



# 고급프로그래밍

static  
friend

연산자 중복

Professor Jeong, Mun-Ho

Robot Vision & Intelligence Laboratory  
Kwangwoon University  
(02-940-5625, mhjeong@kw.ac.kr)

# 실습 3

- 주어진 클래스에서 디폴트 매개변수를 가진 하나의 생성자로 수정하시오

```
class MyVector
{
    int *mem;
    int size;
public:
    MyVector();
    MyVector(int n, int val);
    ~MyVector() { delete [] mem; }
};
```

```
MyVector::MyVector() {
    mem = new int [100];
    size = 100;
    for(int i=0; i<size; i++) mem[i] = 0;
}
```

```
MyVector::MyVector(int n, int val) {
    mem = new int [n];
    size = n;
    for(int i=0; i<size; i++) mem[i] = val;
}
```

```
int main() {
    MyVector a; // a(100, 0);과 동일
    MyVector b(10, 3);

    a.show(); // 100개의 0이 출력
    b.show(); // 10개의 3이 출력
}
```


## 실습 3 - 답

- 주어진 클래스에서 디폴트 매개변수를 가진 하나의 생성자로 수정하시오

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

class MyVector
{
    int *mem;
    int size;
public:
    MyVector(int n=100, int val=0)
    {
        mem = new int[n];
        size = n;
        for(int i=0; i<size; i++)
            mem[i] = val;
    }
    ~MyVector(){ delete[] mem; }
```

# Schedule

week	Topics		Homework	Quiz
1	과목소개	교과목 소개 (1), C++ 시작 (2)		
2	C++ 	C++ 프로그래밍의 기본(3, 3/12), 클래스와 객체(4, 3/14)	1	1
3		휴강(3/17), 객체생성과 사용(5, 3/19)	2	
4		함수와 참조(6, 3/26), 복사 생성자와 함수중복(7. 3/28)	3	2, 3
5		static, friend, 연산자 중복(8, 4/2), 상속가상함수와 추상클래스	4	4
6		템플릿과 STL, 표준 입출력	5	5
7		파일 입출력		
8	중간고사			
9	C++	예외처리 및 C 사용, 람다식	6	6
10		멀티스레딩	7	7
11		멀티스레딩, 고급문법	8	8
12		고급문법	9	9
13	병렬 프로그래밍	병렬프로그래밍		
14		병렬프로그래밍		
15	기말고사			

# 오늘의 학습내용

- static
- friend
- 연산자 중복(operator overloading)

static

# static 멤버와 non-static 멤버

## ■ 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  
    void addMoney(int money) {  
        this->money += money;  
    }  
  
    static int sharedMoney; // 공금  
    static void addShared(int n) {  
        sharedMoney += n;  
    }  
};  
  
int Person::sharedMoney = 10; // sharedMoney를 10으로 초기화
```

