



# 게임 자료구조와 알고리즘

## -CHAPTER 15-

SOULSEEK

# 목차

- 1.** 연산자 오버로딩
- 2.** 함수 포인터
- 3.** 함수 객체
- 4.** 템플릿

# 연산자 오버로딩

# 1. 연산자 오버로딩

- 사용자 정의 타입(Class)에서도 연산자를 사용할 수 있게 하는 문법
- 사용자 정의 타입(Class)은 기본적으로 컴파일 내부 연산이 정의 되어 있지 않다.

**//Exam\_1**

**void main()**

**{**

**int n1 = 10, n2 = 20;**

**cout << n1 + n2 << endl;**

**}**

**//Exam2**

**//사용자가 정의한 클래스**

**class Point**

**{**

**....**

**};**

**void main()**

**{**

**Point p1(2, 3), p2(5, 5);**

**p1 + p2; // 컴파일러는 두 객체의 연산을 알고 있지 않다. 에러!!**

**}**

# 1. 연산자 오버로딩

- 사용자 정의 타입에서 연산을 실행하면 **operator연산자()** 형식의 함수를 호출하기 때문에 멤버 함수로 선언되어 있지 않으면 에러가 발생한다.

```
class Point
{
    int x;
    int y;

    public:

    Point(int _x = 0, int _y = 0) :x(_x), y(_y) { }

    void Print() const { cout << x << ',' << y << endl; }
};

void main()
{
    Point p1(2, 3), p2(5, 5);

    p1 + p2; // => p1.operator+( p2 ); 와 같다.
}
```

```
class Point
{
    int x;
    int y;

    public:

    Point(int _x = 0 , int _y = 0 ):x(_x),y(_y) { }
    void Print( ) { cout << x <<',' << y << endl; }

    void operator+( Point arg)
    {
        cout << "operator+() 함수 호출" << endl;
    }
};

void main( )
{
    Point p1(2,3), p2(5,5);

    p1+p2; // => p1.operator+( p2 ); 와 같다.
}
```

# 1. 연산자 오버로딩

- 사용자 정의 타입에서 연산을 실행 했을 때 취해야 할 행동을 미리 정해두는 함수가 필요하다.
  - ➔ 그냥 일반 함수지만 연산자처럼 사용하게 만들어 주는 것이다
  - ➔ 원래는 “ + ”를 하려고 만들었지만 실제 내부는 “ \* ”를 해주고 있어도 문제가 되지 않는다.

```
class Point
{
    int x;
    int y;

public:
    Point(int _x = 0, int _y = 0) :x(_x), y(_y) { }

    void Print() const { cout << x << ',' << y << endl; }

    void operator+(Point arg)
    {
        Point pt;
        pt.x = this->x + arg.x;
        pt.y = this->y + arg.y;

        cout << "X : " << pt.x << ", Y : " << pt.y << endl;
    }
};

void main()
{
    Point p1(2, 3), p2(5, 5);

    p1 + p2; // => p1.operator+( p2 ); 와 같다.
}
```

# 1. 연산자 오버로딩

```
class Point
{
    int x;
    int y;

    public:
    Point(int _x = 0, int _y = 0) :x(_x), y(_y) { }

    void Print() const { cout << x << ',' << y << endl; }

    const Point operator+(const Point& arg) const
    {
        Point pt;
        pt.x = this->x + arg.x;
        pt.y = this->y + arg.y;

        return pt;
    }
};

void main()
{
    Point p1(2, 3), p2(5, 5);
    Point p3;

    p3 = p1 + p2; // 컴파일러가 p1.operator+(p2)로 해석해서 호출함!
    p3.Print();
    p3 = p1.operator+(p2); // 직접 호출함!
    p3.Print();
}
```

# 1. 연산자 오버로딩

## 단항 연산자 오버로딩

- 오버로딩이 가능한 단항 연산자는 **!, &, ~, \*, +, -, ++, --** 형 변환 연산자
- **+, -**는 부호연산자.

**++, --연산자 오버로딩** – 전위, 후위연산자가 있다.

```
class Point
{
    int x;
    int y;

public:
    Point(int _x = 0, int _y = 0) :x(_x), y(_y) { }
    void Print() const { cout << x << ',' << y << endl; }

    const Point operator++ () // 전위 ++
    {
        ++x;
        ++y;
        return *this;
    }

    const Point operator++(int) // 후위 ++
    {
        x++;
        y++;
        return *this;
    }
};
```

```
void main()
{
    Point p1(2, 3), p2(2, 3);
    Point result;

    result = ++p1; // p1.operator++(); 와 같다.
    p1.Print();
    result.Print();

    result = p2++; // p2.operator++(0); 와 같다.
    p2.Print();
    result.Print();
}
```



# 1. 연산자 오버로딩

## 이항 연산자 오버로딩

- 오버로딩 가능한 이항 연산자로는 `+`, `-`, `*`, `/`, `==`, `<`, `<=` 등이 있다.
- `+(덧셈)`, `-(뺄셈)`

### `==` 연산자 오버로딩

```
class Point
{
    int x;
    int y;
public:
    Point(int _x = 0 , int _y = 0 ):x(_x),y(_y) { }
    void Print( ) const { cout << x << ',' << y << endl; }
    bool operator== (const Point& arg) const
    {
        return x == arg.x && y == arg.y ? true : false;
    }
};

void main( )
{
    Point p1(2,3), p2(5,5), p3(2,3);

    if( p1 == p2 ) // p1.operator==(p2) 와 같다.
        cout << "p1 == p2" << endl;

    if( p1 == p3 ) // p1.operator==(p3) 와 같다.
        cout << "p1 == p3" << endl;
}
```

