C++ 프로그래밍 (C++ 스타일 알아보기)

<C++ 출력문>

```
#include <iostream>

int main() {
        std::cout << "Hello, World!" << std::endl;
}
```

```
#include <iostream>

using namespace std;

int main() {
        cout << "Hello, World!" << endl;
}
```

iostream: stdio.h와 같이 입/출력 함수가 담긴 헤더파일

std::cout – 공간::이름

c++프로그램에서는 변수 이름이나 함수 이름과 같은 수많은 이름(식별자)들이 사용되고, 이들 이름들은 이름 공간(name space)라고 하는 영역으로 분리되어 저장되어 있다.

위는 std라는 이름 공간에 있는 cout 이름의 객체를 불러서 출력 기능을 수행하는 것이다. 만약 입력 기능을 원한다면 cin객체를 불러오면 된다.

<< : 입/출력할 때 문자열을 분리하는 기능을 하는 연산자이다.

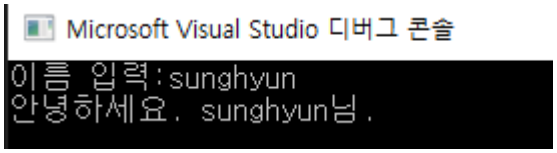
std:endl: endl은 std공간에 있는 '₩n' 키워드라고 생각하면 된다. 즉 end-of-line으로 출력을 끝낸다고 생각하면 된다.

➔ cout객체와 "<<"연산자을 이용해서 "Hello, World!" 와 "₩n"을 나눠 출력한다.

<C++ 문자열 처리>

```
#include <iostream>
#include <string>

using namespace std;

int main() {
        string name;
```

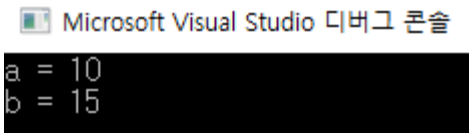
```
        name = "Hello";
        cout << "이름 입력:";
        cin >> name;

        string message = "안녕하세요. " + name + "님.";
        cout << message << endl;
}
```



string헤더 파일을 부르면 string변수를 사용할 수 있다. 이제 문자형 배열의 크기를 고려하지 않아도 되고 '+'연산자를 이용해서 손쉽게 문자열을 합칠 수도 있다.

<C++ 변수 선언 및 초기화>

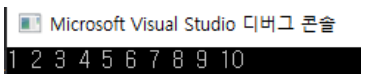
```
#include <iostream>

using namespace std;

int main() {
        int a(10); // int a = 10;
        int b(a+5); // int b = a+5;
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
}
```



c++에서 변수 선언과 초기화를 같이 할 때, 마치 함수를 호출하듯이 초기화를 할 수 있다.

<범위 기반 for문>

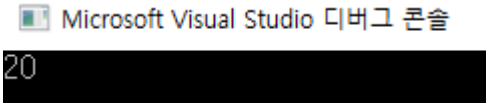
```
#include <iostream>

using namespace std;

int main() {
        int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };
        for (int n : arr) {
                cout << n << ' ';
        }
}
```



arr배열에 있는 모든 원소를 차례차례 n에 넣고 출력하는 소스이다.

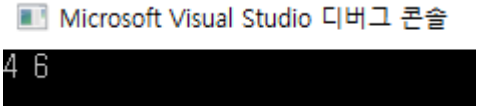
<참조자: 레퍼런스 변수>

```
#include <iostream>

using namespace std;

int main() {
	int var = 10;
	int &ref = var;
	ref = 20;
	cout << var << endl;
}
```



참조자는 &연산자를 붙여서 선언하고 '별명을 붙이는 역할'을 한다. 즉 위 소스에서는 var변수의 별명을 ref로 하겠다는 것이다. ref를 수정하면 ref를 별명으로 가지는 var의 값이 수정된다. var변수 입장에서는 ref가 자신의 별명이고, ref변수 입장에서는 var의 원본을 참조하는 참조자이다. 포인터와 다른 점은 참조자의 주소는 참조하는 변수의 주소와 같아서 공간을 추가하지 않는다는 점이다.

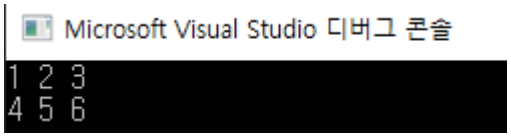
```
#include <iostream>

using namespace std;

void modify(int& x, int&y) {
	x = x * 2;
	y = y * 2;
}

int main() {
	int a = 2, b = 3;
	modify(a, b);

	cout << a << ' ' << b << endl;
	return 0;
}
```



매개변수 x와 y는 변수 a와 b의 원본을 참조하는 참조자이다. 이런 경우 a와 b의 값을 인수로 전달하면 값이 복사되는 것이 아니라 원본 변수가 전달된다. 이를 전문 용어로 (call-by-reference)라고 한다. 참고로 포인터를 사용하면, 변수의 주소를 인수로 전달해서 매개변수 포인터로 받으므로 (call-by-pointer)라고 불린다.

<범위 기반 for문과 참조자>

```
#include <iostream>
#include <string>

using namespace std;

int main() {
	int arr[2][3] = { {1,2,3},{4,5,6} };

	for (int(&ln)[3] : arr) {
		for (int col : ln) {
			cout << col << ' ';
		}
		cout << endl;
	}
}
```

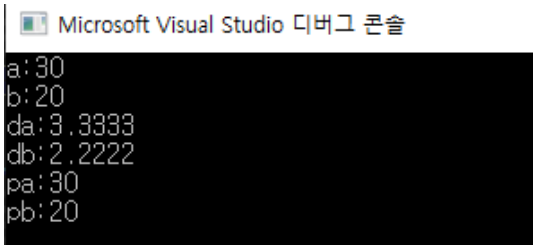


2차원 배열을 범위 기반 for문만으로 나타내려면 참조자가 필요하다. 먼저 위 2차원 배열은 (원소 3개를 가지는 1차원 배열)원소 두 개로 이루어져 있다. 범위 기반 for문은 원소를 차례차례 전달하기 때문에 첫 번째는 {1,2,3}배열, 두 번째는 {4,5,6}배열을 전달한다. 이 배열을 받는 변수는 ln이라는 참조자이다. int(&ln)[3]으로 선언한 이유는 ln참조자가 크기가 3인 배열을 참조하기 때문이다. 안 쪽 for문을 확인해보겠다. 다시 범위 기반 for문에 의해 ln이 참조하는 1차원 배열의 원소가 차례차례 col변수로 들어가고 출력이 된다.

<함수 오버로드 (다중 정의)>

```
#include <iostream>

void swap(int &a, int &b) {
	int tmp = a;
	a = b;
	b = tmp;
}

void swap(double &a, double &b) {
	double tmp = a;
	a = b;
	b = tmp;
}

void swap(int *a, int *b) {
	int *tmp = a;
	a = b;
	b = tmp;
```

```
}

int main() {
	int a(20), b(30);
	double da(2.2222), db(3.3333);
	int *pa = &a, *pb = &b;

	swap(a, b);
	swap(da, db);
	swap(pa, pb);

	std::cout << "a:" << a << std::endl;
	std::cout << "b:" << b << std::endl;

	std::cout << "da:" << da << std::endl;
	std::cout << "db:" << db << std::endl;

	std::cout << "pa:" << *pa << std::endl;
	std::cout << "pb:" << *pb << std::endl;
}
```



swap이라는 함수가 세 번이나 정의되었다. 이런 경우 함수를 호출할 때 컴파일러가 알아서 type, 매개변수 개수를 확인한 뒤 적합한 swap함수를 호출한다. 즉 사용자의 눈으로 구별이 가능한 동일 이름 함수를 컴파일러도 구분할 수 있다는 것이다.

<기본값을 지원하는 함수>

```
#include <iostream>

using namespace std;

int inventory[64] = { 0 };
int score = 0;
int number = 0;

void getItem(int itemId, int cnt=1, int sc=0, int a=7) {
	inventory[itemId]+=cnt;
	score += sc;
	number += a;
}

int main() {
	getItem(3);
	getItem(6, 5);
	getItem(11, 34, 7000);
```

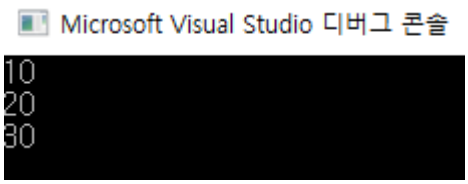
```
        getItem(12, 15, 18, 2000);

        cout << "score: "<< score << endl;
        cout << "number: " << number << endl;
        cout << "inventory: ";
        for (int i=0; i<64; i++)
                cout << inventory[i] << ' ';
        cout << endl;
}
```

함수를 호출할 때 인수가 덜 들어오면 매개변수의 값을 기본 값으로 대체하는 기능이 있다. 예를 들어 getItem(3)으로 호출하면 getItem함수의 매개변수는 (itemId=3, cnt=1, sc=0, a=7)이 된다. 기본값은 반드시 오른쪽부터 순서대로 정의되어야 한다는 특징이 있다.