non-static 멤버 선언

static 멤버 변수 선언

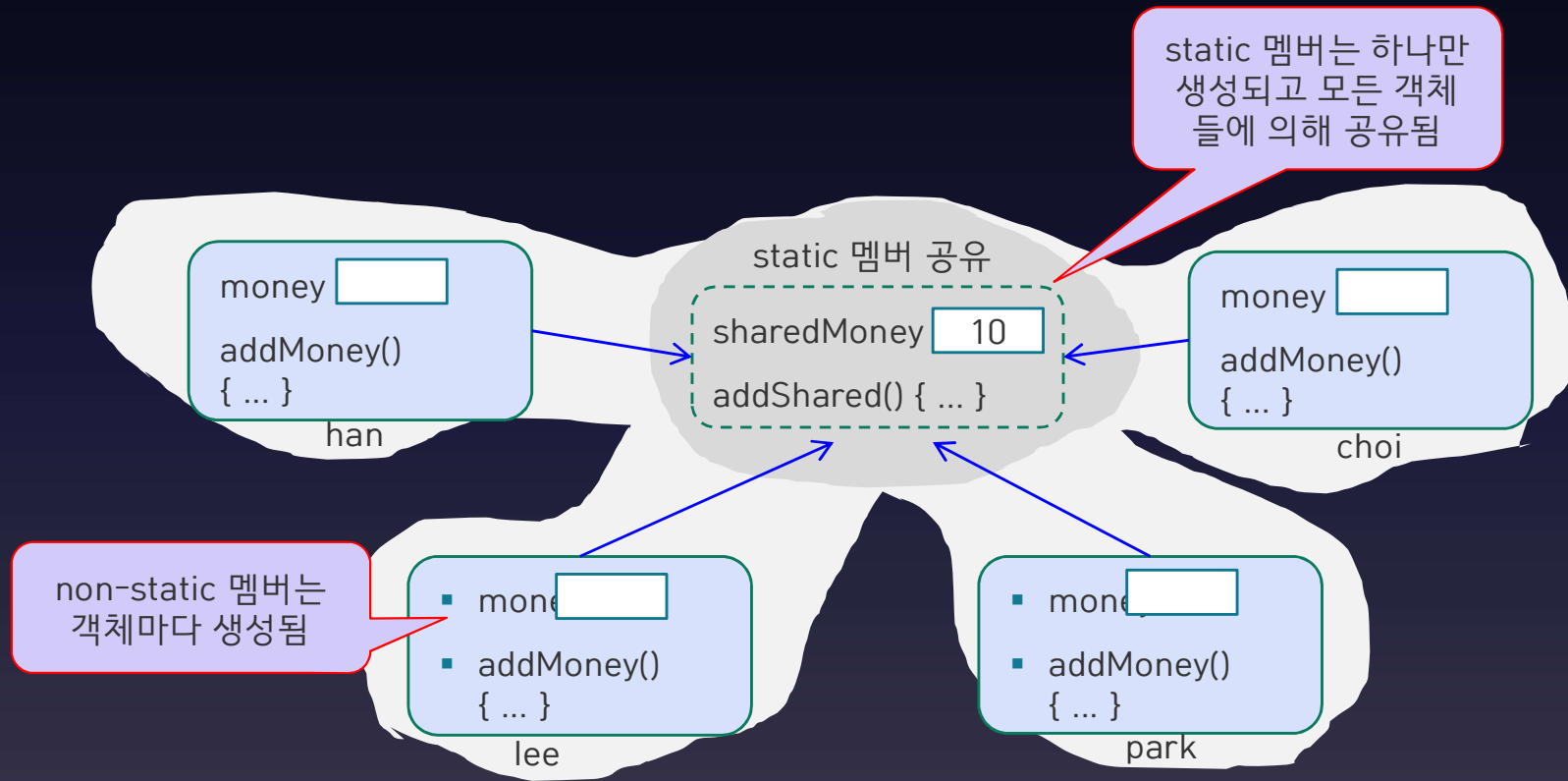
static 멤버 함수 선언

static 변수 공간 할당.  
프로그램의 전역 공간에 선언

- ## ■ static 멤버 변수 생성
- 전역 변수로 생성
  - 외부선언 필수 (링크오류)



# static 멤버와 non-static 멤버 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 멤버 사용

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

```
객체.static멤버  
객체포인터->static멤버
```

- Person 타입의 객체 *lee*와 포인터 *p*를 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

- 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 접근 불가

Person::money = 100; // 컴파일 오류. non-static 멤버는 클래스 명으로 접근불가  
Person::addMoney(200); // 컴파일 오류. non-static 멤버는 클래스 명으로 접근불가

# static 활용

## ■ static의 주요 활용

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

# 예제

## ■ static 멤버를 가진 Math 클래스로 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;
}
```

전역 함수들을 가진 좋지 않음 코딩 사례

```
#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;
}
```

5  
10  
-8

(Math 클래스를 만들고 전역 함수들을 static 멤버로 캡슐화한 프로그램)

# 예제

생존하고 있는 원의 개수 = 10  
생존하고 있는 원의 개수 = 0  
생존하고 있는 원의 개수 = 1  
생존하고 있는 원의 개수 = 2

생성자가 10번 실행되어  
numOfCircles = 10 이 됨

numOfCircles = 0 이 됨

numOfCircles = 1 이 됨

numOfCircles = 2 가 됨

```
class Circle {
private:
    static int numOfCircles;
    int radius;
public:
    Circle(int r=1);
    ~Circle() { numOfCircles--; } // 생성된 원의 개수 감소
    double getArea() { return 3.14*radius*radius;}
    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 함수의 접근성

- 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

# 예제

```
class PersonError {  
    int money;  
public:  
    static int getMoney() { return money; }  
  
    void setMoney(int money) { // 정상 코드  
        this->money = money;  
    }  
};  
  
int main(){  
    int n = PersonError::getMoney();  
  
    PersonError errorKim;  
    errorKim.setMoney(100);  
}
```

컴파일 오류  
static 멤버 함수는 non-static 멤버  
에 접근할 수 없음.