# 1. 연산자 오버로딩

## != 연산자 오버로딩

```
class Point
{
    int x;
    int y;
public:
    Point(int _x =0 , int _y =0 ):x(_x),y(_y) { }
    void Print( ) const { cout << x <<',' << y << endl; }
    bool operator== (const Point& arg) const
    {
        return x==arg.x && y==arg.y ? true : false;
    }
    bool operator!= (const Point& arg) const
    {
        return !(*this == arg);
    }
};

void main( )
{
    Point p1(2,3), p2(5,5), p3(2,3);

    if( p1 != p2 ) // p1.operator!= (p2) 와 같다.
        cout << "p1 != p2" << endl;
    if( p1 != p3 ) // p1.operator!= (p3) 와 같다.
        cout << "p1 != p3" << endl;
}
```

# 1. 연산자 오버로딩

- 멤버 함수를 이용한 오버로딩, 전역 함수를 이용한 오버로딩 두 가지가 있다.
- 멤버 함수를 이용할 수 없는 경우 전역 함수를 이용한 오버로딩을 한다.

## 멤버 함수를 이용한 연산자 오버로딩

**class Point**

```
{  
    int x;  
    int y;  
  
    public:  
  
    Point(int _x = 0, int _y = 0) : x(_x), y(_y) {}  
    void Print()const { cout << x << ', ' << y << endl; }  
    int GetX()const { return x; }  
    int GetY()const { return y; }  
  
    const Point operator-(const Point& arg) const  
    {  
        return Point(this->x - arg.x, this->y - arg.y);  
    }  
};
```

**void main()**

```
{  
    Point p1(2, 3), p2(5, 5);  
    Point p3;  
  
    p3 = p1 - p2; → p1.operator - (p2);  
  
    p3.Print();  
}
```

# 1. 연산자 오버로딩

## 전역 함수를 이용한 연산자 오버로딩

- 전역함수의 인자로 전달되어 연산이 이루어 진다.
- **Getter**를 이용하거나 **friend** 함수를 이용하여 전달한다.

```
class Point
```

```
{
```

```
    int x;
```

```
    int y;
```

```
public:
```

```
    Point(int _x =0 , int _y = 0 ):x(_x),y(_y) { }
```

```
    void Print( )const { cout << x <<',' << y << endl; }
```

```
    int GetX()const { return x; } // x의 getter
```

```
    int GetY()const { return y; } // y의 getter
```

```
};
```

```
const Point operator- (const Point& argL, const Point& argR)
```

```
{
```

```
    return Point( argL.GetX()-argR.GetX(), argL.GetY()-argR.GetY());
```

```
}
```

```
void main( )
```

```
{
```

```
    Point p1(2,3), p2(5,5);
```

```
    Point p3;
```

```
    p3 = p1 - p2; → operator- (p1, p2);
```

```
    p3.Print();
```

```
}
```

friend const Point operator- (const Point& argL,  
const Point& argR);

// friend 함수를 이용한 방법

```
const Point operator- (const Point& argL, const Point& argR)
```

```
{
```

```
    return Point( argL.x-argR.x, argL.y-argR.y );
```

```
}
```

# 1. 연산자 오버로딩

**함수 호출 연산자 오버로딩** - ()연산자

```
class FuncObject
{
public:
    void operator()( int arg ) const
    {
        cout <<"정수 : " << arg << endl;
    }
};

void Print1( int arg )
{
    cout <<"정수 : " << arg << endl;
}

void main( )
{
    void (*Print2)(int) = Print1;
    FuncObject Print3;

    Print1(10); // 첫째, 함수 호출
    Print2(10); // 둘째, 함수 호출
    Print3(10); // 첫째, 함수 호출 Print3.operator(10)을 호출
}
```

# 1. 연산자 오버로딩

```
class FuncObject
{
public:
    void operator()(int arg) const
    {
        cout << "정수 : " << arg << endl;
    }

    void operator()(int arg1, int arg2) const
    {
        cout << "정수 : " << arg1 << ',';
        cout << arg2 << endl;
    }

    void operator()(int arg1, int arg2, int arg3) const
    {
        cout << "정수 : " << arg1 << ',' << arg2 << ',';
        cout << arg3 << endl;
    }
};
```

```
void main()
{
    FuncObject print;
    print(10); //객체 생성 후 호출(암시적)
    print(10, 20);
    print(10, 20, 30);
    cout << endl;

    print.operator()(10); //객체 생성 후 호출(명시적)
    print.operator()(10, 20);
    print.operator()(10, 20, 30);
    cout << endl;

    FuncObject()(10); //임시 객체로 호출(암시적)
    FuncObject()(10, 20);
    FuncObject()(10, 20, 30);
    cout << endl;

    FuncObject().operator()(10); //객체 생성 후 호출(명시적)
    FuncObject().operator()(10, 20);
    FuncObject().operator()(10, 20, 30);
}
```

# 1. 연산자 오버로딩

## 배열 인덱스 연산자 오버로딩 - [] 연산자

```
class Array
{
    int *arr;
    int size;
    int capacity;

public:
    Array(int cap = 100) : arr(0), size(0), capacity(cap)
    {
        arr = new int[capacity];
    }
    ~Array()
    {
        delete[] arr;
    }
    void Add(int data)
    {
        if (size < capacity)
            arr[size++] = data;
    }
    int Size() const
    {
        return size;
    }
    int operator[](int idx) const
    {
        return arr[idx];
    }
};
```

```
void main()
{
    Array ar(10);

    ar.Add(10);
    ar.Add(20);
    ar.Add(30);

    for (int i = 0; i < ar.Size(); i++)
        cout << ar[i] << endl; // ar.operator[](i) 와 같다.
}
```



