```
//스마트 포인터 (저격)
#include <iostream> #include <memory>
struct Widget {
int data;
Widget(int n) : data(n) {}
~Widget() { std::cout << "Destroying: " << data << std::endl; };
//소멸자는 생성자 문장에 그대로 이어서쓰면 된다.
};
int main() {
//new 연산자는 피연산자를 동적 메모리에 할당할 뿐만 아니라 피연산자를 저장한 메모리의 위치를 반환한다.
std::shared_ptr<Widget> sp1(new Widget(10));
//shared_ptr은 참조 카운터를 힙에 주가 할당하여 메모리 누수를 해결한다.
//참조 카운터는 하나의 객체에 대한 shared_ptr가 생성/제거될 때 +-1되며,
//0이 되면 여태까지의 shared_ptr의 동적 할당을 전부 해제하고 대상으로 소멸시킨다.
/* new Widget(10)에 의해 Widget의 객체가 힙에 생성되고, (애초에 우측값이 아니다)
그 주소를 인수로 하여 shared_ptr이 생성된다.
그리고 shared_ptr은 그 주소와 참조 카운터를 가진다.
즉 해당 코드는 일반 포인터를 스마트 포인터로 다시 태어나게 하는 것이다. */
std::shared_ptr<Widget> sp2 = std::make_shared<Widget>(20);
/* shared_ptr를 복사 생성할 경우 반드시 복사할 데이터를 make_shared를 통해 만들어주어야 한다.
shared_ptr은 힙에 데이터와 참조 카운터 이 두가지를 할당하기 때문이다.
즉 그냥 동적 메모리 할당만 하는 타입을 통해 복사 생성하면 불충분하다. */
/* 한편 std::make_shared<Widget>(20)은 shared_ptr 타입의 임시 객체이다.
그리고 sp2는 그 임시 객체가 가리키는 대상을 가리키는 또 다른 주소가 된다. (아마 포인터 복사 생성 특징일 것이다.)
즉 복사 과정이 일절 없다는 것이다! 이제 문장이 끝나면 임시 객체는 죽는다.
허나 임시 객체가 가리켰던 대상은 아직 sp2가 가리키고 있으므로 죽지 않는다.*/
auto sp3 = sp2;
//shared_ptr 간의 복사&할당은 하나의 데이터에 대해 가리키는 주소가 두가지가 된다고 볼 수 있다.
std::cout << sp2.use_count() << std::endl;
//use_count를 쓰면 shared_ptr의 대상을 공유하고 있는 shared_ptr의 개수를 알 수 있다.
sp1.reset(); //데이터에 대한 주소는 하나였으니 데이터도 죽어서 소멸자가 작동한다.
sp2.reset();
sp3.reset(); //비로소 모든 주소가 죽었기 때문에 데이터도 죽어서 소멸자가 작동한다.
//포컬에서 생성된 shared_ptr은 로컬이 끝나면 죽는다. 평범한 동적 할당이 프로그램이 끝나야 죽는 것과 대조적이다.
std::cout << (bool)sp1 << std::endl;
//nullptr은 false다~
Widget* sp7 = new Widget(10);
auto sp8(sp7);
auto sp9(sp7);
//이 경우는 서로 가리키는 대상은 같지만 참조 카운터가 다른 shared_ptr이 생성된다!!
//만약 sp8을 reset해버리면 대상이 죽어버리기에 sp9은 눈 뜨고 코베인 상황이 되어버리므로 오류가 된다.
std::unique_ptr<Widget> sp4 = std::make_unique<Widget>(30);
//auto sp5 = sp4;
//한편 unique_ptr은 하나의 대상에 대해 스마트 포인터를 여러가지 만들 수 없다!
auto sp5 = sp4.release();
//realease는 스마트 포인터의 주소를 반환하고 그 스마트 포인터를 nullptr로 만든다.
auto sp6 = std::move(sp5);
//move는 우측값으로 만들어버리는 함수라는 것을 떠올린다면... sp5를 고자로 만들어버리고
//sp6가 sp5의 주소를 ntr한다고 볼 수 있겠다.
} //main 함수가 끝나서 스마트 포인터들이 모두 죽으면 마침내 데이터도 죽으니 소멸자가 작동한다.