# non-static 멤버 함수의 접근성

```
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;  
    }  
};
```

non-static

static

# 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

프렌드

# C++ 프렌드

## ■ 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 선언 3 종류

- 외부 함수 equals()를 Rect 클래스에 프렌드로 선언

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

- RectManager 클래스의 equals() 멤버 함수를 Rect 클래스에 프렌드로 선언

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

- RectManager 클래스의 모든 멤버 함수를 Rect 클래스에 프렌드로 선언

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

# 예제

```
class Rect:
```

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

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

```
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;
```

```
}
```

not equal

객체 a와 b는 동일한 크기의 사각형이므로  
"not equal" 출력

# 예제

```
class Rect;
```

```
class RectManager { // RectManager 클래스 선언
public:
    bool equals(Rect r, Rect s);
};
```

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

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

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

```
int main() {
    Rect a(3,4), b(3,4);
    RectManager man;
```

```
    if(man.equals(a, b)) cout << "equal" << endl;
    else cout << "not equal" << endl;
```

```
}
```

equal

객체 a와 b는 동일한 크기의 사각형이므로 "equal" 출력

RectManager 클래스의 equals() 멤버를 프렌드로 선언

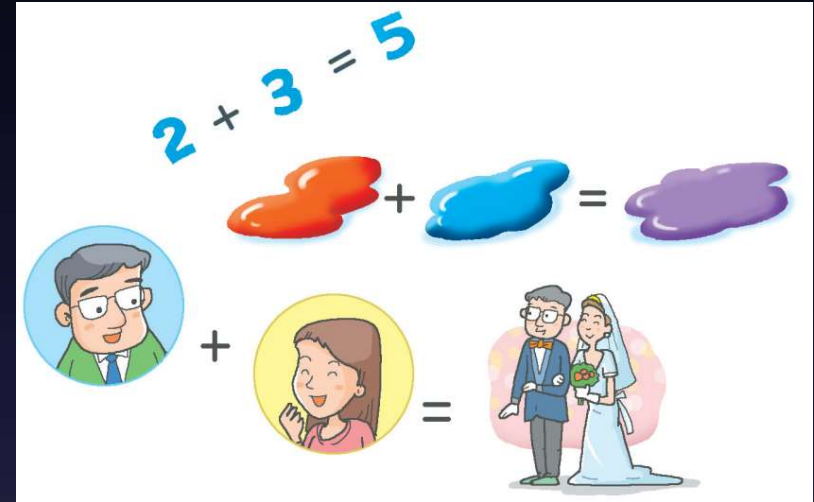
연산자 중복



# 연산자 중복

## ■ 일상 생활에서의 기호 사용

- + 기호의 사례
  - 숫자 더하기 :  $2 + 3 = 5$
  - 색 혼합 : 빨강 + 파랑 = 보라
  - 생활 : 남자 + 여자 = 결혼
- + 기호를 숫자와 물체에 적용, 중복 사용
- + 기호를 숫자가 아닌 곳에도 사용
- 간결한 의미 전달
- 다형성



## ■ C++ 언어에서도 연산자 중복(operator overloading) 가능

- 본래 있던 연산자에 새로운 의미 정의
- 높은 프로그램 가독성

# 연산자 중복의 사례 : + 연산자

- 정수 더하기

```
int a=2, b=3, c;  
c = a + b; // + 결과 5. 정수가 피연산자일 때 2와 3을 더하기
```

- 문자열 합치기

```
string a="C", c;  
c = a + "++"; // + 결과 "C++". 문자열이 피연산자일 때 두 개의 문자열 합치기
```

- 색 섞기

```
Color a(BLUE), b(RED), c;  
c = a + b; // c = VIOLET. a, b의 두 색을 섞은 새로운 Color 객체 c
```

- 배열 합치기

```
SortedArray a(2,5,9), b(3,7,10), c;  
c = a + b; // c = {2,3,5,7,9,10}. 정렬된 두 배열을 결합한(merge) 새로운 배열 생성
```

# 연산자 중복의 특징

- C++에 본래 있는 연산자만 중복 가능
  - 3%%5 // 컴파일 오류
  - 6### 7 // 컴파일 오류
- 피 연산자 타입이 다른 새로운 연산 정의
- 연산자는 함수 형태로 구현 - 연산자 함수(operator function)
- 반드시 클래스와 관계를 가짐
- 피연산자의 개수를 바꿀 수 없음
- 연산의 우선 순위 변경 안됨
- 모든 연산자가 중복 가능하지 않음

중복 가능한 연산자

+	-	*	/	%	^	&
	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	=
<<	>>	>>=	<<=	==	!=	>=
<=	&&		++	--	->*	,
->	[]	()	new	delete	new[]	delete[]

중복 불가능한 연산자

.	.*	::(범위지정 연산자)	? : (3항 연산자)
---	----	--------------	--------------

# 연산자 함수

- 연산자 함수 구현 방법 2 가지
  1. 클래스의 멤버 함수로 구현
  2. 외부 함수로 구현하고 클래스에 프렌드 함수로 선언
- 연산자 함수 형식

```
리턴타입 operator연산자(매개변수리스트);
```

# +와 == 연산자의 작성 사례

연산자 함수 작성에 필요한 코드 사례

```
Color a(BLUE), b(RED), c;
```

```
c = a + b; // a와 b를 더하기 위한 + 연산자 작성 필요  
if(a == b) { // a와 b를 비교하기 위한 == 연산자 작성 필요  
    ...  
}
```

