. 첫 번째 매개 변수 id에 반드시 값을 전달하여야 함
```

```
msg("Hello"); //컴파일 오류. 첫 번째 매개 변수 id에 값이 전달되지 않았음
```

# 디폴트 매개변수

- 디폴트 매개 변수 선언 사례 2

```
void square(int width=1, int height=1);
```

**square( );**

**square(5);**

**square(3, 8);**

→ square(   ,    );

→ square( 5,    );

→ square( 3, 8 );

→ square( **1**, **1** );

→ square( 5, **1** );

→ square( 3, 8 );

컴파일러에 의해  
변환되는 과정

# 디폴트 매개 변수에 관한 제약 조건

- 디폴트 매개 변수는 보통 매개 변수 앞에 선언될 수 없음
  - 디폴트 매개 변수는 끝 쪽에 몰려 선언되어야 함

```
void calc(int a, int b=5, int c, int d=0);    //컴파일 오류  
void sum( int a=0, int b, int c);            //컴파일 오류  
void calc(int a, int b=5, int c=0, int d=0); // 컴파일 성공
```

# 디폴트 매개 변수를 가진 함수 선언 및 호출

```
// 원형 선언
void star(int a=5); //디폴트 매개변수 선언
void msg(int id, string text=""); //디폴트 매개변수 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;
}
int main() {
    star();      //star(5)
    star(10);

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

# 함수 중복 간소화

- 디폴트 매개 변수의 장점 - 함수 중복 간소화

```
class Circle {
    .....
public:
    Circle() { radius = 1; }
    Circle(int r) { radius = r; }
    .....
};
```

2 개의 생성자 함수를  
디폴트 매개 변수를  
가진 하나의 함수로  
간소화

```
class Circle {
    .....
public:
    Circle(int r=1) { radius = r; }
    .....
};
```

- 중복 함수들과 디폴트 매개 변수를 가진 함수를 함께 사용 불가

```
class Circle {
    .....
public:
    Circle() { radius = 1; }
    Circle(int r) { radius = r; }
    Circle(int r=1) { radius = r; }
    .....
};
```

중복된  
함수와 동시  
사용 불가

# 디폴트 매개 변수를 이용하여 중복 함수 간소화 연습

- 다음 두 개의 중복 함수를 디폴트 매개 변수를 가진 하나의 함수로 작성하라.

```
void fillLine() { // 25 개의 '*' 문자를 한 라인에 출력
    for(int i=0; i<25; i++) cout << '*';
    cout << endl;
}
```

```
void fillLine(int n, char c) { // n개의 c 문자를 한 라인에 출력
    for(int i=0; i<n; i++) cout << c;
    cout << endl;
}
```

```
void fillLine(int n=25, char c='*') { // n개의 c 문자를 한 라인에 출력
    for(int i=0; i<n; i++) cout << c;
    cout << endl;
}
```

```
int main() {
    fillLine(); // 25개의 '*'를 한 라인에 출력
    fillLine(10, '%'); // 10개의 '%'를 한 라인에 출력
}
```

# 생성자 함수의 중복 간소화

- 다음 클래스에 중복된 생성자를 디폴트 매개 변수를 가진 하나의 생성자로 작성하라.

```
class MyVector{
    int *p;
    int size;
public:
    MyVector() {
        p = new int [100];
        size = 100;
    }
    MyVector(int n) {
        p = new int [n];
        size = n;
    }
    ~MyVector() { delete [] p; }
};
```

```
class MyVector{
    int *p;
    int size;
public:
    MyVector(int n=100) : size(n), p(new int[n] { }
    ~MyVector() { delete [] p; }
};

int main() {
    MyVector *v1, *v2;
    v1 = new MyVector(); // 디폴트로 정수 100개의 배열 동적 할당
    v2 = new MyVector(1024); // 정수 1024개의 배열 동적 할당
    delete v1;
    delete v2;
}
```