C++ 프로그래밍 (클래스)

1. 네임 스페이스

```
#include <iostream>

using namespace std;

int n;
void set() {
	::n = 10; // 전역변수라는 것을 명시 (:: 빼도 됨)
}

namespace doodle {
	int n;
	void set() {
		doodle::n = 20; // double name space의 n (doodle:: 빼도 됨)
	}
}

namespace google {
	int n;
	void set() {
		google::n = 30; // google name space의 n (google::n 빼도 됨)
	}
}

int main() {
	::set();
	doodle::set();
	google::set();

	cout << ::n << endl;
	cout << doodle::n << endl;
	cout << google::n << endl;
}
```

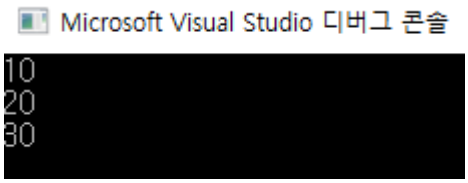


namespace는 변수의 이름, 함수의 이름이 모인 공간이다. 각각의 name space에서 정의된 변수와 함수의 이름이 같다고 하더라도 각각 다른 소속을 지닌다. 뿐 만 아니라 각 name space에서 다른 name space의 변수, 함수를 수정할 수도 있다.

따라서 필자는 이 namespace를 간단한 객체라고 생각한다. 이 객체 안에서 변수를 정의하고 함수를 정의하면 해당 객체의 멤버가 되는 개념이다. 참고로 다른 객체의 변수를 수정할 수도 있다.

2. 중첩 namespace

```
#include <iostream>

using namespace std;

int n;
void set() {
	n = 10;
}

namespace doodle {
	int n;
	void set() {
		n = 20;
	}

	namespace google {
		int n;
		void set() {
			n = 30;
		}
	}
}

int main() {
	::set();
	doodle::set();
	doodle::google::set();

	cout << ::n << endl;
	cout << doodle::n << endl;
	cout << doodle::google::n << endl;
}
```



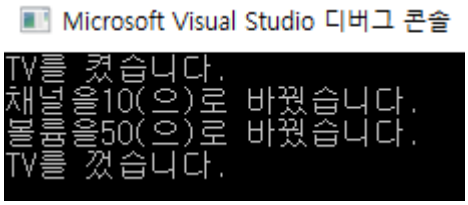
namespace안에 namespace를 만들 수도 있다.

double namespace안에 google namespace 안에 n변수를 정의했다면

double::google::name으로 변수를 불러올 수 있다.

3. 클래스 (접근 제어 지시자)

```
#include <iostream>

using namespace std;

struct TV {
private: // 외부에서 접근 불가
        bool poweron;
        int channel;
        int volume;

public: // 외부에서 접근 가능
        void on() {
                poweron = true;
                cout << "TV를 켰습니다." << endl;
        }
        void off() {
                poweron = false;
                cout << "TV를 껐습니다." << endl;
        }
        void setChannel(int cnl) {
                if (cnl >= 1 && cnl <= 999) {
                        channel = cnl;
                        cout << "채널을" << cnl<<"(으)로 바꿨습니다."<<endl;
                }
        }
        void setVolume(int vol) {
                if (vol >= 0 && vol <= 100) {
                        volume = vol;
                        cout << "볼륨을" << vol << "(으)로 바꿨습니다." << endl;
                }
        }
};

int main() {
        TV lg;
        lg.on();
        lg.setChannel(10);
        lg.setVolume(50);
        lg.off();
}
```

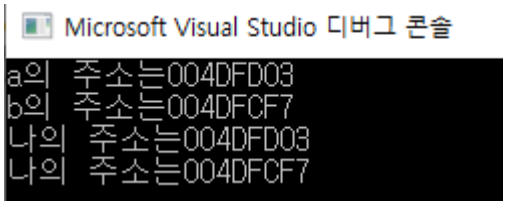


c++에서는 class 혹은 struct키워드를 이용해서 클래스를 만들 수 있다.

그리고 구조체 변수를 선언하듯이 객체를 만들 수 있다. 또한 클래스에서는 접근 제어 지시자(public, protected, private)도 사용이 가능하다.

4. 클래스의 This 포인터

```
#include <iostream>

using namespace std;

class MyClass {
public:
        void PrintThis() {
                // 자신이 소속되어 있는 객체의 주소: this
                cout << "나의 주소는" << this << endl;
        }
};

int main() {
        MyClass a, b;

        cout << "a의 주소는" << &a << endl;
        cout << "b의 주소는" << &b << endl;

        a.PrintThis();
        b.PrintThis();
}
```



This포인터는 보이지는 않지만 객체의 함수를 호출할 때 전달되는 포인터이다.

의미는 해당 객체의 주소이다.

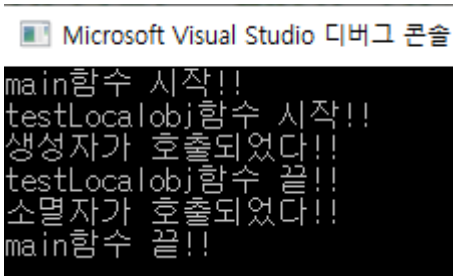
따라서 this는 그냥 객체 안에 항상 있는 포인터라고 생각해도 되고

'자신이 소속되어 있는 객체의 주소'라고 생각하면 된다.

추가로 this를 이용해서 객체 멤버를 수정할 때는 (this->멤버)를 이용하면 된다.

5. 생성자와 소멸자

```
#include <iostream>

using namespace std;

class MyClass {
public:
        // 생성자
        MyClass() {
                cout << "생성자가 호출되었다!!" << endl;
        }
        // 소멸자
        ~MyClass() {
                cout << "소멸자가 호출되었다!!" << endl;
        }
};

void testLocalobj() {
        cout << "testLocalobj함수 시작!!" << endl;
        MyClass localobj;
        cout << "testLocalobj함수 끝!!" << endl;
}

int main() {
        cout << "main함수 시작!!" << endl;
        testLocalobj();
        cout << "main함수 끝!!" << endl;
}
```



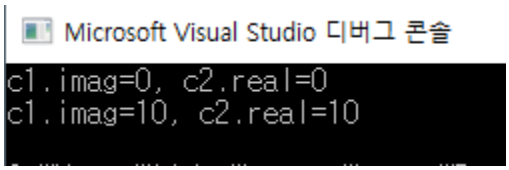
생성자: 객체가 생성될 때 호출되는 함수

소멸자: 객체가 소멸될 때 호출되는 함수

소멸될 때: 객체에 해당하는 혹은 객체를 가리키는 변수가 사라질 때

6. 생성자 오버로딩

```
#include <iostream>

using namespace std;

class Complex {
public:
        Complex() {
                real = 0;
                imag = 0;
        }
        Complex(double real_, double imag_) {
                real = real_;
                imag = imag_;
        }
        double GetReal() {
                return real;
        }
        void SetReal(double real_) {
                real = real_;
        }
        double GetImag() {
                return imag;
        }
        void setImag(double imag_) {
                imag = imag_;
        }
private:
        double real;
        double imag;
};


int main() {
        Complex c1;
        Complex c2(10,10);
        cout << "c1.imag="<<c1.GetImag()<< ", c2.real="<<c1.GetReal()<<endl;
        cout << "c1.imag=" << c2.GetImag()<< ", c2.real=" << c2.GetReal() << endl;
}
```



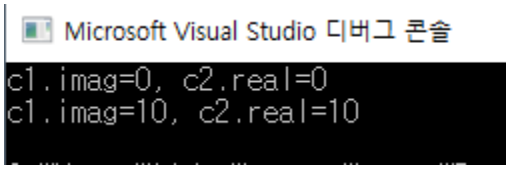
생성자도 함수이기 때문에 오버로딩이 가능하다.

보통 각 객체마다 초기화를 다르게 하고 싶을 때 위처럼 생성자를 오버로딩한다.

오버로딩된 생성자를 호출하려면 객체를 생성할 때 매개변수에 맞게 인수를 같이 주면 된다.

7. 초기화 목록

```
#include <iostream>

using namespace std;

class Complex {
public:
        Complex() :real(0), imag(0) {}
        Complex(double real, double imag) : real(real), imag(imag) {}

        double GetReal() {
                return real;
        }
        void SetReal(double real_) {
                real = real_;
        }
        double GetImag() {
                return imag;
        }
        void setImag(double imag_) {
                imag = imag_;
        }
private:
        double real;
        double imag;
};


int main() {
        Complex c1;
        Complex c2(10,10);
        cout << "c1.imag="<<c1.GetImag()<< ", c2.real="<<c1.GetReal()<<endl;
        cout << "c1.imag=" << c2.GetImag()<< ", c2.real=" << c2.GetReal() << endl;
}
```



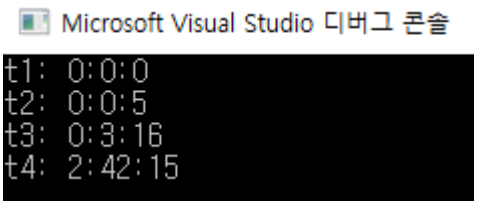
생성자를 간단하게 변경한 형태를 '초기화 목록'이라고 부른다.