# 1. 연산자 오버로딩

```
class Array
{
    int *arr;
    int size;
    int capacity;
    // 복사 함수 생략(복사 생성자,복사 대입 연산자)
public:
    Array(int cap = 100) :arr(0), size(0), capacity(cap)
    {
        arr = new int[capacity];
    }
    ~Array()
    {
        delete[] arr;
    }
    void Add(int data)
    {
        if (size < capacity)
            arr[size++] = data;
    }
    int Size() const
    {
        return size;
    }
    int operator[](int idx) const // 읽기, 쓰기
    {
        return arr[idx];
    }
    int& operator[](int idx) // 읽기
    {
        return arr[idx];
    }
};
```

```
void main()
{
```

```
    Array ar(10);
    ar.Add(10);
    ar.Add(20);
    ar.Add(30);
```

```
    cout << ar[0] << endl; // ar.operator[](int) 를 호출한다.
    cout << endl;
```

```
    const Array& ar2 = ar; // ar을 참조
    cout << ar2[0] << endl; // ar.operator[](int) const 를 호출한다.
    cout << endl;
```

```
    ar[0] = 100; // ar.operator[](int) 를 호출합니다.
    //ar[1] = 100; 에러! 상수 객체(값)를 리턴하므로 대입할 수 없다.
```

```
}
```



# 1. 연산자 오버로딩

## 메모리 접근, 클래스 멤버 접근 연산자 오버로딩

- \*, -> 연산자는 스마트 포인터나 반복자 등의 특수한 객체에 사용된다.

```
class Point
```

```
{
```

```
    int x;
```

```
    int y;
```

```
public:
```

```
    Point(int _x = 0, int _y = 0) : x(_x), y(_y) { }
```

```
    void Print() const { cout << x << ',' << y << endl; }
```

```
};
```

```
class PointPtr
```

```
{
```

```
    Point *ptr;
```

```
public:
```

```
    PointPtr(Point *p) : ptr(p) { }
```

```
    ~PointPtr()
```

```
{
```

```
        delete ptr;
```

```
}
```

```
    Point* operator->() const
```

```
{
```

```
        return ptr;
```

```
}
```

```
    Point& operator*() const
```

```
{
```

```
        return *ptr;
```

```
}
```

```
};
```

```
void main()
```

```
{
```

```
    Point* p1 = new Point(2, 3); //일반 포인터
```

```
    PointPtr p2 = new Point(5, 5); //스마트 포인터
```

```
    p1->Print(); //p1->Print() 호출
```

```
    p2->Print(); //p2.operator->()->Print() 호출
```

```
    cout << endl;
```

```
    (*p1).Print(); //(*p1).Print() 호출
```

```
    (*p2).Print(); //p1.operator*().Print() 호출
```

```
    delete p1;
```

```
    // p2의 소멸자에서 Point 동적 객체를 자동 메모리 제거합니다.
```

```
}
```

# 1. 연산자 오버로딩

## 타입변환 연산자 오버로딩

### 사용자 정의로 사용할 수 있는 타입변환 방법

- 생성자를 이용한 타입 변환
- 타입 변환 연산자 오버로딩을 이용한 타입 변환

### 생성자를 이용한 타입변환

- 암시적으로 형변환이 이루어진다.
- int형 인자를 받는 변환 생성자가 char형이나 float을 받았을 때 Error는 발생하지 않지만 결과는 버그다.
- explicit키워드를 지정해서 명시적인 변환이 이루어지게 만들자.

```
class A
{
};
class B
{
public:
    B() { cout << "B() 생성자" << endl; }
    B(A& _a) { cout << "B(A _a) 생성자" << endl; }
    B(int n) { cout << "B(int n) 생성자" << endl; }
    B(double d) { cout << "B(double d) 생성자" << endl; }
};
```

### void main()

```
{
    A a;
    int n = 10;
    double d = 5.5;

    B b; // B() 생성자 호출
    b = a; // b = B(a) 암시적 생성자 호출 후 대입
    b = n; // b = B(n) 암시적 생성자 호출 후 대입
    b = d; // b = B(d) 암시적 생성자 호출 후 대입
}
```

# 1. 연산자 오버로딩

```
class Point
{
    int x;
    int y;
public:
    Point(int _x =0 , int _y =0 ):x(_x),y(_y) { }
    void Print( ) const { cout << x <<',' << y << endl; }
};
```

```
void main( )
{
    Point pt;
    pt.Print();

    pt = 10; // Point(10,0) 암시적 생성자 호출
    pt.Print();
}
```

```
class Point
{
    int x;
    int y;
public:
    explicit Point(int _x = 0, int _y = 0) :x(_x), y(_y) { }
    void Print() const { cout << x << ',' << y << endl; }
};
```

```
void main()
{
    Point pt;
    pt.Print();

    //pt = 10; // 에러! 암시적 생성자 호출이 불가능!
    pt = Point(10); // 이렇게 명시적 생성자 호출만 가능!
    pt.Print();
}
```

# 1. 연산자 오버로딩

## 타입 변환 연산자 오버로딩을 이용한 타입 변환

```
class A
{
};
class B
{
public:
    operator A()
    {
        cout << "operator A() 호출" << endl;
        return A();
    }
    operator int()
    {
        cout << "operator int() 호출" << endl;
        return 10;
    }
    operator double()
    {
        cout << "operator double() 호출" << endl;
        return 5.5;
    }
};
```

```
void main()
{
    A a;
    int n;
    double d;

    B b;
    a = b; //b.operator A() 암시적 호출
    n = b; //b.operator int() 암시적 호출
    d = b; //b.operator double() 암시적 호출

    cout << endl;
    a = b.operator A(); // 명시적 호출
    n = b.operator int(); // 명시적 호출
    d = b.operator double(); // 명시적 호출
}
```