# 함수 중복의 모호성

- 함수 중복이 모호하여 컴파일러가 어떤 함수를 호출하는지 판단하지 못하는 경우
  - 형 변환으로 인한 모호성
  - 참조 매개 변수로 인한 모호성
  - 디폴트 매개 변수로 인한 모호성



# 함수 중복의 모호성 - 형 변환

- 매개 변수의 형 변환으로 인한 중복 함수 호출의 모호성

```
double square(double a) {
    return a*a;
}
int main() {
    cout << square(3);
}
```

(a) 정상 컴파일

int 타입 3이 double 로 자동 형 변환

(b) 모호한 호출, 컴파일 오류

```
float square(float a) {
    return a*a;
}
double square(double a) {
    return a*a;
}
int main() {
    cout << square(3.0);
    cout << square(3);
}
```

3.0은 double 타입이므로  
모호하지 않음

int 타입 3을 double로  
변환할지 float로 변환할 지  
모호함

# 함수 중복의 모호성 - 참조 매개 변수

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

```
int add(int a, int &b) {  
    b = b + a;  
    return b;  
}
```

```
int main(){  
    int s=10, t=20;  
    cout << add(s, t); // 컴파일 오류  
}
```

두 함수는 근본적으로  
중복 시킬 수 없다.

call by value인지  
call by reference인지 모호

# 함수 중복의 모호성 - 디폴트 매개 변수

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

```
void msg(int id) {
    cout << id << endl;
}
```

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

```
int main(){
    msg(5, "Good Morning"); // 정상 컴파일. 두 번째 msg() 호출
    msg(6);                // 함수 호출 모호. 컴파일 오류
}
```

# static 멤버와 non-static 멤버의 특성

- static
  - 변수와 함수에 대한 기억 부류의 한 종류
    - 생명 주기 – 프로그램이 시작될 때 생성, 프로그램 종료 시 소멸
    - 사용 범위 – 선언된 범위, 접근 지정에 따름
- 클래스의 멤버
  - static 멤버
    - 프로그램이 시작할 때 생성
    - 클래스 당 하나만 생성, 클래스 멤버라고 불림
    - 클래스의 모든 인스턴스(객체)들이 공유하는 멤버
  - non-static 멤버
    - 객체가 생성될 때 함께 생성
    - 객체마다 객체 내에 생성
    - 인스턴스 멤버라고 불림

# static 멤버 선언

- 멤버의 static 선언

```
class Person {
public:
    double money;           // 개인 소유의 돈, non-static 멤버 선언
    void addMoney(int money) {
        this->money += money;
    }
    static int sharedMoney; // static 멤버 변수 선언, 공금
    static void addShared(int n) { // static 멤버 함수 선언
        sharedMoney += n;
    }
};
int Person::sharedMoney = 10; // sharedMoney를 10으로 초기화
```