초기화 목록에서는 객체의 멤버를 초기화하는 역할을 한다.

뿐 만 아니라 매개변수 이름을 멤버의 이름과 똑같이 해도 잘 작동하는 모습을 볼 수 있다.

8. 생성자 위임

```
#include <iostream>

using namespace std;

class Time {
public:
        Time() :h(0), m(0), s(0) {} // 1
        Time(int s_) :Time() { // 2
                s = s_;
        }
        Time(int m_, int s_) :Time(s_) { // 3
                m = m_;
        }
        Time(int h_,int m_,int s_) :Time(m_,s_) { // 4
                h = h_;
        }
        int h;
        int m;
        int s;
};

int main() {
        Time t1;
        Time t2(5);
        Time t3(3, 16);
        Time t4(2, 42, 15);
        cout << "t1: " << t1.h << ":" << t1.m << ":" << t1.s << endl;
        cout << "t2: " << t2.h << ":" << t2.m << ":" << t2.s << endl;
        cout << "t3: " << t3.h << ":" << t3.m << ":" << t3.s << endl;
        cout << "t4: " << t4.h << ":" << t4.m << ":" << t4.s << endl;
}
```



생성자 위임은 생성자를 호출할 때 다른 생성자도 같이 호출하는 것이다.

예를 들어 t4객체를 생성해보자.

4번 생성자가 호출되는데 m_,s_를 넘기면서 3번 생성자를 호출한다.

3번 생성자가 호출되는데 s_를 넘기면서 2번 생성자를 호출한다.

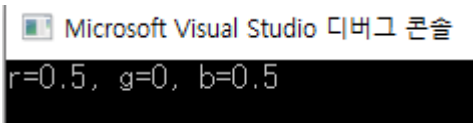
2번 생성자가 호출되는데 1번 생성자를 호출한다.

1번 생성자에서 모든 멤버를 0으로 초기화한다.

이후 2번->3번->4번 블록 내용인 s=s_ -> m=m_ -> h=h_가 실행된다.

9. 정적 메소드

```
#include <iostream>

using namespace std;

class Color {
public:
	Color() :r(0), g(0), b(0) {}
	Color(float r, float g, float b) :r(r), g(g), b(b) {}

	float GetR() { return r; }
	float GetG() { return g; }
	float GetB() { return b; }

	static Color MixColors(Color a, Color b) {
		return Color((a.r + b.r) / 2, (a.g + b.g) / 2, (a.b + b.b) / 2);
	}

private:
	float r;
	float g;
	float b;
};


int main() {
	Color blue(0, 0, 1);
	Color red(1, 0, 0);
	Color purple = Color::MixColors(blue, red);
	cout << "r=" << purple.GetR() << ", g=" << purple.GetG() << ", b=" << purple.GetB() <<
endl;
}
```



Color객체를 반환하는 MixColors메소드를 만들었다.

이 메소드의 역할은 들어온 두 색을 섞는 역할을 한다.

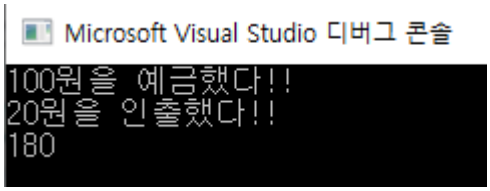
이 때 정적 메소드를 사용하면 main함수에서 클래스 이름으로 해당 메소드를 접근할 수 있다.

또한 이렇게 정적 메소드를 사용하면 private한 멤버들도 사용이 가능해져서 편리해진다.

tmi) 정적 변수를 사용하면 클래스 이름으로 해당 변수를 접근할 수 있다.

10. 매개변수,메소드의 상수화

```
#include <iostream>

using namespace std;

class Account {
public:
	Account() :money(0) {}
	Account(int money) :money(money) {}

	void Deposit(const int d) {
		// 상수 매개변수는 수정이 불가능함
		money += d;
		cout << d << "원을 예금했다!!" << endl;
	}
	void Draw(const int d) {
		if (money > d) {
			money -= d;
			cout << d << "원을 인출했다!!" << endl;
		}
	}
	int ViewMoney() const {
		// 상수 메소드에서는 멤버 변수 수정이 불가능함
		return money;
	}

private:
	int money;
};

int main() {
	Account doodle(100);
	doodle.Deposit(100);
	doodle.Draw(20);
	cout << doodle.ViewMoney() << endl;
}
```



위는 실수가 일어나면 치명적인 은행 프로그램이다.

만약 d값만큼 인출을 해야 되는데 d를 수정해서 수정한 만큼 인출하거나

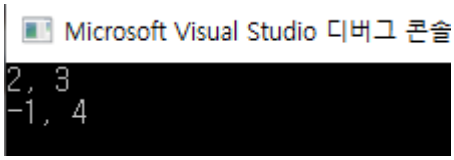
viewmoney에서 money멤버를 수정하면 문제가 생긴다.

즉 매개변수가 수정되면 안 될 때가 존재하고 메소드에서 멤버를 수정하면 안 될 때가 존재한다.

이럴 때 사용하는 것이 (상수 매개변수, 상수 메소드)이다.

11. 멤버 메소드의 선언, 정의 분리하기

```
#include <iostream>

using namespace std;

class Vector2 {
public:
        Vector2();
        Vector2(float x, float y);

        float GexX() const;
        float GetY() const;

private:
        float x;
        float y;
};

int main() {
        Vector2 a(2, 3);
        Vector2 b(-1, 4);

        cout << a.GexX() << ", " << a.GetY() << endl;
        cout << b.GexX() << ", " << b.GetY() << endl;
}

Vector2::Vector2() :x(0), y(0) {}
Vector2::Vector2(float x, float y) : x(x), y(y) {}
float Vector2::GexX() const { return x; }
float Vector2::GetY() const { return y; }
```



멤버 메소드는 (생성자,get메소드,set메소드)를 의미한다.

멤버 메소드는 위 코드처럼 선언과 정의를 다른 구간에 할 수도 있다.

12. 정적 메소드와 동적 메소드

```
#include <iostream>

using namespace std;

class Vector2 {
public:
	Vector2();
	Vector2(float x, float y);

	float GexX() const;
	float GetY() const;

	// 정적 메소드
	static Vector2 Sum(Vector2 a, Vector2 b) {
		return Vector2(a.x + b.x, a.y + b.y);
	}

	// 동적 메소드
	Vector2 Add(Vector2 rhs) {
		return Vector2(x + rhs.x, y + rhs.y);
	}

private:
	float x;
	float y;
};

int main() {
	Vector2 a(2, 3);
	Vector2 b(-1, 4);
	Vector2 c1 = Vector2::Sum(a, b); // 벡터a+벡터b
	Vector2 c2 = a.Add(b); // 벡터a+벡터b

	cout << c1.GexX() << ", " << c1.GetY() << endl;
	cout << c2.GexX() << ", " << c2.GetY() << endl;
}

Vector2::Vector2() :x(0), y(0) {}
Vector2::Vector2(float x, float y) : x(x), y(y) {}
float Vector2::GexX() const { return x; }
float Vector2::GetY() const { return y; }
```

Microsoft Visual Studio 디버그 콘솔

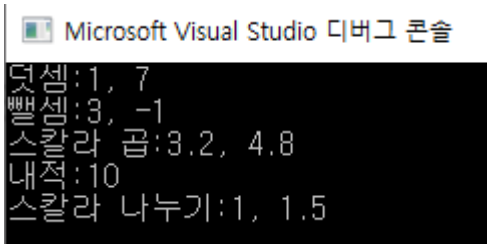
```
1, 7
1, 7
```

두 벡터 합을 구하는 과정을 두 방법으로 풀었다.

정적 메소드를 사용할 때는 '클래스 안의 함수'를 이용하므로 두 벡터 객체를 보내야 한다.
동적 메소드(일반적인 메소드)를 사용할 때는 '호출된 메소드를 멤버로 가지는 객체'를 이용하므로 한 벡터 객체만 보내도 된다.