#include <iostream>
template <class T1, class T2>
double aver(T1 a, T2 b) { //다른 타입을 인수로 받고 싶으면 이렇게 하기
return (a + b) / 2.; }
template <class T>
T* new_con(int size) { //파라미터로 템플릿 타입을 쓰지 않는 경우도 가능
return new T[size]; }
template <class T, int N> //비템플릿 변수도 올 수 있어요
T scale(const T& value) {
return value * N; }
int main() {
auto k = aver(2, 2.3);
auto* h = new_con<int>(10); //이렇게 템플릿 타입을 <>으로 정해준다!
auto g = scale<int, 4>(10); }
template <class T>
class Point {
public:
T x, T y;
Point(T x=, T y) : x(x), y(y) {};
void Print(); };
template <class T>
void Point<T>::Print() { //메서드를 외부에서 선언하는 법!
std::cout << x << y << std::endl; }
template <>
void Point<int>::Print() {
//특정 타입을 템플릿 타입으로 받았을 경우만 얘로 작동하게 할 수 있다!
//specialization (저격)
std::cout << x << ':' << y << std::endl; }

#include <iostream> #include <string> #include <complex> #include <sstream> #include <random>
int main() {
//String Objects (저격)
std::string name = "junsik";
auto arr_name = name.c_str(); //C string을 반환
name += "um";
if (name.length() == name.size()) {
std::cout << name.find('i') << std::endl;
}
if (!(name.empty())) {
auto name2 = name.substr(0, 6);
name[1] = 'a';
std::cout << name2 << std::endl;
}
//Complex Numbers (저격)
std::complex<double> dc(2., 3.);
//complex는 템플릿 클래스입니다
double re = dc.real();
double im = dc.imag();
std::cout << abs(dc) << arg(dc) << norm(dc) << conj(dc).imag() << std::endl;
//규모, 위상, 크기, 컬레복소수
//String stream (저격)
//문자열 안에 다른 타입의 값을 넣거나 문자열로 만들기
std::stringstream ss;
for (int i = 0; i < 10; i++) {
ss << i << ", ";
} //이, 을 ss에 넣어주기
std::cout << ss.str() << std::endl;
//stringstream 타입을 c++ string 타입으로 반환
std::string tmp, str = "FUCK YOU";
std::stringstream ss1(str);
while (ss1 >> tmp) { //스페이스와 엔터 기준으로 문자열 나누기
std::cout << tmp << std::endl;
}
char a, b, c, d, e, f, g;
ss >> a >> b >> c >> d >> e >> f >> g;
std::cout << a << b << c << d << e << f << g << std::endl; //문자열 뽑아내기
//Better Pseudorandom Number Gen. (저격)
//rand보다 더 나은 난수 생성법
std::random_device rdev; //랜덤 변수 시드
std::mt19937 mt(rdev()); //메르센 트위스터 제너레이터
std::uniform_int_distribution<int> dist1(0, 99); //정수 분포 [0~99]
std::uniform_real_distribution<double> dist2(0., 99.); //실수 분포 [0~99)
std::normal_distribution<double> dist3(50., 10.);
//가우시안 정규 분포 평균 50, 표준편차 10
int arr[100] = { 0 };
for (int i = 0; i < 1000000; i++) {
int r = dist1(mt);
arr[r]++;
} //인덱스를 0~99의 난수로 하여 무지성 i가 박히도록 하기
}
#include <iostream> #include <vector>
class C1 {
int x, y;
public:
static double pi; //이 클래스에 대한 모든 객체가 같은 값을 공유하는 static 멤버
C1() : x(0), y(0) {} //default 생성자
C1(int x, int y) : x(x), y(y) {} //생성자 오버로드
C1(const C1& c) : x(c.x) {} //카피 생성자 (기본 제공)
//C1(const C1&) = delete;를 쓰면 카피 생성자 컬 가능
//각종 연산자 오버로드 (저격)
C1& operator= (const C1& c) { //카피 연산자 (기본 제공)
x = c.x;
return *this;
}
const C1& operator++() { //전위증가 오버로드
x += 1; y += 1;
return *this;
}
const C1& operator ++(int) { //후위증가 오버로드
auto k = *this;
x += 1; y += 1;
return k;
}
operator double() const { //캐스팅 오버로드
return static_cast<double>(x), static_cast<double>(y); }
C1 operator+(const C1& c) const { //연산자 오버로드
//파라미터로 객체를 쓰고 싶으면 레퍼런스로, 수정 여부는 const로
//const 메소드는 멤버 수정불가
//const(객체, 메소드)는 오직 const(메소드)만 호출 가능 (오직 호출 관계에 대해서만 그런거임!)