**static 변수 공간 할당.**  
프로그램의 전역 공간에 선언

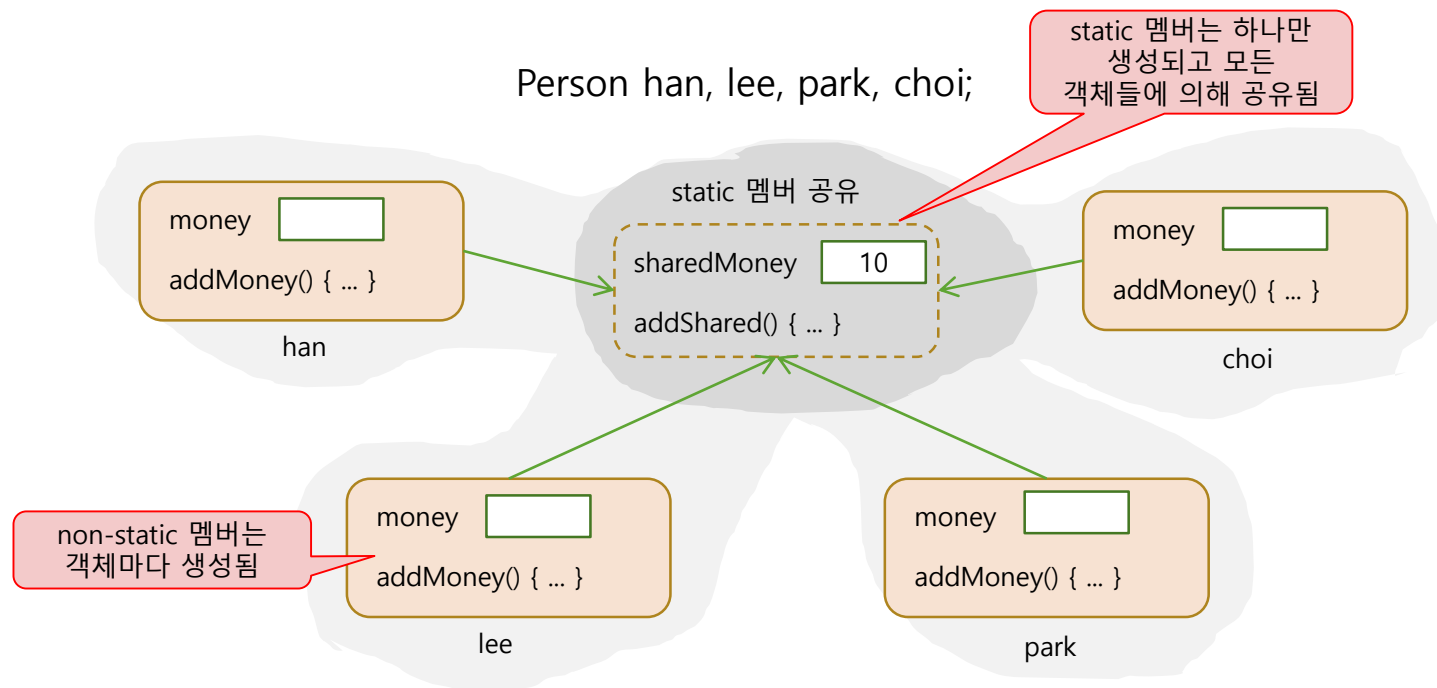
- static 멤버 변수 생성

- 전역 변수로 생성
- 전체 프로그램 내에 한 번만 생성

- static 멤버 변수에 대한 **외부 선언**이 없으면 링크 오류 발생

//C++ 17부터 static 데이터 멤버를 inline으로 선언, 프로그램 전역 공간에 선언할 필요 없음  
**inline** static int sharedMoney;

# static 멤버와 non-static 멤버의 관계



- han, lee, park, choi 등 4 개의 Person 객체 생성
- sharedMoney와 addShared() 함수는 하나만 생성되고 4 개의 객체들의 의해 공유됨
- sharedMoney와 addShared() 함수는 han, lee, park, choi 객체들의 멤버임

# static 멤버와 non-static 멤버 비교

항목	non-static 멤버	static 멤버
선언 사례	<pre>class Sample {     int n;     void f(); };</pre>	<pre>class Sample {     static int n;     static void f(); };</pre>
공간 특성	<p>멤버는 객체마다 별도 생성</p> <ul style="list-style-type: none"> <li>• 인스턴스 멤버라고 부름</li> </ul>	<p>멤버는 클래스 당 하나 생성</p> <ul style="list-style-type: none"> <li>• 멤버는 객체 내부가 아닌 별도의 공간에 생성</li> <li>• 클래스 멤버라고 부름</li> </ul>
시간적 특성	<p>객체와 생명을 같이 함</p> <ul style="list-style-type: none"> <li>• 객체 생성 시에 멤버 생성</li> <li>• 객체 소멸 시 함께 소멸</li> <li>• 객체 생성 후 객체 사용 가능</li> </ul>	<p>프로그램과 생명을 같이 함</p> <ul style="list-style-type: none"> <li>• 프로그램 시작 시 멤버 생성</li> <li>• 객체가 생기기 전에 이미 존재</li> <li>• 객체가 사라져도 여전히 존재</li> <li>• 프로그램이 종료될 때 함께 소멸</li> </ul>
공유의 특성	<p>공유되지 않음</p> <ul style="list-style-type: none"> <li>• 멤버는 객체 별로 따로 공간 유지</li> </ul>	<p>동일한 클래스의 모든 객체들에 의해 공유됨</p>