# 1. 연산자 오버로딩

```
class Point
{
    int x;
    int y;
public:
    explicit Point(int _x = 0, int _y = 0) :x(_x), y(_y) { }
    void Print() const { cout << x << ',' << y << endl; }

    operator int() const
    {
        return x;
    }
};

void main()
{
    int n = 10;

    Point pt(2, 3);
    n = pt; // pt.operator int() 암시적 호출
    cout << n << endl;
}
```

# 1. 연산자 오버로딩

## 학습과제

- 단항 연산자(!, &, ~, \*, +, -, --)를 연산자 오버로딩 기능을 구현하자.
- 이항 연산자(+, -, \*, /, <, <=)를 연산자 오버로딩 기능을 구현하자.

# 함수 포인터

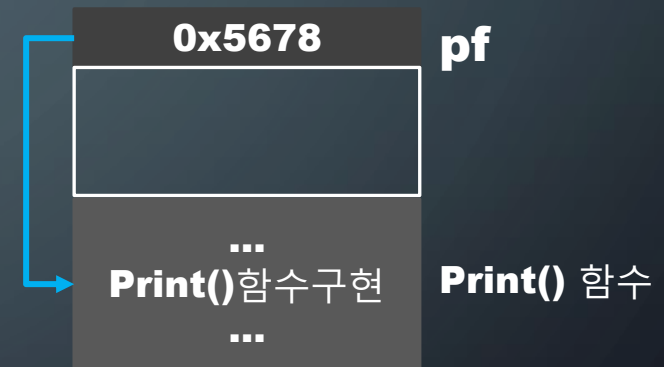
## 2. 함수 포인터

- 함수의 시작 주소를 지정하는 포인터
- Int func(int a, int b)**를 시그니처로 갖는 함수의 함수 포인터는 **int (\*pf)(int, int)**와 같이 선언

```
void Print(int n)
{
    cout << "정수: " << n << endl;
}
void main()
{
    // void Print(int n)의 함수 포인터 선언
    void (*pf)(int );
    // 함수의 이름은 함수의 시작 주소
    pf=Print;

    Print( 10 ); //1. 함수 호출
    pf( 10 ); //2. 포인터를 이용한 함수 호출, 첫 번째 방법
    (*pf)( 10 ); //3. 포인터를 이용한 함수 호출, 두 번째 방법

    cout << endl;
    cout << Print << endl;
    cout << pf << endl;
    cout << *pf << endl;
}
```





## 2. 함수 포인터

### 함수 포인터의 종류

- 정적 함수 호출(정적 함수)
- 객체로 멤버 함수 호출(멤버 함수)
- 객체의 주소로 멤버 함수 호출(멤버 함수)

C++에서 함수의 호출

```
void Print( )
```

```
{
```

```
    cout <<"정적 함수 Print()"<< endl;
```

```
}
```

```
class Point
```

```
{
```

```
public:
```

```
    void Print( )
```

```
    {
```

```
        cout <<"멤버 함수 Print()"<< endl;
```

```
    }
```

```
};
```

```
void main()
```

```
{
```

```
    Point pt;
```

```
    Point * p = &pt;
```

```
    Print(); // 첫째, 정적 함수 호출
```

```
    pt.Print(); // 둘째, 객체로 멤버함수 호출
```

```
    p->Print(); // 셋째, 주소로 멤버함수 호출
```

```
}
```

## 2. 함수 포인터

### 정적 함수 호출

- 정적 함수인 전역 함수, **namespace** 내의 전역 함수, **static** 멤버 함수는 모두 함수 호출 규약이 같아서 함수 포인터가 같다.

```
void Print(int n)
{
    cout << "전역 함수: " << n << endl;
}

namespace A
{
    void Print(int n)
    {
        cout << "namespace A 전역 함수: “;
        cout << n << 두이;
    }
}

Class Point
{
Public:
    Static void Print(int n)
    {
        cout << “Point 클래스의 정적 멤버 함수: “;
        cout << n << endl;
    }
};
```

```
void main()
{
    void(*pf)(int);

    // 1. namespace 없는 전역 함수 호출
    Print(10);
    // 2, namespace A의 전역 함수 호출
    A::Print(10);
    // 3, Point 클래스의 정적 멤버 함수 호출
    Point::Print(10);
    cout << endl;

    pf = Print;
    // 1. 함수 포인터로 namespace 없는 전역 함수 호출
    pf(10);
    pf = A::Print;
    // 2. 함수 포인터로 namespace A의 전역 함수 호출
    pf(10);
    pf = Point::Print;
    // 3. 함수 포인터로 Point 클래스의 정적 멤버 함수 호출
    pf(10);
}
```

## 2. 함수 포인터

### 객체와 주소로 멤버 함수 호출

```
void main()
{
    Point pt(2, 3);
    Point *p = &pt;

    // 멤버 함수 포인터 선언
    void (Point::*pf1)() const;
    pf1 = &Point::Print;

    // 멤버 함수 포인터 선언
    void (Point::*pf2)(int);
    pf2 = &Point::PrintInt;

    pt.Print();
    pt.PrintInt(10);
    cout << endl;

    (pt.*pf1)(); // 객체로 멤버 함수 포인터를 이용한 호출
    (pt.*pf2)(10); // 객체로 멤버 함수 포인터를 이용한 호출
    cout << endl;

    (p->*pf1)(); // 주소로 멤버 함수 포인터를 이용한 호출
    (p->*pf2)(10); // 주소로 멤버 함수 포인터를 이용한 호출
}
```

```
class Point
{
    int x;
    int y;

public:
    explicit Point(int _x = 0, int _y = 0) : x(_x), y(_y) { }

    void Print() const
    {
        cout << x << ',' << y << endl;
    }
    void PrintInt(int n)
    {
        cout << "테스트 정수 : " << n << endl;
    }
};
```