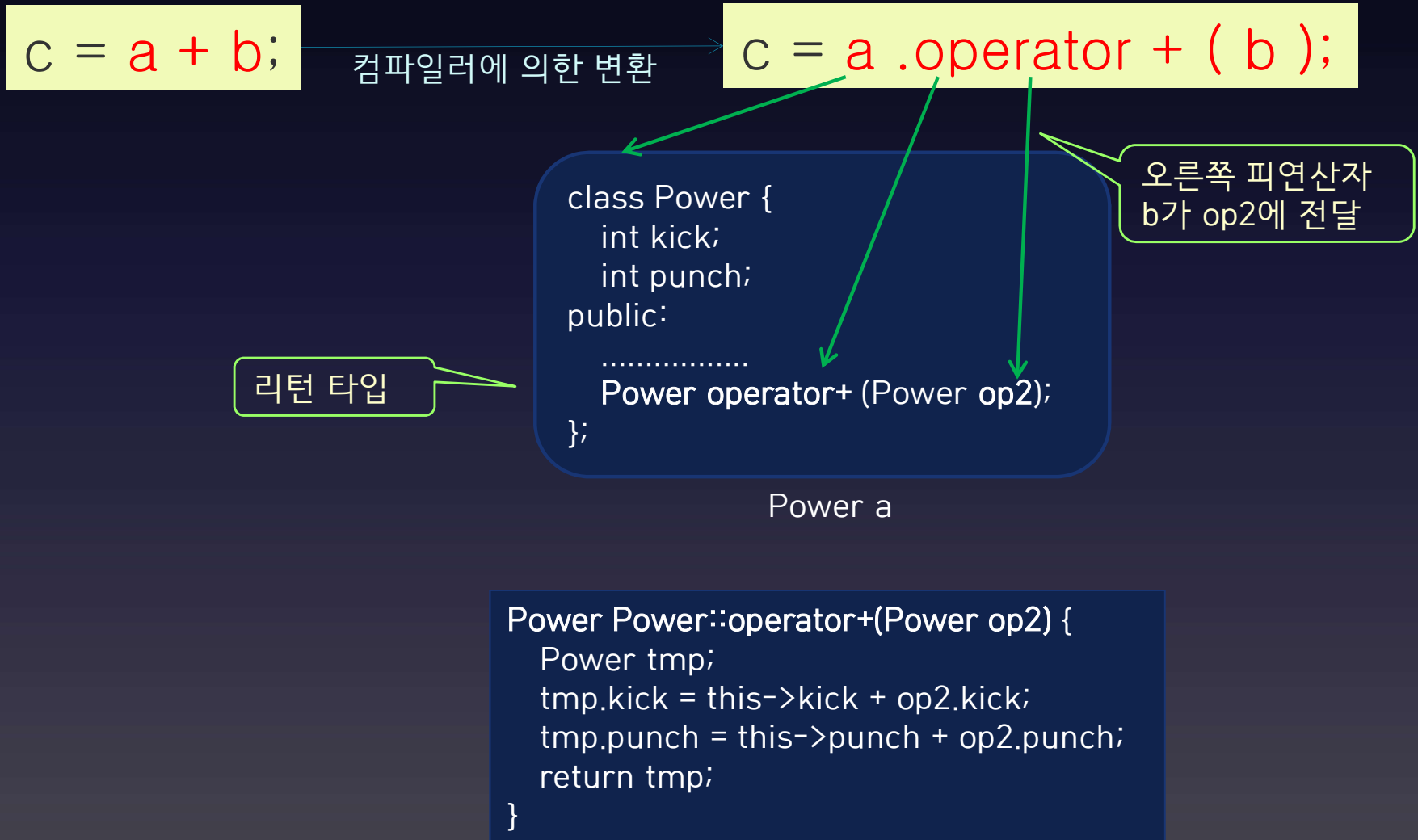
외부 함수로 구현되고 클래스에  
프렌드로 선언되는 경우

```
Color operator + (Color op1, Color op2); // 외부 함수  
bool operator == (Color op1, Color op2); // 외부 함수  
  
class Color {  
    ...  
    friend Color operator+ (Color op1, Color op2);  
    friend bool operator== (Color op1, Color op2);  
};
```

클래스의 멤버 함수로 작성되는 경우

```
class Color {  
    ...  
    Color operator + (Color op2);  
    bool operator == (Color op2);  
};
```