return C1(this->x + c.x, this->y + c.y);
} //this는 오직 메소드 내에서만 사용하며 해당 메소드를 호출한 객체의 주소를 의미
//-> 연산자는 객체의 주소와 멤버의 관계
int getX() const; //class 외부 선언 함수는 프로토타입 작성 필요
friend std::ostream& operator<<(std::ostream& os, const C1& c);
//friend 함수의 선언
friend class C2;
//friend 클래스의 선언
};
//각종 연산자 오버로드 (저격)
std::ostream& operator<<(std::ostream& os, const C1& c) {
os << c.x << ", " << c.y;
return os; }
int C1::getX() const { //::연산자를 통해 범위를 지정해주면 외부에서도 메소드 정의 가능
return x; }
double C1::pi = 3.141592; //단 static 멤버의 정의는 전역에서 해줘야 한다
int main() {
C1 a(1, 5); C1 b(4, 6);
std::cout << a + b << std::endl;
//Pointers to Object & Obejects Arrays
C1* p = &a; //pointer to object
p = new C1(2, 3); //dyanmic object
delete p;
//std::vector<C1>(10); //C1의 default 생성자가 없으면 불가능한 문장
std::vector<C1*> v(10);
//그래서 컨테이너에는 객체의 포인터를 엘리먼트로 쓰도록 한다.
int k = 0;
for (auto iter = v.begin(); iter != v.end(); iter++) { *iter = new C1(k, k + 1); k++;
} //엘리먼트의 정의는 동적으로 해주면 굿.
}

#include <iostream>
class B {
int x;
protected: //상속 타입 중 하나
public:
B(int x) : x(x) {}
virtual ~B() {}
//소멸자는 먼저 파생이 소멸되고, 부모가 소멸되는 순서가안정적이기에,
//부모 소멸자에는 virtual을 써주는 것이 좋다.
int getX() return x;
void print() {
std::cout << x << std::endl;
}
virtual void vprint() { //가상 함수의 정의
std::cout << x << std::endl;
}
};
struct D : public B { //B와D는 상속 관계
//struct은 private 영역이 없는 클래스
int x;
D(int x, int y) : x(x), B(y) {}
//생성자를 통해 파생 클래스가 써먹을 베이스 클래스의 객체를 생성
//이렇게 안해주면 default 생성자로 생성
~D() {}
void print() { //오버라이드
std::cout << B::getX() << ", " << this->x << std::endl;
}
virtual void vprint() override { //가상 함수의 오버라이드는 명시해줘야 함
std::cout << B::getX() << ", " << this->x << std::endl;
}
};
void Print(B& b) { //가상 함수의 진가가 나오는 상황
//만약 Print가 D 타입을 인수로 받는다면, 알짤딱깔센으로 적절한 vprint를 쓸 것이다.
b.vprint();
}
class AC {
//순수 가상 함수를 포함하는 클래스는 추상클래스로, 객체를 만들 수 없다.
int x;
public:
AC(int x) : x(x) {}
virtual void print() = 0; //순수 가상 함수의 선언. 즉 정의하지 않은 가상 함수.
};
int main() {
B b1(1);
D d1(1, 2);
B b2 = d1;
//D& d2 = b2;
//부모와 파생 중, 어느쪽이 더 데이터를 많이 가지고 있을까?