13. 연산자 오버로딩

```
#include <iostream>

using namespace std;

class Vector2 {
public:
	Vector2();
	Vector2(float x, float y);

	float GexX() const;
	float GetY() const;

	// 'operator()' -> '()'로 메소드 호출이 가능
	Vector2 operator+(const Vector2 rhs) const;
	Vector2 operator-(const Vector2 rhs) const;
	Vector2 operator*(const float rhs) const;
	float operator*(const Vector2 rhs) const;
	Vector2 operator/(const float rhs) const;

private:
	float x;
	float y;
};

int main() {
	Vector2 a(2, 3);
	Vector2 b(-1, 4);
	Vector2 c1 = a + b; // 벡터a+벡터b
	Vector2 c2 = a - b; // 벡터a-벡터b
	Vector2 c3 = a * 1.6; // 벡터a*스칼라
	float c4 = a * b; // 벡터a와 b 내적
	Vector2 c5 = a/2; // 벡터a/스칼라

	cout << c1.GexX() << ", " << c1.GetY() << endl;
	cout << c2.GexX() << ", " << c2.GetY() << endl;
	cout << c3.GexX() << ", " << c3.GetY() << endl;
	cout << c4<< endl;
	cout << c5.GexX() << ", " << c5.GetY() << endl;
}

Vector2::Vector2() :x(0), y(0) {}
Vector2::Vector2(float x, float y) : x(x), y(y) {}
float Vector2::GexX() const { return x; }
float Vector2::GetY() const { return y; }
Vector2 Vector2::operator+(const Vector2 rhs) const { return Vector2(x + rhs.x, y + rhs.y); }
Vector2 Vector2::operator-(const Vector2 rhs) const { return Vector2(x - rhs.x, y - rhs.y); }
Vector2 Vector2::operator*(const float rhs) const { return Vector2(x * rhs, y * rhs); }
float Vector2::operator*(const Vector2 rhs) const { return (x * rhs.x + y * rhs.y); }
Vector2 Vector2::operator/(const float rhs) const { return Vector2(x*1/rhs, y*1/rhs); }
```

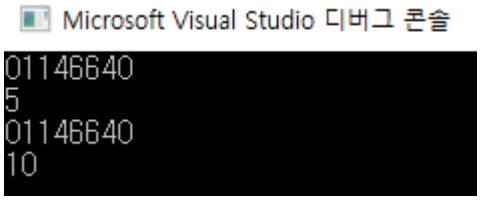


메소드를 연산자처럼 쓰려면 operator() 예약어를 사용해서 메소드 이름을 정하면 된다.

C++ 프로그래밍 (동적 할당과 객체 복사)

1. 동적 할당

```
#include <iostream>

using namespace std;

int main() {
	int *a = new int(5);

	cout << a << endl;
	cout << *a << endl;

	*a = 10;

	cout << a << endl;
	cout << *a << endl;

	delete a;
}
```



동적 할당: 컴파일 시간에 배열의 크기가 결정되는 정적 배열이 아닌 실행 시간에 크기를 변경할 수 있는 배열. 또한 정적 할당과 달리 변수 값만큼 할당이 가능하고 더 크기가 크게 할당이 가능하다. (메모리의 heap영역에 저장되므로)

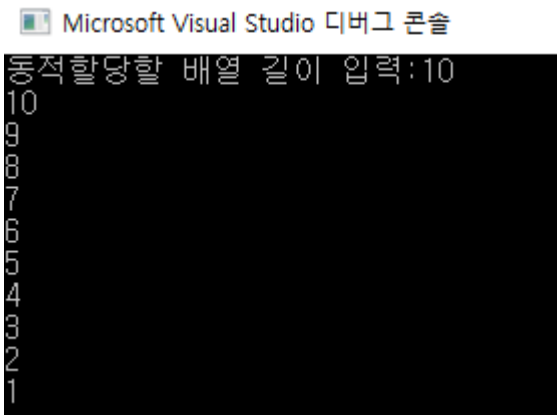
new연산자는 malloc과 비슷한 역할을 한다. 즉 동적 할당을 하기 위해 필요하다.

delete연산자는 free()와 같은 역할을 한다. 동적 할당

2. 배열에 대한 동적 할당

```
#include <iostream>

using namespace std;

int main() {
	int* arr;
	int len;

	cout << "동적할당할 배열 길이 입력:";
```

```
        cin >> len;

        arr = new int[len];

        for (int i = 0; i < len; i++)
                arr[i] = len - i;
        for (int i = 0; i < len; i++)
                cout << arr[i] << endl;

        delete[] arr;
}
```



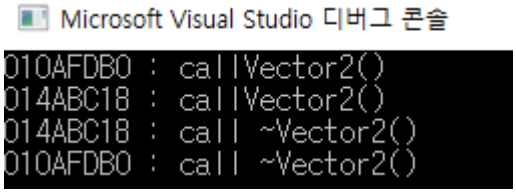
배열 만드는 것처럼 만드는데 new연산자를 앞에 붙여서 만든다.

배열을 동적 할당으로 만들면 반드시 delete[]로 해제해야 한다.

3. 객체에 대한 동적 할당

```
#include <iostream>

using namespace std;

class Vector2 {
public:
        Vector2() :x(0), y(0) {
                cout << this << " : callVector2()" << endl;
        }
        ~Vector2() {
                cout << this << " : call ~Vector2()" << endl;
        }
        float GetX() const { return x; }
        float GetY() const { return y; }
private:
        float x;
        float y;
};

int main() {
        Vector2 s1 = Vector2();
        Vector2* s2 = new Vector2();
```

```
        delete s2;
}
```



객체도 자유롭게 동적으로 생성이 가능하다.

4. 깊은 복사와 얕은 복사

```
#include <iostream>

using namespace std;

int main() {
        int *a = new int(5);
        int *b = new int(3);

        a = b; // 얕은 복사 (참조를 복사)
        *a = *b; // 깊은 복사 (값을 복사)

        delete a;
}
```

(두 포인터가 변수를 가리킨다고 가정)

얕은 복사: 저장하는 주소 값을 전달해서 가리키는 변수를 변경시킨다.

<자신이 가리키고 있는 것을 남도 가리키게 한다>

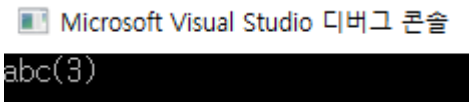
깊은 복사: 가리키는 변수의 값을 전달해서 가리키는 변수의 값을 변경시킨다.

<자신이 가리키고 있는 것과 같은 형태로 하나 더 만든다>

<string 클래스 직접 구현하기>

```
#pragma warning (disable:4996)
#include <iostream>

using namespace std;

class String {
public:
        String() {
                strData = NULL;
                len = 0;
```

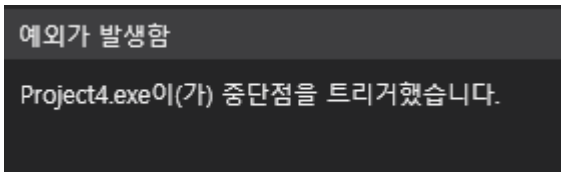
```
        }
        String(const char* str) {
                len = strlen(str);
                strData = new char[len+1];
                strcpy(strData, str); // strcpy도 깊은 복사이다.
        }
        ~String() {
                delete[] strData;
        }
        char* GetStrData() const { return strData; }
        int GetLen() const { return len; }
private:
        char* strData;
        int len;
};

int main() {
        String a;
        String b("abc");
        // char*를 출력할 때는 문자열이 출력된다. (%s 형식 지정자 사용)
        cout << b.GetStrData() << "(" << b.GetLen() << ")" << endl;
}
```



5. 복사 생성자 (멤버를 복사하는 생성자)

```
#pragma warning (disable:4996)
#include <iostream>

using namespace std;

class String {
public:
        String() {
                strData = NULL;
                len = 0;
        }
        String(const char* str) {
                len = strlen(str);
                strData = new char[len+1];
                strcpy(strData, str); // strcpy도 깊은 복사이다.
        }
        ~String() {
                delete[] strData;
        }
        char* GetStrData() const { return strData; }
        int GetLen() const { return len; }
private:
        char* strData;
        int len;
```

```
};

int main() {
	String s1("안녕");
	String s2(s1); // 복사 생성자 (개발자가 만든 생성자가 아님)

	cout << s1.GetStrData() << "(" << s1.GetLen() << ")" << endl;
	cout << s2.GetStrData() << "(" << s2.GetLen() << ")" << endl;
	// s2의 소멸자 호출 시 문제 생김 -> strData가 가리키는 배열이 동일한데 이미
삭제됐으므로
}
```



<기본적인 복사 생성자의 형태>

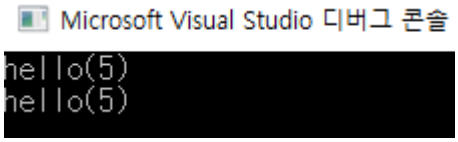
```
String(String& rhs) {
	strData = rhs.strData; // 얕은 복사
	len = rhs.len; // 깊은 복사
}
```

➔ **얕은 복사가 일어나면 같은 메모리 공간을 가리키므로 delete을 여러 번 할 경우 문제가 발생할 수 있음**

<임의로 복사 생성자 변경>

```
String(String& rhs) {
	len = rhs.len; // 깊은 복사
	strData = new char[len+1];
	strcpy(strData, rhs.strData); // 깊은 복사
}
```

➔ **모두 깊은 복사로 바꿔 버리면 문제가 발생하지 않는다.**



6. 대입 연산자 오버로딩

```
#pragma warning (disable:4996)
#include <iostream>

using namespace std;

class String {
public:
	String() {
		strData = NULL;
		len = 0;
	}
	String(const char* str) {
		len = strlen(str);
		strData = new char[len+1];
		strcpy(strData, str); // strcpy도 깊은 복사이다.
	}
	String(const String& rhs) {
		len = rhs.len; // 깊은 복사
		strData = new char[len+1];
		strcpy(strData, rhs.strData); // 깊은 복사
	}
	~String() {
		delete[] strData;
	}
	String& operator=(const String& rhs) {
		if (this == &rhs) // 같은 객체가 왔을 때
			return *this;
		delete[] strData;
		len = rhs.len; // 깊은 복사
		strData = new char[len + 1];
		strcpy(strData, rhs.strData); // 깊은 복사
		return *this;
	}
	char* GetStrData() const { return strData; }
	int GetLen() const { return len; }
private:
	char* strData;
	int len;
};

int main() {
	String s1("hello");
	String s2(s1); // 복사 생성자
	String s3;
	s3 = s1; // 연산자 오버로딩 (=) s3.operator=(s1);

	cout << s1.GetStrData() << "(" << s1.GetLen() << ")" << endl;
	cout << s2.GetStrData() << "(" << s2.GetLen() << ")" << endl;
	cout << s3.GetStrData() << "(" << s3.GetLen() << ")" << endl;
```

Microsoft Visual Studio 디버그 콘솔

```
hello(5)
hello(5)
hello(5)
}
```

연산자 오버로딩을 이용해서 깊은 복사를 하는 모습이다.