# 함수 객체

### 3. 함수 객체

- 함수자(Functor) 라고 불린다.
- 함수처럼 동작하는 객체를 말한다.
- 매개변수 리스트를 갖는 함수 객체도 만들 수 있다.
- 함수처럼 동작하려면 ()연산자를 정의해야 하기때문에 ()연산자를 오버로딩 해야 한다.

```
void Print()
{
    cout << "전역 함수!" << endl;
}

class Functor
{
public:
    void operator( )()
    {
        cout << "함수 객체!" << endl;
    }
};

void main()
{
    Functor functor;

    Print(); // 전역 함수 호출
    functor(); // 멤버 함수 호출 functor.operator()( )와 같다
}
```

### 3. 함수 객체

```
void Print(int a, int b)
{
    cout << "전역 함수: " << a << ',' << b << endl;
}

class Functor
{
public:
    void operator( )(int a, int b)
    {
        cout << "함수 객체: " << a << ',' << b << endl;
    }
};

void main()
{
    Functor functor;

    Print(10, 20); // 전역 함수 호출
    functor(10, 20); // 멤버 함수 호출 functor.operator()(10, 20)와 같다;
}
```

### 3. 함수 객체

#### 함수 객체의 장점

- 일반 함수보다 속도가 빠르다.
- 함수 객체의 서명이 같더라도 객체 타입이 다르면 서로 전혀 다른 타입으로 인식한다.
- 함수 객체는 인라인 될 수 있고, 컴파일러가 쉽게 최적화 할 수 있다.

```
class Adder
```

```
{
```

```
    int total;
```

```
public:
```

```
    explicit Adder(int n=0):total(n) { }
```

```
    int operator( )(int n) // 클래스 내부에서 암묵적으로 인라인 함수가 된다.
```

```
    {
```

```
        return total += n;
```

```
    }
```

```
};
```

```
void main( )
```

```
{
```

```
    Adder add(0); //초기값 0
```

```
    cout << add(10) << endl; //10을 누적=>10
```

```
    cout << add(20) << endl; //20을 누적=>30
```

```
    cout << add(30) << endl; //30을 누적=>60
```

```
}
```

### 3. 함수 객체

```
#include <algorithm> // fo_each() 알고리즘(서버)을 사용하기 위한 헤더
```

```
struct Functor1
```

```
{
```

```
    // 공백을 이용하여 원소를 출력
```

```
    void operator()(int n)
```

```
    {
```

```
        cout << n << ' ';
```

```
    }
```

```
};
```

```
struct Functor2
```

```
{
```

```
    // 각 원소를 제곱하여 출력
```

```
    void operator()(int n)
```

```
    {
```

```
        cout << n * n << " ";
```

```
    }
```

```
};
```

```
struct Functor3
```

```
{
```

```
    // 문자열과 endl을 이용하여 원소를 출력
```

```
    void operator()(int n)
```

```
    {
```

```
        cout << "정수 : " << n << endl;
```

```
    }
```

```
};
```

```
void main()
```

```
{
```

```
    int arr[5] = { 10,20,30,40,50 };
```

```
    //임시 함수자 객체(Functor1())를 만들어 함수로 전달
```

```
    for_each(arr, arr + 5, Functor1());
```

```
    cout << endl << endl;
```

```
    //임시 함수자 객체(Functor2())를 만들어 함수로 전달
```

```
    for_each(arr, arr + 5, Functor2());
```

```
    cout << endl << endl;
```

```
    //임시 함수자 객체(Functor3())를 만들어 함수로 전달
```

```
    for_each(arr, arr + 5, Functor3());
```

```
}
```



### 3. 함수 객체

#### 함수 객체 구현

두 정수를 입력 받아 '<' 연산을 수행하는 함수와 함수객체를 구현

```
bool Pred_less(int a, int b)
{
    return a < b;
}
```

```
class Less
{
public:
    bool operator()(int a, int b)
    {
        return a < b;
    }
};
```

```
void main()
{
    Less less;

    cout << Pred_less(10, 20) << endl;
    cout << Pred_less(20, 10) << endl;
    cout << endl;

    cout << less(10, 20) << endl; // less 객체로 암묵적 함수 호출
    cout << less(20, 10) << endl; // less 객체로 암묵적 함수 호출
    cout << Less()(10, 20) << endl; // 임시객체로 암묵적 함수 호출
    cout << Less()(20, 10) << endl; // 임시객체로 암묵적 함수 호출
    cout << endl;

    cout << less.operator()(10, 20) << endl; // 명시적 호출
    cout << Less().operator()(10, 20) << endl; // 명시적 호출
}
```

### 3. 함수 객체

```
#include <functional> //STL less<>, greater<>
```

```
class Less
```

```
{
```

```
public:
```

```
    bool operator()(int a, int b)
```

```
    {
```

```
        return a < b;
```

```
    }
```

```
};
```

```
class Greater
```

```
{
```

```
public:
```

```
    bool operator()(int a, int b)
```

```
    {
```

```
        return a > b;
```

```
    }
```

```
};
```

```
void main()
```

```
{
```

```
    //사용자 Less, Greater 사용
```

```
    cout << Less()(10, 20) << endl;
```

```
    cout << Less()(20, 10) << endl;
```

```
    cout << Greater()(10, 20) << endl;
```

```
    cout << Greater()(20, 10) << endl;
```

```
    cout << endl;
```

```
    //STL의 less, greater 사용
```

```
    cout << less<int>()(10, 20) << endl;
```

```
    cout << less<int>()(20, 10) << endl;
```

```
    cout << greater<int>()(10, 20) << endl;
```

```
    cout << greater<int>()(20, 10) << endl;
```

```
}
```