# 이항 연산자 중복 : + 연산자



# 예제

```
class Power {
    int kick;
    int punch;
public:
    Power(int kick=0, int punch=0) {
        this->kick = kick; this->punch = punch;
    }
    void show();
    Power operator+ (Power op2); // + 연산자 함수 선언
};

void Power::show() {
    cout << "kick=" << kick << ', ' << "punch=" << punch
    << endl;
}

Power Power::operator+(Power op2) {
    Power tmp; // 임시 객체 생성
    tmp.kick = this->kick + op2.kick; // kick 더하기
    tmp.punch = this->punch + op2.punch; // punch 더하기
    return tmp; // 더한 결과 리턴
}
```

객체 a의 operator+()  
멤버함수 호출

```
int main() {
    Power a(3,5), b(4,6), c;
    c = a + b; // 파워 객체 + 연산
    a.show();
    b.show();
    c.show();
}
```

```
kick=3,punch=5
kick=4,punch=6
kick=7,punch=11
```

# 실습

- 다음과 같은 프로그램의 실행결과가 아래와 같이 나오도록 Point 클래스의 연산자를 정의하시오.

```
class Point {  
    int x;  
    int y;  
public:  
    Point(int x1=0, int y1=0) {  
        x = x1; y = y1;  
    }  
    void show( ){  
        cout << "(" << x << "," << y << ")" << endl;  
    }  
};
```

```
int main( )  
{  
    Point a(3,5), b(2,0), c;  
    c = a + b;  
    c.show( );  
}
```

(5,5)



## 실습 - 답

- 다음과 같은 프로그램의 실행결과가 아래와 같이 나오도록 Point 클래스의 연산자를 정의하시오.

```
class Point {
    int x;
    int y;
public:
    Point(int x1=0, int y1=0) {
        x = x1; y = y1;
    }
    void show() {
        cout << "(" << x << "," << y << ")" << endl;
    }
    Point operator +(const Point& b);
};

Point Point::operator +(const Point& b)
{
    Point out;
    out.x = x + b.x;
    out.y = y + b.y;
    return out;
}
```

```
int main( )
{
    Point a(3,5), b(2,0), c;
    c = a + b;
    c.show( );
}
```

(5,5)

# $+=$ 연산자 중복

$c = a += b;$

컴파일러에 의한 변환

$c = a.operator += (b);$

class Power {

.....

public:

Power& operator+=(Power op2);  
};

리턴 타입

오른쪽 피연산자  
b가 op2에 전달

```
Power& Power::operator+=(Power op2)
{
    kick = kick + op2.kick;
    punch = punch + op2.punch;
    return *this; // 자신의 참조 리턴
}
```

# 예제

```
class Power {
    int kick;
    int punch;
public:
    Power(int kick=0, int punch=0) {
        this->kick = kick; this->punch = punch;
    }
    void show();
    Power& operator+=(Power op2); // += 연산자 함수
선언
};

void Power::show() {
    cout << "kick=" << kick << ',' << "punch="
<< punch << endl;
}

Power& Power::operator+=(Power op2) {
    kick = kick + op2.kick; // kick 더하기
    punch = punch + op2.punch; // punch 더하기
    return *this; // 합한 결과 리턴
}
```

+= 연산자 멤버 함수 구현

```
int main() {
    Power a(3,5), b(4,6), c;
    a.show();
    b.show();
    c = a += b; // 파워 객체 더하기
    a.show();
    c.show();
}
```

```
kick=3,punch=5
kick=4,punch=6
kick=7,punch=11
kick=7,punch=11
```

# + 연산자 : $b = a + 2$ ;

```
class Power {
    int kick;
    int punch;
public:
    Power(int kick=0, int punch=0) {
        this->kick = kick; this->punch = punch;
    }
    void show();
    Power operator+ (int op2); // + 연산자 함수 선언
};

void Power::show() {
    cout << "kick=" << kick << ',' << "punch=" << punch
    << endl;
}
```

+ 연산자 멤버 함수 구현

```
Power Power::operator+(int op2) {
    Power tmp; // 임시 객체 생성
    tmp.kick = kick + op2; // kick에 op2 더하기
    tmp.punch = punch + op2; // punch에 op2 더하기
    return tmp; // 임시 객체 리턴
}
```

```
int main( )
{
    Power a(3,5), b;
    a.show();
    b.show();
    b = a + 2; // 파워 객체와 정수 더하기
    a.show();
    b.show();
}
```