//Virtual Funciton (= Polymorphism) (저격)
//가상 함수는 파생 클래스가 부모 클래스의 레퍼런스나 포인터에 할당될때 진가를 발휘함
B& r1 = b1;
B& r2 = d1;
B* p1 = &b1;
B* p2 = &d1;
r1.print(); //당연히 B의 print가 호출됨
r2.print(); //그런데 얘도 B의 print가 호출됨 (feat. 생성자에서 생성한 B의 객체)
p1->print(); //즉 비가상 함수들은 명시된 타입을 중시함을 볼 수 있음
p2->print();
//하지만 가상 함수의 경우
r1.vprint(); //B의 vprint가 호출되고
r2.vprint(); //D의 vprint가 호출됨!!!
p1->vprint(); //즉 가상 함수들은 실제의 타입을 중시함을 볼 수 있음
p2->vprint();
r2.B::vprint(); //심지어 가상 함수더라도 ::으로 부모클래스의 것도 호출 가능함
Print(b1);
Print(d1);
//Virtual destructor
B* b3 = new D(1, 2);
delete b3; //위에 맥락과 동일하다고 보면 된다
//Dynamic Cast //virtual을 하나라도 포함되어있는 객체에 대해서만 사용 가능
//캐스팅에 실패하면 nullptr이나 exception으로 빠짐
B* p3 = new B(10);
D* p4 = dynamic_cast<D*>(p3); //B < D이므로 실패
D* p5 = new D(20, 30);
B* p6 = dynamic_cast<B*>(p5); //성공
B* p7 = new D(20, 30);
D* p8 = dynamic_cast<D*>(p7); //성공!! }

#include <algorithm> #include <numeric>
//iota(반복자, 반복자, 초기값) - 초기값부터 차례대로 1씩 증가하도록 범위내 원소 변경
//find(반복자, 반복자, 목표값) - 범위 내에 첫번째 목표값의 위치 반환
//find_if(반복자, 반복자, 조건) - 범위 내에 조건을 만족하는(인수로 들어가 함수가 참을 반환하게 하는) 첫번째 원소 위치 반환
//count(반복자, 반복자, 목표값) - 범위 내에 목표값의 개수 반환
//count_if(반복자, 반복자, 조건) - 범위 내에 조건 만족 값 개수 반환
//copy(반복자1, 반복자1, 반복자2) - 범위1만큼 위치2에 복사
//transform(반복자1, 반복자1, (반복자2), 반복자3, 함수) - 범위1(과 범위2)의 원소(쌍)을 함수로 돌려나온 반환값을 위치3에 복사
//붙여넣기 대상이 std::ostream_iterator<int>(std::cout, " ")이라면 구분자와 함께 복사된 데이터가 출력된다.
//generate(반복자, 반복자, 함수) - 범위 내 원소를 인수로 하여 돌린 함수의 반환값을 범위에 복사
//accumulate(반복자, 반복자, 초기값) - 초기값 + 범위 내 원소 반환
//partition(반복자, 반복자, 조건) - 범위 내 조건을 만족하는 원소들을 앞으로 이동 (순서는 무작위)
//merge(반복자1, 반복자1, 반복자2, 반복자2, 반복자3) - 범위1과 범위2 합치고 위치3에 복사
//remove(반복자, 반복자, 목표값) - 범위 내에서 목표값을 지운 결과를 범위 맨 앞에 복사하고 복사한 곳 그 다음 위치를 반환
//remove_if(반복자, 반복자, 조건) - 범위 내에서 조건을 만족하는 원소를 지운 결과를 범위 맨 앞에 복사하고 복사한 곳 그 다음 위치를 반환
//all_of(반복자, 반복자, 조건) - 범위 내 모든 원소가 조건을 만족하는가?
//any_of(반복자, 반복자, 조건) - 범위 내 원소 중 하나라도 만족하는가?
//none_of(반복자, 반복자, 조건) - 범위 내 모든 원소가조건을 만족하지 않는가?
//distance(반복자, 반복자) - 범위의 크기를 반환.