### 3. 함수 객체

```
#include <functional> //plus<>, minus<>
```

```
class Plus
```

```
{
```

```
public:
```

```
    int operator()(int a, int b)
```

```
    {
```

```
        return a + b;
```

```
    }
```

```
};
```

```
class Minus
```

```
{
```

```
public:
```

```
    int operator()(int a, int b)
```

```
    {
```

```
        return a - b;
```

```
    }
```

```
};
```

```
void main()
```

```
{
```

```
    //사용자 Plus, Minus 사용
```

```
    cout << Plus()(10, 20) << endl;
```

```
    cout << Plus()(20, 10) << endl;
```

```
    cout << Minus()(10, 20) << endl;
```

```
    cout << Minus()(20, 10) << endl;
```

```
    cout << endl;
```

```
    //STL의 plus, minus 사용
```

```
    cout << plus<int>()(10, 20) << endl;
```

```
    cout << plus<int>()(20, 10) << endl;
```

```
    cout << minus<int>()(10, 20) << endl;
```

```
    cout << minus<int>()(20, 10) << endl;
```

```
}
```

템플릿

## 4. 템플릿

- 여러 타입의 함수나 클래스를 쉽게 구현 가능하게 한다.
- 프로그램의 속도에 영향을 주지 않지만, 컴파일하는 도중에 클래스나 함수를 생성해야 하므로 컴파일이 느려 지는 단점은 있다.
- 함수 템플릿과 클래스 템플릿이 있다.

### 함수 템플릿

```
void Print(int a, int b)
```

```
{  
    cout << a << ", " << b << endl;  
}
```

```
void Print(double a, double b)
```

```
{  
    cout << a << ", " << b << endl;  
}
```

```
void Print(const char* a, const char* b)
```

```
{  
    cout << a << ", " << b << endl;  
}
```

```
void main( )
```

```
{  
    Print(10, 20); // 정수 출력  
    Print(0.123, 1.123); // 실수 출력  
    Print("ABC", "abcde"); // 문자열 출력  
}
```



- 함수앞에 **template<T>**키워드를 붙이면 된다.
- 매개 변수도 **<T>**여야 한다.
- 명시적, 암시적 호출 둘 다 알맞은 형태의 함수로 변형 시키기 때문에 구분할 필요 없다.

```
template <typename T>
```

```
void Print(T a, T b)
```

```
{  
    cout << a << ", " << b << endl;  
}
```

```
void main()
```

```
{  
    //암시적 호출  
    Print<int>(10, 20); // 정수 출력  
    Print<double>(0.123, 1.123); // 실수 출력  
    Print<const char*>("ABC", "abcde"); // 문자열 출력  
  
    //명시적 호출  
    Print<int>(10, 20); // 정수 출력  
    Print<double>(0.123, 1.123); // 실수 출력  
    Print<const char*>("ABC", "abcde"); // 문자열 출력  
}
```

## 4. 템플릿

둘 이상 여러 매개변수를 가지는 경우.

```
template <typename T1, typename T2>
```

```
void Print(T1 a, T2 b)
```

```
{  
    cout << a << ", " << b << endl;  
}
```

```
void main()
```

```
{  
    Print(10, 1.5); // 정수, 실수 출력  
    Print("Hello!", 100); // 문자열, 정수 출력  
    Print(1.5, "Hello!"); // 실수, 문자열 출력  
}
```

템플릿 함수 정의의 연산이 가능한 객체, 즉 인터페이스를 지원하는 객체라면 모두 올 수 있다, 다시 말해 **T temp = a** 문장에서 복사 생성자를 호출하므로 복사 생성자를 지원해야 하며 **a = b**나 **b = temp** 문장에서 대입 연산자를 호출하므로 대입 연산자를 지원해야 한다.

```
template <typename T>
```

```
void Swap(T &a, T &b)
```

```
{  
    T temp = a;  
    a = b;  
    b = temp;  
}
```

```
void main()
```

```
{  
    int n1 = 10, n2 = 20;  
    double d1 = 1.1, d2 = 2.2;  
  
    cout << n1 << ", " << n2 << endl;  
    Swap(n1, n2);  
    cout << n1 << ", " << n2 << endl;  
    cout << endl;  
  
    cout << d1 << ", " << d2 << endl;  
    Swap(d1, d2);  
    cout << d1 << ", " << d2 << endl;  
  
}
```

## 4. 템플릿

배열을 템플릿으로 사용하는 경우 - 정수도 가능하다.

```
template <typename T, int size>
```

```
void PrintArray(T* arr)
```

```
{
```

```
    for(int i = 0 ; i < size ; ++i)
```

```
    {
```

```
        cout << "[" << i << "]: " << arr[i] << endl;
```

```
    }
```

```
    cout << endl;
```

```
}
```

```
void main( )
```

```
{
```

```
    int arr1[5] = { 10, 20, 30, 40, 50};
```

```
    PrintArray<int, 5>(arr1); // 명시적 호출
```

```
    double arr2[3] = { 1.1, 2.2, 3.3};
```

```
    PrintArray<double, 3>(arr2); // 명시적 호출
```

```
}
```

함수 인자가 **arr1**이라는 정보만을 제공하므로 **5**라는 템플릿 매개변수 인자를 컴파일러가 추론할 수 없기 때문에 명시적으로 호출해 컴파일러가 함수 템플릿 인스턴스를 생성하게 한다.