여기서 살펴볼 점은 매개변수 타입이 'String&'라는 점. 반환형이 'String&'라는 점이다.

일단 Call by reference이기 때문에 원본 자체(s1객체)가 전달된다.

String&은 별명이 전달되는 것이 아니라 String형이 전달되는데 원본이 간다고 생각해야 된다.

따라서 rhs를 return하면 String을 return하는 것이니 오류가 발생한다.

함수의 반환형은 String&로 (원본에 해당하는 String객체)이다. 단순히 rhs라고 하면 이것이 원본인지 복사본인지 컴파일러가 알 수가 없다.

따라서 매개변수로 받은 객체는 반환할 수 없고 this가 가리키는 객체 그 자체를 반환한 것이다.

하지만 어차피 원본이 가서 원본이 수정되니 void로 반환하는 것이 가장 깔끔하다고 생각한다.

그리고 s3=s1이란 것 자체가 s3.operator=(s1)이니 반환 값을 다시 저장하는 개념도 아니다.

```
	void operator=(const String& rhs) {
		if (this != &rhs) { // 같은 객체가 왔을 때
			delete[] strData;
			len = rhs.len; // 깊은 복사
			strData = new char[len + 1];
			strcpy(strData, rhs.strData); // 깊은 복사
		}
	}
```

7. 함수에서 객체를 반환하기

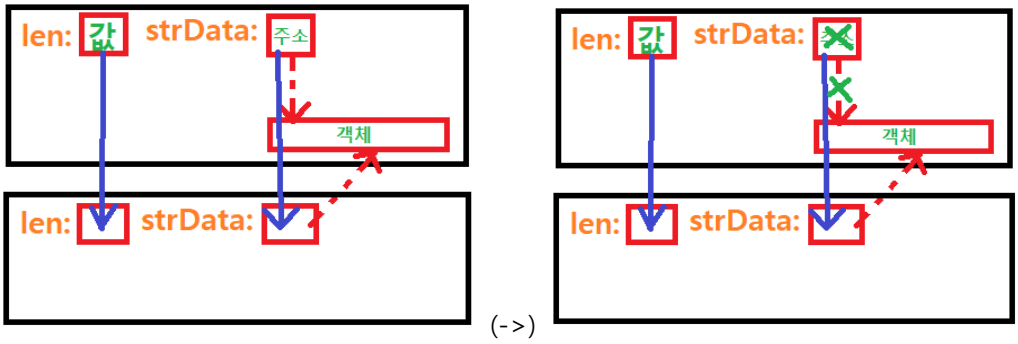
r-value: 저장이 안 되는 변수 (오른쪽에만 위치한다. 단, 정의 선언을 같이 할 때는 예외 ex) 객체, 배열, 함수가 반환한 것)

l-value: 저장이 되는 변수 (왼쪽,오른쪽 둘 다 위치할 수 있다. ex) 변수, 포인터)

```
String getName() {
	String res("Doodle");
	cout << "hi2" << endl;
	/* 객체를 리턴할 때는 항상 복사 생성자를 호출해서 임시 객체를 리턴한다.
	   즉 return String temp(res);를 한다고 생각하면 된다.
	   근데 이 과정에서 깊은 복사가 일어나서 메모리 낭비가 발생할 수 있다.
	   따라서 r-value(저장이 안 되는 변수)를 매개변수로 받는 복사 생성자를 만들고
	   이동 시맨틱을 하면 되다.
	   이동 시맨틱은 얕은 복사를 한 다음 해제를 해서 뒷 문제(같은 것을 delete)가 발생하지 않게 하는 것이다.
	*/
	return res;
}
```

<이동 시맨틱>

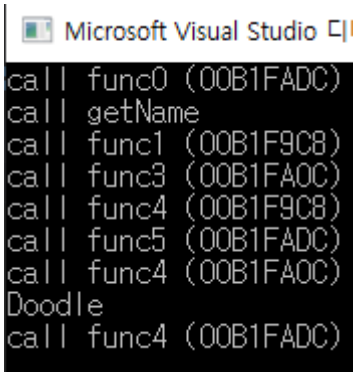
```
String(String&& rhs) { // 매개변수가 r-value
        cout << "hi" << endl;
        len = rhs.len;
        strData = rhs.strData;
        rhs.strData = NULL;
}
```



(파란 실선: 전달(len:깊은 복사, strData:얕은 복사),빨간 점선: 주소에 해당하는 객체, 초록 엑스: 해제[null주기])

```
#pragma warning (disable:4996)
#include <iostream>

using namespace std;

class String {
public:
        String() {
                cout << "call func0" << " (" << this << ")" << endl;
                strData = NULL;
                len = 0;
        }
        String(const char* str) {
                cout << "call func1" << " (" << this << ")" << endl;
                len = strlen(str);
                strData = new char[len+1];
                strcpy(strData, str); // strcpy도 깊은 복사이다.
        }
        String(const String& rhs) {
                cout << "call func2" << " (" << this << ")" << endl;
                len = rhs.len; // 깊은 복사
                strData = new char[len+1];
                strcpy(strData, rhs.strData); // 깊은 복사
        }
        String(String&& rhs) { // 매개변수가 r-value
                cout << "call func3" << " (" << this << ")" << endl;
                len = rhs.len;
                strData = rhs.strData;
                rhs.strData = NULL;
```

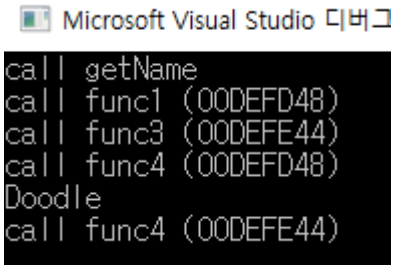
```
        }
        ~String() {
                cout << "call func4" << " ("<<this<<")"<<endl;
                delete[] strData;
        }
        void operator=(const String& rhs) {
                cout << "call func5" << " (" << this << ")" << endl;
                if (this != &rhs) { // 같은 객체가 안 왔을 때
                        delete[] strData;
                        len = rhs.len; // 깊은 복사
                        strData = new char[len + 1];
                        strcpy(strData, rhs.strData); // 깊은 복사
                }
        }
        char* GetStrData() const { return strData; }
        int GetLen() const { return len; }
private:
        char* strData;
        int len;
};

String getName() {
        cout << "call getName" << endl;
        String res("Doodle");
        return res;
}

int main() {
        String s1;
        s1=getName();
        cout << s1.GetStrData() << endl;
}
```



main 첫 줄에서 s1(FADC)객체 생성[func0] -> getName()에서 res(F9C8)객체 생성[func1] -> getName()의 return절에서 임시(FA0C)객체 생성<이동 시맨틱>[func3] -> getName()에서 리턴하고 res(F9C8)객체 해제[func4] -> main의 연산자 오버로딩으로 깊은 복사[func5] -> s1선언이 완료되고 임시(FA0C)객체 해제[func4] -> s1 객체 멤버 출력 -> main끝나고 s1(FACD)객체 해제[func4]

[참고]

```
int main() {
	String s1 =getName();
	cout << s1.GetStrData() << endl;
}
```



위처럼 수정할 시 s1선언이 끝나도 임시 객체는 사라지지 않는다. 그냥 임시 객체를 s1으로 쓰는 형태이다. (객체 생성[func0],깊은 복사 과정 빠짐[func5]) -> 정의,선언 한 번에 하는 것이 이득