//find_end(범위자1, 반복자1, 반복자2, 반복자2, (조건)) - 범위1 내에서 범위2의 원소가 등장하는 마지막 위치 반환
//for_each(반복자, 반복자, 함수) - 범위 내 원소를 인수로 하여 차례대로 함수 실행, 단 원소 수정 불가 및 함수 반환값 무시
//reverse(반복자, 반복자) - 범위 내 원소의 순서 반대로 해서 범위에 복사
//sort(반복자, 반복자, 조건) - 범위 내 원소를 양 옆 원소끼리 조건에 맞게 정렬
//erase(반복자, 반복자) - 범위 내 원소 삭제
#include <iostream> #include <set>
int main() {
std::set<int> x;
x.insert(10);
x.insert(10); //씹힌다
x.insert(11);
x.insert(9);
}
#include <iostream> #include <queue> #include <vector>
struct Cmp {
bool operator()(int x, int y) {
return x > y; //내림차순 우선순위 });
int main() {
std::priority_queue<int, std::vector<int>, Cmp> queue;
//자료형, 컨테이너, 비교함수 순서로 작성
queue.push(11);
queue.push(7);
while (!queue.empty()) {
std::cout << queue.top() << ' '; //가장 우선 순위가 높은 놈 반환
queue.pop(); //우선 순위 높은 놈 킬 }}
//랄다함수
#include <iostream> #include <vector> #include <algorithm> #include <functional>
int main() {
std::cout << [](int x, int y) -> int {return x + y; } (3, 2) << std::endl;
//간이 함수를 정의하고 인수를 3, 2로 하여 사용.
auto F1 = [](auto a, auto b, auto c) {return a + b + c; };
//간이 함수를 정의하여 F1에 할당.
std::cout << F1(10, 20, 30) << std::endl;
//인수를 10, 20, 30으로 하여 반환값 얻기.
std::vector<int> v = { 1,2,3,4,5 };
std::for_each(v.begin(), v.end(), [&sum](int element) {sum += element; });
//외부 변수 sum을 레퍼런스로 사용하여 모든 원소에 대해서 sum에 더하기, 즉 원소의 총합.
}
std::function<double(double)> derivative(std::function<double(double)> f, double h) {
return [f, h](double x) { return (f(x + h) - f(x)) / h; };
}
//인수를 받아 호출 가능한 자료형은 전부 std::function<리턴 타입(인수 타입)>으로 나타낼 수 있다.
//double형을 반환하고 double형을 인수로 받는 호출 가능한 자료형을 반환하는 함수 derivative는 //반환하는 자료형과 동일한 타입인 f와 double형 h를 파라미터로 한다.
//derivative는 간이 함수를 반환한다. 간이 함수는 외부 자료 f와 h를 카피로 사용하며,
//파라미터 x에 대해 미분꼴을 계산한 결과를 반환한다.

class MyException : public std::exception {public: const char* what() const noexcept override
{ return "My custom exception"; }};
int main() { try { if (errorOccured) { throw std::runtime_error("Something went wrong!"); }
} catch (const std::runtime_error& e) { std::cout << "Caught an exception: " << e.what()
<< std::endl; }

#include <iostream> #include <vector>
class MyIterator {
std::vector<int>::iterator current;
public:
MyIterator(std::vector<int>::iterator it) : current(it) {}
// Dereference operator
int& operator*() const {
return *current;
}
// Pre-increment operator
MyIterator& operator++() {
++current;
return *this;
}
// Post-increment operator
MyIterator operator++(int) {
MyIterator temp = *this;
++current;
return temp;
}
// Equality operator
bool operator==(const MyIterator& other) const {
return current == other.current;
}
// Inequality operator
bool operator!=(const MyIterator& other) const {
return current != other.current;
}
};

class MyCollection {
std::vector<int> data;
public:
MyCollection(std::initializer_list<int> list) : data(list) {}
MyIterator begin() {
return MyIterator(data.begin());
}
MyIterator end() {
return MyIterator(data.end());
}
int& operator[](std::size_t index) {
return data[index];
}
std::size_t size() const {
return data.size();
}
void print() const {
for (const auto& elem : data) {
std::cout << elem << " ";
}
std::cout << std::endl;
}
};

using namespace std;
namespace another{
int max(const std::vector<int>& vec) {
auto p = std::begin(vec); int m = *p++;
while (p != std::end(vec)) {
if (*p > m) m = *p;
p++;
}
int count = 0;
for (auto elem : vec) if (elem == m) count++;
return count; }}

#include <iostream>
void print(int a[], int n) {
//정적 배열을 파라미터로 쓰는 경우,
//배열의 주소와 사이즈 정보 파라미터 두가지를 보내야 한다.