## 4. 템플릿

연산자 오버로딩의 지원이 불가능한 경우

```
class Point
{
    int x; int y;
public:
    explicit Point(int _x =0 , int _y =0 ):x(_x),y(_y) { }
    void Print( ) const { cout << x <<',' << y << endl; }
};
```

```
template <typename T>
void Print(T a)
```

```
{
    cout << a << endl;
}
```

```
void main( )
{
```

```
    int n = 10;
    double d = 2.5;
    Point pt(2,3);
```

```
    Print( n );
```

```
    Print( d );
```

```
    Print( pt ); //에러! cout << pt; 연산이 불가능하므로
```

```
}
```

템플릿의 매개변수 타입 객체인 **a**가 템플릿 함수 정의의 연산을 지원해야 하는데 **pt**는 **cout**의 **<<** 연산을 지원하지 않는다. 그래서 연산자 오버로딩 함수를 추가하거나 **특수화된 함수 템플릿을 지원**하는 방법이 있다.



## 4. 템플릿

```
class Point
{
    int x; int y;

public:
    explicit Point(int _x =0 , int _y =0 ):x(_x),y(_y) { }
    void Print( ) const { cout << x <<',' << y << endl; }
};

// 일반화 함수 템플릿
template <typename T>
void Print(T a)
{
    cout << a << endl;
}

// 특수화 함수 템플릿
template < >
// 명시적인 코드: void Print<Point>(Point a)
void Print(Point a)
{
    cout << "Print 특수화 버전: ";
    a.Print();
}

void main( )
{
    int n = 10;
    double d = 2.5;
    Point pt(2,3);

    Print( n ); // Print<int>( n ) 일반화 버전 호출
    Print( d ); // Print<double>( d ) 일반화 버전 호출
    Print( pt ); // Print<Point>( pt ) 특수화 버전 호출
}
```

# 4. 템플릿

## 클래스 템플릿

- 클래스를 정의하기 위한 메타 클래스 코드
- 템플릿 매개변수 인자를 통해 클래스에 사용될 타입을 결정할 수 있다.

```
class IntArray //정수 Array
{
    int *buf;
    int size;
    int capacity;

public:
    explicit IntArray(int cap = 100) :buf(0), size(0), capacity(cap)
    {
        buf = new int[capacity];
    }
    ~IntArray() { delete[] buf; }

    void Add(int data) { buf[size++] = data; }
    int operator[](int idx) const { return buf[idx]; }
    int GetSize() const { return size; }
};

class StringArray //문자열 Array
{
    string *buf;
    int size;
    int capacity;

public:
    explicit StringArray(int cap = 100) :buf(0), size(0), capacity(cap)
    {
        buf = new string[capacity];
    }
    ~StringArray() { delete[] buf; }

    void Add(string data) { buf[size++] = data; }
    string operator[](int idx) const { return buf[idx]; }
    int GetSize() const { return size; }
};
```

```
void main()
{
    IntArray iarr; // 정수 Array 객체
    iarr.Add(10);
    iarr.Add(20);
    iarr.Add(30);

    for (int i = 0; i < iarr.GetSize(); ++i)
        cout << iarr[i] << " ";

    cout << endl;

    StringArray sarr; // 문자열 Array 객체
    sarr.Add("abc");
    sarr.Add("ABC");
    sarr.Add("Hello!");

    for (int i = 0; i < sarr.GetSize(); ++i)
        cout << sarr[i] << " ";

    cout << endl;
}
```

## 4. 템플릿

```
template <typename T> // 클래스 템플릿 Array 정의
class Array
```

```
{
```

```
    T *buf;
```

```
    int size;
```

```
    int capacity;
```

```
public:
```

```
    explicit Array(int cap = 100) : buf(0), size(0), capacity(cap)
```

```
    {
```

```
        buf = new T[capacity];
```

```
    }
```

```
    ~Array() { delete[] buf; }
```

```
    void Add(T data)
```

```
    {
```

```
        buf[size++] = data;
```

```
    }
```

```
    T operator[](int idx) const
```

```
    {
```

```
        return buf[idx];
```

```
    }
```

```
    int GetSize() const
```

```
    {
```

```
        return size;
```

```
    }
```

```
};
```

```
void main()
```

```
{
```

```
    //정수(클라이언트가 T 타입 결정) Array 객체
```

```
    Array<int> iarr;
```

```
    iarr.Add(10);
```

```
    iarr.Add(20);
```

```
    iarr.Add(30);
```

```
    for (int i = 0; i < iarr.GetSize(); ++i)
```

```
        cout << iarr[i] << " ";
```

```
    cout << endl;
```

```
    //문자열(클라이언트가 T 타입 결정) Array 객체
```

```
    Array<string> sarr;
```

```
    sarr.Add("abc");
```

```
    sarr.Add("ABC");
```

```
    sarr.Add("Hello!");
```

```
    for (int i = 0; i < sarr.GetSize(); ++i)
```

```
        cout << sarr[i] << " ";
```

```
    cout << endl;
```

```
}
```