# static 멤버 사용 : 객체의 멤버로 접근

- static 멤버는 객체 이름이나 객체 포인터로 접근
  - 보통 멤버처럼 접근할 수 있음

```
객체.static멤버  
객체포인터->static멤버
```

- Person 타입의 객체 lee와 포인터 p를 이용하여 static 멤버를 접근하는 예

```
Person lee;  
lee.sharedMoney = 500; // 객체.static멤버 방식  
  
Person *p;  
p = &lee;  
p->addShared(200);    // 객체포인터->static멤버 방식
```



# static 멤버 사용 : 객체의 멤버로 접근

```
class Person {  
public:  
    double money;           // 개인 소유의 돈  
    void addMoney(int money) { this->money += money; }  
    static int sharedMoney; // 공금  
    static void addShared(int n) { sharedMoney += n; }  
};
```

```
int Person::sharedMoney=10; //static 변수 생성, 전역 공간에 생성 10으로 초기화
```

```
int main() {  
    Person han;  
    han.money = 100;           // han의 개인 돈=100  
    han.sharedMoney = 200; // static 멤버 접근, 공금=200  
  
    Person lee;  
    lee.money = 150;           // lee의 개인 돈=150  
    lee.addMoney(200);         // lee의 개인 돈=350  
    lee.addShared(200);        // static 멤버 접근, 공금=400  
  
    cout << han.money << ' ' << lee.money << endl;  
    cout << han.sharedMoney << ' ' << lee.sharedMoney << endl;  
}
```

# static 멤버 사용 : 클래스명과 범위 지정 연산자(::)로 접근

- 클래스 이름과 범위 지정 연산자(::)로 접근 가능
  - static 멤버는 클래스마다 오직 한 개만 생성되기 때문

클래스명::static멤버

```
han.sharedMoney = 200; <-> Person::sharedMoney = 200;  
lee.addShared(200);      <-> Person::addShared(200);
```

- non-static 멤버는 클래스 이름을 사용하여 접근 불가

```
// 컴파일 오류. non-static 멤버는 클래스 명으로 접근불가  
Person::money = 100;
```

```
// 컴파일 오류. non-static 멤버는 클래스 명으로 접근불가  
Person::addMoney(200);
```

# static 멤버 사용 : 클래스명과 범위 지정 연산자(::)로 접근

```
class Person {
public:
    double money;           // 개인 소유의 돈
    void addMoney(int money) { this->money += money; }
    static int sharedMoney; // 공금
    static void addShared(int n) { sharedMoney += n; }
};
```

int Person::sharedMoney=10; // 10으로 초기화, static 변수 생성. 전역 공간에 생성

```
int main() {
    Person::addShared(50); // static 멤버 접근, 공금=60, han 객체가 생기기전부터 static 멤버 접근
    cout << Person::sharedMoney << endl;

    Person han;
    han.money = 100;
    han.sharedMoney = 200; // static 멤버 접근, 공금=200
    Person::sharedMoney = 300; // static 멤버 접근, 공금=300
    Person::addShared(100); // static 멤버 접근, 공금=400

    cout << han.money << ' ' << Person::sharedMoney << endl;
}
```

# static 활용

- static의 주요 활용
  - 전역 변수나 전역 함수를 클래스에 캡슐화
    - 전역 변수나 전역 함수를 가능한 사용하지 않도록
    - 전역 변수나 전역 함수를 static으로 선언하여 클래스 멤버로 선언
  - 객체 사이에 공유 변수를 만들고자 할 때
    - static 멤버를 선언하여 모든 객체들이 공유