operator+(int) 함수 호출

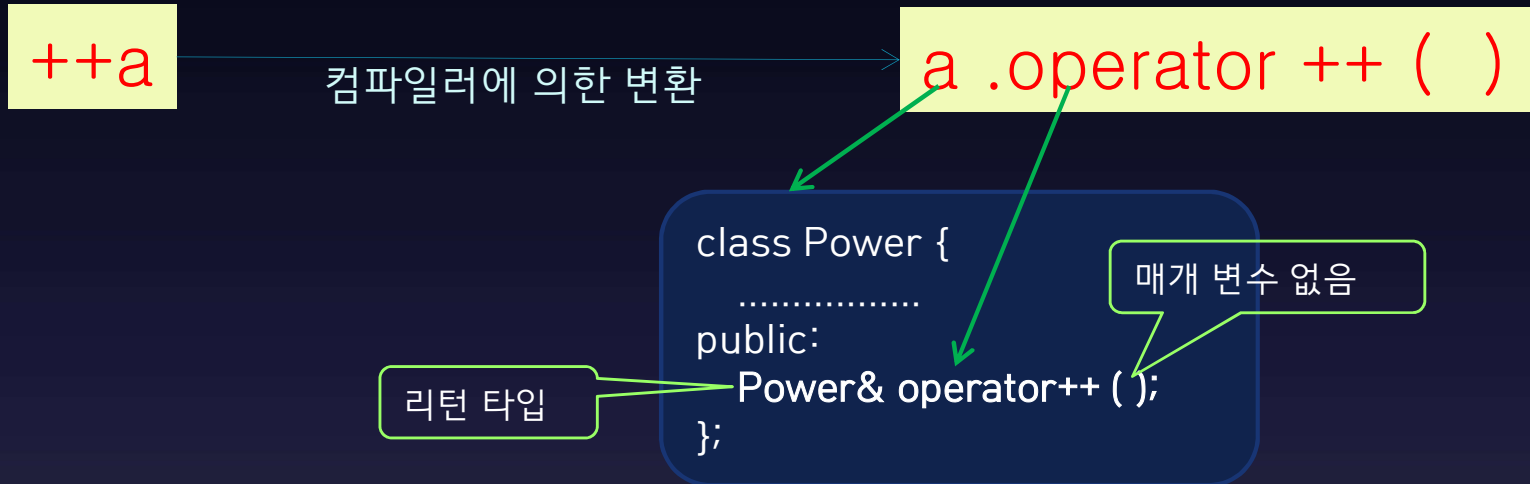
```
kick=3,punch=5
kick=0,punch=0
kick=3,punch=5
kick=5,punch=7
```

# 단항 연산자 중복

## ■ 단항 연산자

- 피연산자가 하나 뿐인 연산자
  - 연산자 중복 방식은 이항 연산자의 경우와 거의 유사함
- 단항 연산자 종류
  - 전위 연산자(prefix operator)
    - !op, ~op, ++op, --op
  - 후위 연산자(postfix operator)
    - op++, op--

# 전위 ++ 연산자 중복



```
Power& Power::operator++( )  
{  
    // kick과 punch는 a의 멤버  
    kick++;  
    punch++;  
    return *this; // 변경된 객체 자신(객체 a)의 참조 리턴  
}
```

# 예제

```
class Power {
    int kick;
    int punch;
public:
    Power(int kick=0, int punch=0) {
        this->kick = kick; this->punch = punch;
    }
    void show();
    Power& operator++ (); // 전위 ++ 연산자 함수 선언
};

void Power::show() {
    cout << "kick=" << kick << ',' << "punch=" << punch
    << endl;
}

Power& Power::operator++() {
    kick++;
    punch++;
    return *this; // 변경된 객체 자신(객체 a)의 참조 리턴
}
```

전위 ++ 연산자 멤버 함수 구현

```
int main() {
    Power a(3,5), b;
    a.show();
    b.show();
    b = ++a; // 전위 ++ 연산자 사용
    a.show();
    b.show();
}
```

operator++() 함수 호출

```
kick=3,punch=5
kick=0,punch=0
kick=4,punch=6
kick=4,punch=6
```

# 후위 연산자 중복, ++ 연산자

`a++`

컴파일러에 의한 변환

`a.operator++ ( 임의의 정수 )`

```
class Power {  
    .....  
public:  
    Power operator ++ (int x );  
};
```

```
Power Power::operator++(int x)  
{  
    Power tmp = *this; // 증가 이전 객체 상태 저장  
    kick++;  
    punch++;  
    return tmp; // 증가 이전의 객체(객체 a) 리턴  
}
```



# 예제

```
class Power {
    int kick;
    int punch;
public:
    Power(int kick=0, int punch=0) {
        this->kick = kick; this->punch = punch;
    }
    void show();
    Power operator++ (int x); // 후위 ++ 연산자 함수 선언
};

void Power::show() {
    cout << "kick=" << kick << ' '
        << "punch=" << punch << endl;
}

Power Power::operator++(int x) {
    Power tmp = *this; // 증가 이전 객체 상태를 저장
    kick++;
    punch++;
    return tmp; // 증가 이전 객체 상태 리턴
}
```