## 4. 템플릿

디폴트 매개 변수로 사용

```
template <typename T = int, int capT = 100> // int, 100 디폴트 매개 변수 값 지정
```

```
class Array
```

```
{
```

```
    T *buf;
```

```
    int size;
```

```
    int capacity;
```

```
public:
```

```
    explicit Array(int cap = capT) : buf(0), size(0), capacity(cap)
```

```
    {
```

```
        buf = new T[capacity];
```

```
    }
```

```
    ~Array() { delete[] buf; }
```

```
    void Add(T data)
```

```
    {
```

```
        buf[size++] = data;
```

```
    }
```

```
    T operator[](int idx) const
```

```
    {
```

```
        return buf[idx];
```

```
    }
```

```
    int GetSize() const
```

```
    {
```

```
        return size;
```

```
    }
```

```
};
```

```
void main()
```

```
{
```

```
    // 디폴트 매개 변수 값 int, 100 사용
```

```
    Array<> iarr;
```

```
    iarr.Add(10);
```

```
    iarr.Add(20);
```

```
    iarr.Add(30);
```

```
    for (int i = 0; i < iarr.GetSize(); ++i)
```

```
        cout << iarr[i] << " ";
```

```
    cout << endl;
```

```
    // 디폴트 매개 변수 값 사용하지 않음
```

```
    Array<string, 10> sarr;
```

```
    sarr.Add("abc");
```

```
    sarr.Add("ABC");
```

```
    sarr.Add("Hello!");
```

```
    for (int i = 0; i < sarr.GetSize(); ++i)
```

```
        cout << sarr[i] << " ";
```

```
    cout << endl;
```

```
}
```

## 4. 템플릿

템플릿, 템플릿 함수, 템플릿 함수 객체를 추가하는 예제..

```
//begin은 배열의 시작 주소, end는 배열의 끝 주소,  
//pf는 클라이언트 함수 포인터.  
void For_each(int *begin, int *end, void (*pf)(int ) )  
{  
    while( begin != end )  
    {  
        pf( *begin++ );  
    }  
}  
void PrintInt(int n)  
{  
    cout << n <<" ";  
}  
void main( )  
{  
    int arr[5] = {10, 20, 30, 40, 50};  
    For_each(arr, arr+5, PrintInt); // 정수 출력  
    cout << endl;  
}
```

## 4. 템플릿

**For\_each**를 함수 템플릿으로...

```
template <typename IterT, typename Func>  
void For_each(IterT begin, IterT end, Func pf )
```

```
{  
    while( begin != end )  
    {  
        pf( *begin++ );  
    }  
}
```

```
void PrintInt(int data)  
{  
    cout << data << " ";  
}
```

```
void PrintString(string data)  
{  
    cout << data << " ";  
}
```

```
void main( )  
{
```

```
    int arr[5] = {10, 20, 30, 40, 50};  
    For_each(arr, arr+5, PrintInt); // 정수 출력  
    cout << endl;
```

```
    string sarr[3] = {"abc", "ABCDE", "Hello!"};  
    For_each(sarr, sarr+3, PrintString); // 문자열 출력  
    cout << endl;
```

```
}
```

## 4. 템플릿

명시적 For\_each()호출

```
template <typename IterT, typename Func>  
void For_each(IterT begin, IterT end, Func pf )
```

```
{  
    while( begin != end )  
    {  
        pf( *begin++ );  
    }  
}
```

```
void PrintInt(int data)  
{  
    cout << data << " ";  
}
```

```
void PrintString(string data) {  
{  
    cout << data << " ";  
}
```

```
void main( )
```

```
    int arr[5] = {10, 20, 30, 40, 50};  
    For_each<int* , void (*)(int) >(arr, arr+5, PrintInt); // 정수 출력  
    cout << endl;
```

```
    string sarr[3] = {"abc", "ABCDE", "Hello!"};  
    For_each<string* , void (*)(string) >(sarr, sarr+3, PrintString); // 문자열 출력  
    cout << endl;
```

```
}
```



## 4. 템플릿

**Print()**함수도 템플릿 함수로 변경..

```
template <typename IterT, typename Func>
void For_each(IterT begin, IterT end, Func pf )
{
    while( begin != end )
    {
        pf( *begin++ );
    }
}

template <typename T>
void Print(T data)
{
    cout << data <<" ";
}

void main( )
{
    int arr[5] = {10, 20, 30, 40, 50};
    For_each(arr, arr+5, Print<int>); // 정수 출력
    cout << endl;

    string sarr[3] = {"abc", "ABCDE", "Hello!"};
    For_each(sarr, sarr+3, Print<string>); // 문자열 출력
    cout << endl;
}
```



## 4. 템플릿

함수객체를 사용해서 만들어 보자.

```
template <typename IterT, typename Func>  
void For_each(IterT begin, IterT end, Func pf )
```

```
{  
    while( begin != end )  
    {  
        pf( *begin++ );  
    }  
}
```

```
template <typename T>  
class PrintFunctor  
{
```

```
    string sep; // 출력 구분자 정보
```

```
public:
```

```
    explicit PrintFunctor(const string& s=" ") : sep(s) { }
```

```
    void operator()(T data) const
```

```
    {  
        cout << data << sep;  
    }
```

```
};
```

```
void main( )  
{
```

```
    int arr[5] = {10, 20, 30, 40, 50};  
    For_each(arr, arr+5, PrintFunctor<int>());  
    cout << endl;
```

```
    string sarr[3] = {"abc", "ABCDE", "Hello!"};  
    For_each(sarr, sarr+3, PrintFunctor<string>("*\n"));  
    cout << endl;
```

```
}
```

## 4. 템플릿

함수 객체의 반환 타입과 매개변수 타입을 결정하여 함수객체를 만드는 예제..

```
template <typename RetType, typename ArgType>  
class Functor  
{  
public :  
    RetType operator( ) (ArgType data)  
    {  
        cout << data << endl;  
        return RetType();  
    }  
};  
  
void main( )  
{  
    Functor< void, int > functor1;  
    functor1( 10 );  
    Functor< bool, string > functor2;  
    functor2( "Hello!" );  
}
```