8. 묵시적 형변환

```
#pragma warning (disable:4996)
#include <iostream>

using namespace std;

class Item {
public:
	Item(int num) :num(num) { // 변환 생성자
		cout << "call_start" << this << endl;
	}
	~Item() {
		cout << "call_end" <<this<< endl;
	}
private:
	int num;
};

int main() {
	cout << "1:" << " ";
	Item i1=Item(1); // 첫 번째 생성자 호출
	cout << "2:" << " ";
	Item i2(2); // 첫 번째 생성자 호출
	cout << "3:" << " ";
	Item i3 = 3; // 첫 번째 생성자 호출
	cout << "3(update):" << " ";
	i3 = 3; // 묵시적 형변환 -> i3=(Item)3; -> 첫 번째 생성자 호출해서 객체 생성하고 i3로 얕은 복사
	cout << "4:" << " ";
	Item i4 = (Item)4; // 명시적 형변환 -> 첫 번째 생성자 호출해서 객체 생성하고 그 객체를 i4로 씀
	cout << "main_end" << endl;
```

}

Microsoft Visual Studio 디버그 콘솔

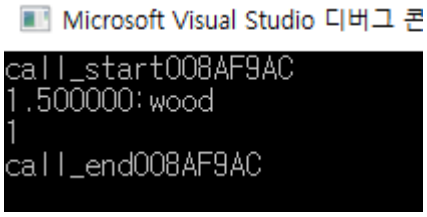
```
1: call_start00D8FA5C
2: call_start00D8FA34
3: call_start00D8FA0C
3(update): call_start00D8F8FC
call_end00D8F8FC
4: call_start00D8F9E4
main_end
call_end00D8F9E4
call_end00D8FA0C
call_end00D8FA34
call_end00D8FA5C
```

(Item)을 이용해서 명시적으로 형변환이 가능하고 그냥 묵시적으로도 가능하다.

9. 형변환 연산자 오버로딩

```
#pragma warning (disable:4996)
#include <iostream>
#include <string>

using namespace std;

class Item {
public:
	Item(double num,string name) :num(num), name(name) {
		cout << "call_start" << this << endl;
	}
	~Item() {
		cout << "call_end" <<this<< endl;
	}
	// int로의 형변환 연산자 오버로딩
	operator int() const {
		return num;
	}
	// string으로의 형변환 연산자 오버로딩
	operator string() const {
		return to_string(num) + ":" + name;
	}
private:
	double num;
	string name;
};

void println(string s) {
	cout << s << endl;
}

void println(int s) {
	cout << s << endl;
```

```
}

int main() {
        Item i1{ 1.5,"wood" };
        println((string)i1); // 명시적 형변환 후 출력
        println((int)i1); // 명시적 형변환 후 출력
        /* 오류
        println(i1); // 묵시적 형변환을 int로 할지 string으로 할지 모름
        */
}
```



형변환 연산자 오버로딩은 반환형을 뒤로 적는 특징이 있다. 해당 객체에 대해 형변환을 진행한다.

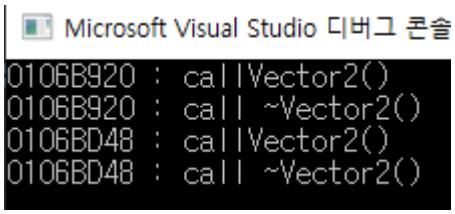
TMI) explicit 을 변환 생성자 앞에 작성하면 묵시적 형변환이 불가능하다.

ex)

```
explicit Item(int num, string name) : num(num), name(name) {
    cout << "Item(int, string)" << endl;
}
```

10. 스마트 포인터

```
#include <iostream>

using namespace std;

class Vector2 {
public:
	Vector2() :x(0), y(0) {
		cout << this << " : callVector2()" << endl;
	}
	~Vector2() {
		cout << this << " : call ~Vector2()" << endl;
	}
	float GetX() const { return x; }
	float GetY() const { return y; }
private:
	float x;
	float y;
};

int main() {
	// 일반 포인터가 동적 객체를 가리킬 때
	Vector2* s2 = new Vector2();
	// delete연산자로 삭제해야 됨
	delete s2;

	// 스마트 포인터가 동적 객체를 가리킬 때
	unique_ptr<Vector2> s3(new Vector2());
	// delete연산자로 삭제할 필요 없음 (자동 삭제)
}
```

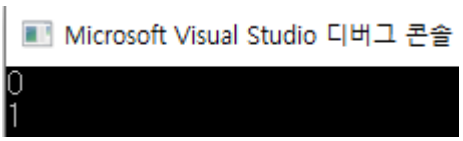


스마트 포인터를 사용하면 귀찮게 delete 연산을 할 필요가 없어진다.

C++ 프로그래밍 (객체지향 프로그래밍)

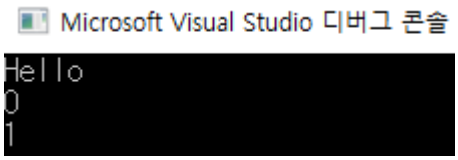
1. 상속과 접근 제어

```
#pragma warning (disable:4996)
#include <iostream>

using namespace std;

class Base {
public:
	void bFunc() {
		cout << "Hello" << endl;
	}
	int bNum;
};

class Derived :public Base {
public:
	void dFunc() {
		cout << "Hello" << endl;
	}
	int dNum;
};

int main() {
	Derived d;
	d.dNum = 0; // d객체의 고유 멤버
	d.bNum = 1; // d객체의 상속받은 멤버
	cout << d.dNum << endl;
	cout << d.bNum << endl;
}
```



상속을 이용하면 부모의 멤버와 메소드를 물려 받는다.

```cpp
#pragma warning (disable:4996)
#include <iostream>

using namespace std;

class Base {
public:
	void bFunc() {
		cout << "Hello" << endl;
	}
protected: // 외부 접근x, 자식 클래스 접근 가능
	int bNum;
};

class Derived :public Base {
public:
	void dFunc() {
		cout << "Hello" << endl;
		dNum = 0;
		bNum = 1;
	}
	int get_dNum() const { return dNum; }
	int get_bNum() const { return bNum; }

private:
	int dNum;
};

int main() {
	Derived d;
	d.dFunc();
	cout << d.get_dNum() << endl;
	cout << d.get_bNum() << endl;
}
```



자식 클래스에서 public으로 상속 받음

➔ 부모 클래스의 public, protected 멤버를 그대로 가져옴

자식 클래스에서 protected로 상속 받음

➔ 부모 클래스의 public, protected 멤버를 protected로 바꿔서 가져옴 (외부에서 접근 불가)

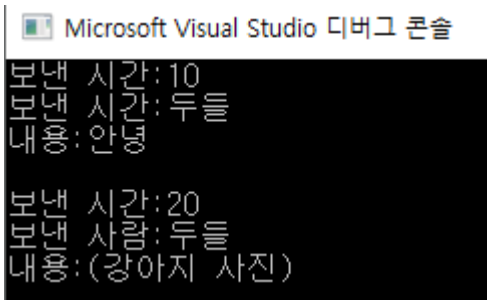
자식 클래스에서 private로 상속 받음

➔ 부모 클래스의 public, protected 멤버를 private로 바꿔서 가져옴 (외부,자식에서 접근 불가)

2. 상속이 필요한 이유

```
#pragma warning (disable:4996)
#include <iostream>
#include <string>

using namespace std;

class Image {
public:
	Image(string image_txt) {
		this->image_txt=image_txt;
	}
	operator string() {
		return "("+image_txt+" 사진)";
	}
private:
	string image_txt;
};

class Message {
public:
	Message(int sendTime, string sendName) {
		this->sendTime = sendTime;
		this->sendName = sendName;
	}
	int get_sendTime() const { return sendTime; }
	string get_sendName() const { return sendName; }
private:
	int sendTime;
	string sendName;
};


class ImageMessage:public Message {
public:
	// 부모의 생성자 접근 -> 본인 생성자 접근
	ImageMessage(int sendTime, string sendName, Image* image)
		:Message(sendTime,sendName) {
		this->image = image;
	}
	Image* get_image() const { return image; }

private:
	Image* image;
};

class TextMessage :public Message {
public:
	// 부모의 생성자 접근 -> 본인 생성자 접근
	TextMessage(int sendTime, string sendName, string text)
		:Message(sendTime, sendName) {
		this->text = text;
	}
	string get_text() const { return text; }
```

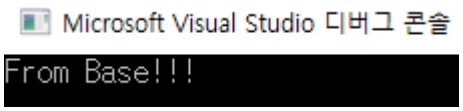
```
private:
        string text;
};

int main() {
        // 두들: "안녕"
        // 두들: "(강아지 사진)"
        Image* p_dogImage = new Image("강아지");
        TextMessage* hello = new TextMessage(10, "두들", "안녕");
        ImageMessage* dog = new ImageMessage(20, "두들", p_dogImage);

        cout << "보낸 시간:" << hello->get_sendTime() << endl;
        cout << "보낸 시간:" << hello->get_sendName() << endl;
        cout << "내용:" << hello->get_text() << endl << endl;

        cout << "보낸 시간:" << dog->get_sendTime() << endl;
        cout << "보낸 사람:" << dog->get_sendName() << endl;
        cout << "내용:" << (string)*(dog->get_image()) << endl;

        delete hello;
        delete dog;
        delete p_dogImage;
}
```



상속을 해서 부모 생성자를 접근하고 자식 생성자를 접근하면 코드량을 확실히 줄일 수 있다. 상속받은 멤버 초기화는 부모에서만 진행해도 되기 때문이다.