후위 ++ 연산자 멤버  
함수 구현

```
int main()
{
    Power a(3,5), b;
    a.show();
    b.show();
    b = a++; // 후위 ++ 연산자 사용
    a.show(); // a의 파워는 1 증가됨
    b.show(); // b는 a가 증가되기 이전 상태를 가짐
}
```

operator++(int) 함수 호출

```
kick=3,punch=5
kick=0,punch=0
kick=4,punch=6
kick=3,punch=5
```

# 2 + a 덧셈

Power a(3,4), b;  
b = 2 + a;

b = 2 + a;

① 변환 불가능

b = 2 . + ( a );

오른쪽  
피연산자

② 변환 가능

b = operator+ ( 2 , a );

외부 연산자  
함수명

왼쪽  
피연산자

b = 2 + a;

컴파일러에 의한 변환

b = operator+ ( 2 , a );

```
Power operator+ (int op1, Power op2) {  
    Power tmp;  
    tmp.kick = op1 + op2.kick;  
    tmp.punch = op1 + op2.punch;  
    return tmp;  
}
```

# 예제

```
class Power {  
    int kick;  
    int punch;  
public:  
    Power(int kick=0, int punch=0) {  
        this->kick = kick; this->punch = punch;  
    }  
    void show();  
    friend Power operator+(int op1, Power op2);  
};  
  
void Power::show() {  
    cout << "kick=" << kick << ' ' << "punch=" << punch  
    << endl;  
}  
  
Power operator+(int op1, Power op2) {  
    Power tmp; // 임시 객체 생성  
    tmp.kick = op1 + op2.kick; // kick 더하기  
    tmp.punch = op1 + op2.punch; // punch 더하기  
    return tmp; // 임시 객체 리턴  
}
```

private 속성인 kick, punch를 접근하도록 하기 위해, 연산자 함수를 friend로 선언해야 함

+ 연산자 함수를 외부 함수로 구현

```
int main( )  
{  
    Power a(3,5), b;  
    a.show();  
    b.show();  
    b = 2 + a; // 파워 객체 더하기 연산  
  
    a.show();  
    b.show();  
}
```

operator+(2, a) 함수 호출

```
kick=3,punch=5  
kick=0,punch=0  
kick=3,punch=5  
kick=5,punch=7
```

# + 연산자를 외부 프렌드 함수로 구현

`c = a + b;`

컴파일러에 의한 변환

`c = operator+ ( a , b );`

```
Power operator+ (Power op1, Power op2)
{
    Power tmp;
    tmp.kick = op1.kick + op2.kick;
    tmp.punch = op1.punch + op2.punch;
    return tmp;
}
```

# 예제

```
class Power {
    int kick;
    int punch;
public:
    Power(int kick=0, int punch=0) {
        this->kick = kick; this->punch = punch;
    }
    void show();
    friend Power operator+(Power op1, Power op2);
};

void Power::show() {
    cout << "kick=" << kick << ',' << "punch=" << punch
        << endl;
}

Power operator+(Power op1, Power op2) {
    Power tmp; // 임시 객체 생성
    tmp.kick = op1.kick + op2.kick; // kick 더하기
    tmp.punch = op1.punch + op2.punch; // punch 더하기
    return tmp; // 임시 객체 리턴
}
```

```
int main( )
{
    Power a(3,5), b(4,6), c;
    c = a + b; // 파워 객체 + 연산
    a.show();
    b.show();
    c.show();
}
```

```
kick=3,punch=5
kick=4,punch=6
kick=7,punch=11
```

# 단항 연산자 ++를 프렌드로 작성하기

(a) 전위 연산자

**++a**

컴파일러에 의한 변환

**operator++ ( a )**

```
Power& operator++ (Power& op) {  
    op.kick++;  
    op.punch++;  
    return op;  
}
```

0은 의미 없는 값으로  
전위 연산자와 구분하  
기 위함

(b) 후위 연산자

**a++**

컴파일러에 의한 변환

**operator ++ ( a, 0 )**

```
Power operator++ (Power& op, int x) {  
    Power tmp = op;  
    op.kick++;  
    op.punch++;  
    return tmp;  
}
```

# 예제

```
class Power {
    int kick;
    int punch;
public:
    Power(int kick=0, int punch=0) { this->kick = kick; this->punch = punch; }
    void show();
    friend Power& operator++(Power& op); // 전위 ++ 연산자 함수
    friend Power operator++(Power& op, int x); // 후위 ++ 연산자 함수
};
```

```
void Power::show() {
    cout << "kick=" << kick << ', ' << "punch=" << punch << endl;
}

Power& operator++(Power& op) { // 전위 ++ 연산자 함수 구현
    op.kick++;
    op.punch++;
    return op; // 연산 결과 리턴
}
```

```
Power operator++(Power& op, int x) { // 후위 ++ 연산자 함수 구현
    Power tmp = op; // 변경하기 전의 op 상태 저장
    op.kick++;
    op.punch++;
    return tmp; // 변경 이전의 op 리턴
}
```

```
int main()
{
    Power a(3,5), b;
    b = ++a; // 전위 ++ 연산자
    a.show(); b.show();

    b = a++; // 후위 ++ 연산자
    a.show(); b.show();
}
```

```
kick=4,punch=6
kick=4,punch=6
kick=5,punch=7
kick=4,punch=6
```