//이렇게 a[]라고 쓴 것은 int* a와 다를 없다.
for (int i = 0; i < n; i++) std::cout << a[i] << ", ";
std::cout << std::endl;
}
void recursive_print(int* a, int n) {
//정적 배열을 파라미터로 쓸땐 주소와 크기 두가지 데이터면 충분,
//이렇게 recursion을 사용해서 print를 만들 수도 있다.
if (n > 0) {
std::cout << *a << ", ";
recursive_print(a + 1, n - 1);
}
}
//아니면 정적 배열이여도 반복자는 지원하니까 begin end 사용하자.
void multi_print(int(* a)[2], int x, int y) {
//다차원 정적 배열을 출력하기.
//첫번째 엘리먼트의 타입은 크기가 2인 정적 배열이므로 int (*a)[2]이 적절한 파라미터.
for (int i = 0; i < x; i++) {
for (int j = 0; j < y; j++) std::cout << a[i][j] << ", ";
std::cout << std::endl;
}
}
int main() {
//Static Arrays
int a1[3]; //엘리먼트는 전부 쓰레기값
int a2[5] = { 1,2,3 }; //나머지 엘리먼트는 0으로 초기화
int a3[] = { 1,2,3 }; //자동 크기 3
print(a2, sizeof(a2) / sizeof(int));
//인수는 단순히 배열 이름(주소)과 사이즈(배열의 메모리 크기/엘리먼트 타입의 메모리 크기)
std::cout << a2 << std::endl;
//정적 배열의 이름은 첫번째 엘리먼트를 가리키는 포인터
//(산수 계산 시 가리키는 엘리먼트가 바뀌조, 주소는 엘리먼트 데이터 타입만큼 바뀌고)
//연산자 꼬기 (저격)
auto p = a2;
std::cout << *p++ << std::endl;
//*(a2++) = *(a2) = 1, 이후 p는 2를 가리킴. (주소 값 증가)
std::cout << (*p)++ << std::endl;
//(*p)++ = 2++ = 2, 이후 2는 3이 됨. (엘리먼트 값 증가)
/*
int* a[3] -> std::array<int>[3]* a
int (*a)[3] -> std::array<int, 3>* a
int** p[3] -> std::array<int*, 3>* a
*/
//Dynamic Arrays (저격)
auto a4 = new int[3]; //new elemType[size]
delete[] a4;
//Copying Arrays
int a[10], b[10];
for (int i = 0; i < 10; i++) a[i] = i;
//b = a; //정적 배열의 첫번째 엘리먼트 주소는 수정 불가능하다.
int* c;
c = a; //c는 사실상 a의 레퍼런스처럼 된다.
auto d = new int[10];
//d = a; //b와 동일한 원리로 불가능하다.
for (int i = 0; i < 10; i++) d[i] = a[i];
//이렇게 쓰면 복사지요.
//Multidimensional Arrays (저격)
int a5[3][2] = { {1}, {21, 22}, {31, 32} };
multi_print(a5, 3, 2); //a5는 a5의 첫번째 엘리먼트({1})의 주소.
int* p1 = new int[3]; //크기가 3인 엘리먼트가 int인 배열을 동적 메모리에 생성. 3은 dynamic.
delete[] p1;
int(*p2)[3] = new int[2][3];
delete[] p2;
//크기가 3인 엘리먼트가 int인 배열을 엘리먼트로 하는 크기가 2인 배열을 동적 메모리에 생성
//그건 크기가 3인 엘리먼트가 int인 배열을 가리키는 포인터 p2가 관장.
//즉 3은 static이지만 2는 dynamic.
int** p3 = new int* [2];
for (int i = 0; i < 2; i++) {
p3[i] = new int[3];
}
for (int i = 0; i < 2; i++) {
delete[] p3[i];
}
delete[] p3;
//크기가 2인 엘리먼트가 int의 포인터인 배열의 포인터가 p3다.
//이 방법이 바로 다차원 배열의 모든 요소를 동적으로 할당하는 방법이다!
}
```