3. 정적 바인딩

```
#pragma warning (disable:4996)
#include <iostream>
#include <string>

using namespace std;

class Base {
public:
	int a = 10; // 임의 생성자에서 멤버 초기화함
	void print() {
		cout << "From Base!!!" << endl;
	}
};

class Derived :public Base {
public:
	int a = 20;
	 // 오버라이딩
	void print() {
		cout << "From Derived!!!" << endl;
	}
};

int main() {
	// 부모 객체를 가리킬 수 있는 포인터는 자식 객체도 가리킬 수 있다.
	Base* b = new Derived();
	// 정적 바인딩 (부모 포인터로 자식 객체를 가리킬 때는 부모 메소드가 호출된다)
	b->print(); // 부모 객체의 메소드 호출
	delete b;
	Base b;
}
```

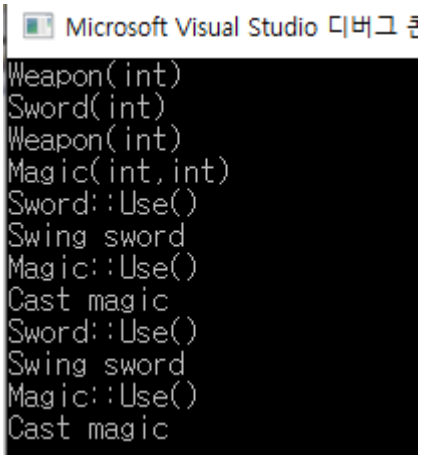


정적 바인딩이라는 것은 '부모 포인터로 자식 객체를 가리킬 때 부모 메소드가 호출된다'라는 것이다. 오버라이딩된 것은 상관없다.

4. 가상 함수와 동적 바인딩

```
#pragma warning (disable:4996)
#include <iostream>
#include <string>

using namespace std;

class Weapon {
public:
	Weapon(int power) :power(power) {
		cout << "Weapon(int)" << endl;
	}
	virtual void Use() { // 가상 함수
		cout << "Weapon::Use()" << endl;
	}
protected:
	int power;
};

class Sword :public Weapon {
public:
	Sword(int power) :Weapon(power) {
		cout << "Sword(int)" << endl;
	}
	void Use() {
		cout << "Sword::Use()" << endl;
		Swing();
	}
private:
	void Swing() {
		cout << "Swing sword" << endl;
	}
};

class Magic :public Weapon {
public:
	Magic(int power, int manaCost) :Weapon(power), manaCost(manaCost) {
		cout << "Magic(int,int)" << endl;
	}
	void Use() {
		cout << "Magic::Use()" << endl;
		Cast();
	}
private:
	void Cast() {
		cout << "Cast magic" << endl;
	}
	int manaCost;
};

int main() {
	Sword s(10);
	Magic m(15, 7);
	s.Use();
```

```
        m.Use();

        Weapon* w;
        w = &s;
        // virtual처리->동적 바인딩
        // (부모 포인터로 자식 객체를 가리킬 때 오버라이딩된 자식 메소드가 호출된다)
        w->Use();

        w = &m;
        w->Use(); // 동적 바인딩
}
```



동적 바인딩이라는 것은 '부모 포인터로 자식 객체를 가리킬 때 오버라이딩된 자식 메소드가 호출된다'라는 것이다. 동적 바인딩이 되려면 부모에 있는 해당 메소드가 가상 함수여야만 한다.

5. 상속이 필요한 이유 2

```
#pragma warning (disable:4996)
#include <iostream>
#include <string>

using namespace std;

class Image {
public:
	Image(string image_txt) {
		this->image_txt = image_txt;
	}
	operator string() {
		return "(" + image_txt + " 사진)";
	}
private:
	string image_txt;
};

class Message {
public:
	Message(int sendTime, string sendName) {
		this->sendTime = sendTime;
		this->sendName = sendName;
	}
	int get_sendTime() const { return sendTime; }
	string get_sendName() const { return sendName; }
	virtual string get_content() const { return ""; }

private:
	int sendTime;
	string sendName;
};


class ImageMessage :public Message {
public:
	// 부모의 생성자 접근 -> 본인 생성자 접근
	ImageMessage(int sendTime, string sendName, Image* image)
		:Message(sendTime, sendName) {
		this->image = image;
	}
	string get_content() const { return (string)*image; } // override

private:
	Image* image;
};

class TextMessage :public Message {
public:
	// 부모의 생성자 접근 -> 본인 생성자 접근
	TextMessage(int sendTime, string sendName, string text)
		:Message(sendTime, sendName) {
		this->text = text;
```

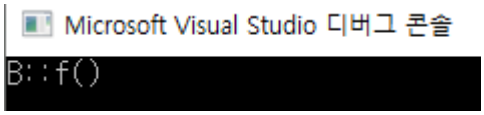
```
        }
        string get_content() const { return text; } // override

private:
        string text;
};

void printMessage(Message* m) {
        cout << "보낸 시간:" << m->get_sendTime() << endl;
        cout << "보낸 시간:" << m->get_sendName() << endl;
        cout << "내용:" << m->get_content() << endl << endl;
}

int main() {
        Image* p_dogImage = new Image("강아지");

        // 포인터 배열
        Message* messages[] = {
                new TextMessage(10,"두들","안녕"),
                new TextMessage(11,"두들","안녕"),
                new TextMessage(12,"두들","안녕"),
                new ImageMessage(20, "두들", p_dogImage)
        };

        for (Message* m : messages)
                printMessage(m);

        delete p_dogImage;
}
```

```
Microsoft Visual Studio 디버그 콘솔
보낸 시간:10
보낸 시간:두들
내용:안녕

보낸 시간:11
보낸 시간:두들
내용:안녕

보낸 시간:12
보낸 시간:두들
내용:안녕

보낸 시간:20
보낸 시간:두들
내용:(강아지 사진)
```

오버라이딩함(get_content()) -> printMessage를 오버로딩해서 만들지 않고 한 함수로 처리 가능해짐

6. 순수 가상함수와 추상 클래스

```
#pragma warning (disable:4996)
#include <iostream>
#include <string>

using namespace std;

// 추상 클래스 (순수 가상 함수를 가진 클래스)
class A {
public:
	virtual void f() = 0; // 순수 가상 함수
};

class B :public A {
public:
	// override
	void f() {
		cout << "B::f()" << endl;
	}
};

int main() {
	A *a = new B;
	a->f();
}
```



순수 가상 함수는 기능을 하지 않는 가상 함수이다.

즉 오버라이딩을 해야 기능을 한다.

그리고 순수 가상 함수를 가진 클래스를 추상 클래스라고 한다.

이 추상 클래스는 인스턴스화가 불가능하다.

하지만 추상 클래스를 상속받아 오버라이딩하는 자식 클래스가 있다면 그 자식을 인스턴스화할 수는 있다.

(다른 예제)

```
#pragma warning (disable:4996)
#include <iostream>
#include <string>
#define PI 3.141592

using namespace std;

class Shape {
public:
	virtual double get_area() const = 0;
	virtual void resize(double f) = 0;
};

class Circle: public Shape {
public:
	Circle(double r) :r(r) {}
	// override
	double get_area() const { return PI * r * r; }
	void resize(double f) { r*=f; }

private:
	double r;
};

class Rectangle : public Shape {
public:
	Rectangle(double a, double b) :a(a), b(b) {}
	// override
	double get_area() const { return a * b; }
	void resize(double f) { a *= f; b *= f; }
private:
	double a, b;
};

int main() {
	Shape* s[] = {
		new Circle(10),
		new Rectangle(20,30)
	};
	for (Shape* s : s) {
		s->resize(2);
	}
	for (Shape* s : s) {
		cout << s->get_area() << endl;
	}
	for (Shape* s : s) {
		delete s;
	}
}
```

Microsoft Visual Studio 디버그 콘솔

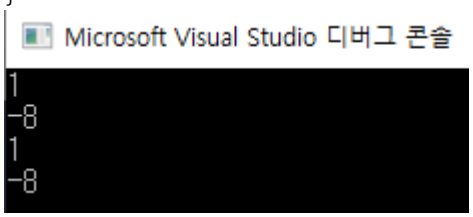
```
1256.64
2400
```

공통된 변수 멤버는 없지만 공통된 메소드는 존재할 때

추상 클래스를 이용한다.

7. 함수 객체와 람다식

```
#pragma warning (disable:4996)
#include <iostream>

using namespace std;

class Square {
public:
	int operator()(int n) { return n * n; }
}square;

class MyFunc {
public:
	int operator()(int n) { return n * 2; }
}myFunc;

template<typename T>
int arrFnMin(const int arr[], int n, T f) {
	int min = f(arr[0]);
	for (int i = 1; i < n; i++) {
		if (f(arr[i]) < min)
			min = f(arr[i]);
	}
	return min;
}

int main() {
	int arr[7] = { 3,1,-4,1,5,9,-2 };
	// 객체 함수를 이용
	cout << arrFnMin(arr, 7, square) << endl;
	cout << arrFnMin(arr, 7, myFunc) << endl;
	// 람다식을 이용
	cout << arrFnMin(arr, 7, [](int n)->int {return n * n; }) << endl;
	cout << arrFnMin(arr, 7, [](int n)->int {return n * 2; }) << endl;
}
```



함수 객체와 람다식을 이용해서 매개변수로 함수를 보내고 최소값을 찾는 모습이다.