# 예제 - << 연산자

- Power 객체의 kick과 punch에 정수를 더하는 << 연산자를 멤버 함수로 작성하라

```
class Power {
    int kick;
    int punch;
public:
    Power(int kick = 0, int punch = 0) {
        this->kick = kick; this->punch = punch;
    }
    void show();
    Power& operator << (int n); // 연산 후 Power 객체의 참조 리턴
};

void Power::show() {
    cout << "kick=" << kick << ',' << "punch=" << punch << endl;
}

Power& Power::operator <<(int n) {
    kick += n;
    punch += n;
    return *this; // 이 객체의 참조 리턴
}
```

```
int main() {
    Power a(1, 2);
    a << 3 << 5 << 6;
    a.show();
}
```

객체 a에 3, 5, 6이  
순서대로 더해진다

kick=15,punch=16



# 연산자 중복 : [ ]

```
#include <iostream>
using namespace std;
class Complex
{
private :
    int real, image;

public :
    Complex(int r=0, int i=0) : real(r), image(i) { };
    ~ Complex( );
    void show(Complex& b);
    Complex operator-(const Complex& C) const;
    Complex operator-( ) const;
    int operator [ ](const int& idx) const;
    int& operator [ ](const int& idx);
};

void Complex::show(Complex& b)
{
    cout << b.real<<"+"<< b.image<<"j" << endl;
}

Complex Complex::operator -(const Complex& C) const
{
    return Complex(real-C.real, image-C.image);
}
```

```
Complex Complex::operator -( ) const {
    return Complex(-real, -image);
}

int Complex::operator [ ](const int& idx) const {
    return ( idx == 0 ? real : image);
}

int& Complex::operator [ ](const int& idx) {
    return ( idx == 0 ? real : image);
}

int main()
{
    Complex a(10,10), b;
    b[0] = 1.0; b[1] = 2.0;
    show(b);

    b = -a; b[0] = a[0];
    show(b);
}
```

## 실습

- 다음 프로그램이 가능하도록 `Circle` 클래스의 연산자를 프렌드 함수로 작성하시오

```
class Circle {
    int radius;
public:
    Circle(int radius = 0) { this->radius = radius; }
    void show() {
        cout << "radius = " << radius << " circle" << endl;
    }
};

int main() {
    Circle a(5), b(4);
    ++a; // 반지름을 1 증가 시킨다.
    b = a++; // 반지름을 1 증가 시킨다.
    a.show();
    b.show();
}
```

## 실습 - 답

- 다음 프로그램이 가능하도록 `Circle` 클래스의 연산자를 프렌드 함수로 작성하시오

```
class Circle {
    int radius;
public:
    Circle(int radius = 0) { this->radius = radius; }
    void show() {
        cout << "radius = " << radius << " 인 원" << endl;
    }

    friend Circle& operator ++(Circle& c);
    friend Circle operator ++(Circle& c, int x);
};

Circle& operator ++(Circle& c) { // 전위 ++. ++a를 위함
    c.radius++;
    return c;
}

Circle operator ++(Circle& c, int x) { // 후위 ++. a++를 위함
    Circle tmp = c;
    c.radius++;
    return tmp;
}
```

# 함수 객체(Function Object)

- 객체를 함수처럼 사용
- 연산자 중복으로 선언함: operator ( )

```
2 #include <iostream>
3 using namespace std;
4
5 class Plus
6 {
7 public:
8     int operator( )(int a, int b)
9     {
10         return a + b;
11     }
12 };
13
14 int main()
15 {
16     Plus pls;
17
18     cout << "pls(10, 20): " << pls(10, 20) << endl;
19     return 0;
20 }
```

explicit call: pls.operator()(10,20)

implicit call

# 함수 객체

- 사용하는 이유
  - 객체 멤버 활용
  - 일반함수 보다 호출이 빠름

```
2 #include <iostream>
3 using namespace std;
4
5 class MoneyBox
6 {
7     int total;
8
9 public:
10     MoneyBox(int _init = 0) : total(_init) { }
11
12     int operator( )(int money)
13     {
14         total += money;
15         return total;
16     }
17 };
18
19 int main()
20 {
21     MoneyBox mb;
22
23     cout << "mb(100): " << mb(100) << endl;
24     cout << "mb(500): " << mb(500) << endl;
25     cout << "mb(2000): " << mb(2000) << endl;
26
27     return 0;
28 }
```