# static 멤버를 가진 Math 클래스 작성

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

int abs(int a) { return a>0?a:-a; }
int max(int a, int b) { return a>b?a:b; }
int min(int a, int b) { return (a>b)?b:a; }

int main() {
    cout << abs(-5) << endl;
    cout << max(10, 8) << endl;
    cout << min(-3, -8) << endl;
}
```

(a) 전역 함수들을 가진 좋지 않은 코딩 사례

- 왼쪽 코드를 static 멤버를 가진 Math 클래스로 작성하고 멤버 함수를 호출하라.

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

class Math {
public:
    static int abs(int a) { return a>0?a:-a; }
    static int max(int a, int b) { return (a>b)?a:b; }
    static int min(int a, int b) { return (a>b)?b:a; }
};

int main() {
    cout << Math::abs(-5) << endl;
    cout << Math::max(10, 8) << endl;
    cout << Math::min(-3, -8) << endl;
}
```

(b) Math 클래스를 만들고 전역 함수들을 static 멤버로 캡슐화한 프로그램

# static 멤버를 공유의 목적으로 사용하는 예

```
class Circle {
private:
    static int numOfCircles;    int radius;
public:
    Circle(int r = 1);
    ~Circle() { numOfCircles--; } // 생성된 원의 개수 감소
    static int getNumOfCircles() { return numOfCircles; }
};

Circle::Circle(int r) : radius(r) { numOfCircles++; } // 생성된 원의 개수 증가
int Circle::numOfCircles = 0; // 0으로 초기화

int main() {
    Circle *p = new Circle[10]; // 10개의 생성자 실행
    cout << "생존하고 있는 원의 개수 = " << Circle::getNumOfCircles() << endl;

    delete[] p; // 10개의 소멸자 실행
    cout << "생존하고 있는 원의 개수 = " << Circle::getNumOfCircles() << endl;

    Circle a; // 생성자 실행
    cout << "생존하고 있는 원의 개수 = " << Circle::getNumOfCircles() << endl;
    Circle b; // 생성자 실행
    cout << "생존하고 있는 원의 개수 = " << Circle::getNumOfCircles() << endl;
}
```

# static 멤버 함수는 static 멤버만 접근 가능

- static 멤버 함수가 접근할 수 있는 것
  - static 멤버 함수
  - static 멤버 변수
  - 함수 내의 지역 변수
- static 멤버 함수는 non-static 멤버에 접근 불가
  - 객체가 생성되지 않은 시점에서 static 멤버 함수가 호출될 수 있기 때문

# static 멤버 함수 getMoney()가 non-static 멤버 변수 money를 접근하는 오류

```
class PersonError {
    int money;
public:
    static int getMoney() {
        //컴파일 오류: static 멤버함수는 non-static 멤버접근 불가
        return money; }

```

```
    void setMoney(int money) { // 정상 코드
        this->money = money;
    }
};

```

```
int main(){
    int n = PersonError::getMoney();

    PersonError errorKim;
    errorKim.setMoney(100);
}

```

main()이 시작하기 전

```
static int getMoney()
{
    return money;
}

```

money는 아직  
생성되지 않았음.

n = PersonError::getMoney();

```
static int getMoney()
{
    return money;
}

```

생성되지 않는 변수를 접근하게  
되는 오류를 범함

PersonError errorKim;

errorKim

```
static int getMoney()
{
    return money;
}

```

money   
setMoney() { ... }

errorKim 객체가 생길 때  
money가 비로소 생성됨



# non-static 멤버 함수는 static에 접근 가능

```
class Person {  
    public:  
        double money;    // 개인 소유의 돈  
        static int sharedMoney; // 공금  
  
    ....  
    int total() {  
        // non-static 함수는 non-static이나 static 멤버에 모두 접근 가능  
        return money + sharedMoney;  
    }  
};
```