(auto)

컴파일러가 자동으로 어떤 타입을 가져야 할지 알도록 함

// 주로 람다식을 정의할 때 사용함

```
int main() {
	int arr[7] = { 3,1,-4,1,5,9,-2 };
	// 객체 함수를 이용
	cout << arrFnMin(arr, 7, square) << endl;
	cout << arrFnMin(arr, 7, myFunc) << endl;
	// 람다식을 이용
	auto f1 = [](int n)->int {return n * n; };
	auto f2 = [](int n)->int {return n * 2; };
	cout << arrFnMin(arr, 7, f1) << endl;
	cout << arrFnMin(arr, 7, f2) << endl;
}
```

➔ **결과는 같음**

C++ 프로그래밍 (STL 프로그래밍)

STL: standard template library의 약자로서 많은 프로그래머들이 공통적으로 사용하는 자료구조와 알고리즘을 구현한 클래스들로 이루어져 있다. STL은 템플릿 기법을 사용하였기 때문에 어떤 자료형에 대해서도 사용할 수 있다.

템플릿: 다양한 자료형을 사용할 수 있도록 클래스를 설계하는 기법

(템플릿으로 작성된 클래스를 사용할 때 원하는 자료형을 지정한다.)

ex) vector<배열의 자료형> 배열의 이름(배열의 크기);

1. 벡터 (Vector)

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>

using namespace std;

class Person {
public:
        Person(string n, int a) :name(n), age(a) {}
        string get_name() const { return name; }
        int get_age() const { return age; }
        void print() {
                cout << name << " " << age << endl;
        }

public:
        string name;
        int age;
};

// 특정 멤버를 기준으로 오름차순
bool compare(Person& p, Person& q) {
        return p.get_age() < q.get_age();
}

int main() {
        // 벡터 및 반복자 생성
        vector<Person> v;
        vector<Person>::iterator it;

        // 벡터의 뒷 부분에서 추가
        v.push_back(Person("Kim", 30));
        v.push_back(Person("Park", 22));
```

```
        v.push_back(Person("Lee", 26));
        v.push_back(Person("Ahn", 23));
        v.push_back(Person("Hyun", 19));
        v.push_back(Person("Park", 22));

        // 벡터의 특정 영역에 추가
        // v.begin() -> 벡터의 시작 주소(첫 원소 주소)
        // v.end() -> 벡터의 끝 주소(마지막 원소에서 한 칸 더 간 주소)
        v.insert(v.begin(), Person("Kim2", 31));
        v.insert(v.begin() + 1, Person("Kim3", 32));
        v.insert(v.begin() + 2, Person("Kim4", 32));

        // 벡터의 특정 영역 제거
        v.erase(v.begin());
        v.erase(v.begin(), v.begin() + 2); // begin~begin+1

        // 벡터의 뒷 부분에서 제거
        v.pop_back();
        v.pop_back();

        // 안정정렬
        sort(v.begin(), v.end(), compare);

        // 벡터 원소 출력 (반복자 사용)
        for (it = v.begin(); it != v.end(); it++)
                it->print();
        cout << endl;

        // 벡터 원소 출력 (범위 기반 for문)
        for (auto& e : v)
                e.print();

        // 벡터의 모든 요소 삭제
        if (!v.empty()) {
                cout <<"벡터 원소"<< v.size() <<"개 삭제"<<endl;
                v.clear();
        }

        return 0;
}
```

Microsoft Visual Studio 디버그 콘솔

```
Park 22
Ahn 23
Lee 26
Kim 30

Park 22
Ahn 23
Lee 26
Kim 30
벡터 원소4개 삭제
```

2. 스택 (Stack)

```
#include <iostream>
#include <stack>

using namespace std;

int main() {
	stack<int> st;
	int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };

	// 삽입
	for(auto& a : arr)
		st.push(a);

	// 삭제
	while (!st.empty()) {
		cout << st.top() <<", size="<<st.size()<< endl;
		st.pop();
	}

	return 0;
}
```

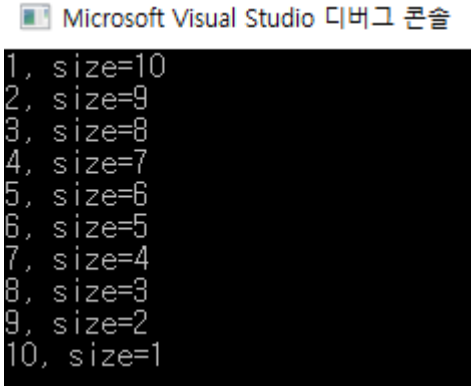
Microsoft Visual Studio 디버그 콘솔

```
10, size=10
9, size=9
8, size=8
7, size=7
6, size=6
5, size=5
4, size=4
3, size=3
2, size=2
1, size=1
```

3. 큐 (Queue)

```
#include <iostream>
#include <queue>

using namespace std;

int main() {
	queue<int> q;
	int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };

	// 삽입
	for(auto& a : arr)
		q.push(a);

	// 삭제
```

```
        while (!q.empty()) {
                cout << q.front() <<", size="<< q.size()<< endl;
                q.pop();
        }

        return 0;
}
```



4-1. 우선순위 큐 (Priority Queue-Max Heap)

```
#include <iostream>
#include <queue>

using namespace std;

int main() {
        priority_queue<int> pq;
        int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };

        // 삽입
        for(auto& a : arr)
                pq.push(a);

        // 삭제
        while (!pq.empty()) {
                cout << pq.top() <<", size="<< pq.size()<< endl;
                pq.pop();
        }

        return 0;
}
```

Microsoft Visual Studio 디버그 콘솔

```
10, size=10
9, size=9
8, size=8
7, size=7
6, size=6
5, size=5
4, size=4
3, size=3
2, size=2
1, size=1
```

4-2. 우선순위 큐 (Priority Queue-Min Heap)

```
#include <iostream>
#include <queue>

using namespace std;

int main() {
	priority_queue<int, vector<int>,greater<int>> pq;
	int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };

	// 삽입
	for(auto& a : arr)
		pq.push(a);

	// 삭제
	while (!pq.empty()) {
		cout << pq.top() <<", size="<< pq.size()<< endl;
		pq.pop();
	}

	return 0;
}
```

Microsoft Visual Studio 디버그 콘솔

```
1, size=10
2, size=9
3, size=8
4, size=7
5, size=6
6, size=5
7, size=4
8, size=3
9, size=2
10, size=1
```

5. 덱 (Deque)

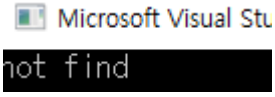
```cpp
#include <iostream>
#include <deque>

using namespace std;

int main() {
	deque<int> dq;
	int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };

	// 삽입
	for (auto& a : arr) {
		if (a <= 5)
			dq.push_back(a); // 1 2 3 4 5 <- 후단 삽입
		else
			dq.push_front(a); // 전단 삽입 -> 10 9 8 7 6
	}
	// 10 9 8 7 6 1 2 3 4 5

	// 후단 삭제
	while (dq.size()>5) {
		cout << dq.back() <<", size="<< dq.size()<< endl;
		dq.pop_back();
	}

	// 전단 삭제
	while (!dq.empty()) {
		cout << dq.front() << ", size=" << dq.size() << endl;
		dq.pop_front();
	}

	return 0;
}
```

Microsoft Visual Studio 디버그 콘

```
5, size=10
4, size=9
3, size=8
2, size=7
1, size=6
10, size=5
9, size=4
8, size=3
7, size=2
6, size=1
```

6. 집합 (Set)

```
#include <iostream>
#include <set>

using namespace std;

int main() {
	set<int> s;

	s.insert(1);
	s.insert(2);
	s.insert(3);
	s.erase(2);

	// 집합 원소 못 찾으면 end() 반환
	auto pos = s.find(2);
	if (pos != s.end())
		cout << "find" << endl;
	else
		cout << "not find" << endl;

	return 0;
}
```



7. 맵 (Map)

```cpp
#include <iostream>
#include <string>
#include <map>

using namespace std;

int main() {
	map<string, string> m;

	// 삽입
	m["Ahn"] = "010-1234-5678";
	m["Park"] = "010-2345-6789";
	m["Ho"] = "010-3456-7890";
	m.insert({"Kim","010-1212-3434"});

	// 삭제
	m.erase("Kim");

	// 검색
	if (m.find("Ho") == m.end())
		cout << "not find" << endl;

	// 출력
	for (auto& e : m)
		cout << e.first << ":" << e.second << endl;

	return 0;
}
```