# static 멤버 함수는 this 사용 불가

- static 멤버 함수는 객체가 생기기 전부터 호출 가능
  - static 멤버 함수에서 this 사용 불가

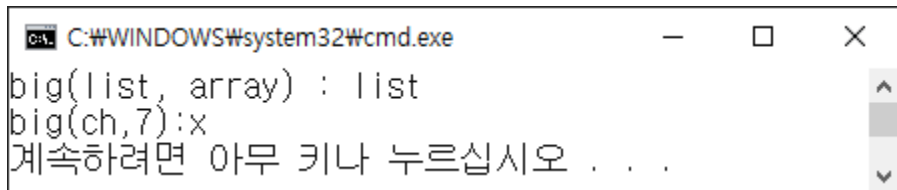
```
class Person {  
public:  
    double money;           // 개인 소유의 돈  
    static int sharedMoney; // 공금  
    ....  
    static void addShared(int n) { // static 함수에서 this 사용 불가  
        this->sharedMoney += n; // this를 사용하므로 컴파일 오류  
    }  
};
```



sharedMoney += n;  
으로 하면 정상 컴파일

# 활용 1 - 함수 중복

- 큰 값을 리턴하는 big 함수를 중복 구현하라.



```
C:\WINDOWS\system32\cmd.exe
big(list, array) : list
big(ch,7):x
계속하려면 아무 키나 누르십시오 . . .
```

```
class Big {
public:
    // a와 b 중 큰 문자열 리턴
    // 배열 a에 저장된 문자 중 아스키 코드 값이 큰 문자의 인덱스 리턴
};
```

```
char ch[] = { 'd', 'a', 'e', 'A', 'p', 'x', 'q' };
cout<<big("list", "array");
cout<<big(ch, 7);
```

## 활용 2 - 디폴트 매개 변수 활용

- 디폴트 매개변수를 갖는 하나의 add() 함수를 작성하고 프로그램을 완성 하시오

```
int main() {  
    int a[] = {1,2,3,4,5};  
    int b[] = {6,7,8,9,10};  
    int c = add(a, 5); // 배열 a의 정수를 모두 더한 값 리턴  
    int d = add(a, 5, b); // 배열 a와 b의 정수를 모두 더한 값 리턴  
    cout << c << endl; // 10 출력  
    cout << d << endl; // 55 출력  
}
```

## 활용 3 – static 멤버 활용

- 2명의 사람이 번갈아 숨겨진 답을 찾는 사람이 이긴다.
- 찾아야 할 답은 난수로 생성하며 static 멤버로 구성하는 UpAndDownGame 클래스
- 선수를 다루기 위한 Person 클래스를 구현하여 처리하도록 한다. 난수의 범위 1~100

```

C:\WINDOWS\system32\cmd.exe
Up & Down 게임을 시작합니다.
이 난수를 입력하세요 >> 양현수
이 난수를 입력하세요 >> 김인자
난수 1과 100사이에 있습니다.
양현수>>60
난수 1과 60사이에 있습니다.
김인자>>50
난수 50과 60사이에 있습니다.
양현수>>55
난수 50과 55사이에 있습니다.
김인자>>54
난수 50과 54사이에 있습니다.
양현수>>52
양현수가 이겼습니다!!
계속하려면 아무 키나 누르십시오 . . .
  
```

# Q & A

- “함수 중복과 static 멤버”에 대한 학습이 끝났습니다.
- 모든 내용을 이해 하셨나요?
- 아직 이해가 안되는 내용이 있다면 다시 한번 복습하시기 바랍니다.
- 질문은 한림 SmartLEAD 쪽지 또는 e-mail 또는 전화상담을 이용하시기 바랍니다.
- 과제와 퀴즈(제한 기간내 2회 응시 가능)가 있습니다.
- 다음에는 프렌드 함수와 연산자 중복에 대하여 학습하도록 하겠습니다.
- 수고하셨습니다.^^