**〈C++ 정리본〉**
**객체 지향 프로그래밍**
: 절차가 아닌 데이터를 중심으로 하여 현실의 사물을
객체로 정의하고 잘 만들어진 객체를 조립하여
프로그램을 완성하는 방식
: 데이터와 알고리즘이 묶여 있음
: 함수 등의 구조를 통해 데이터와 알고리즘을
분리 및 구조화
: 대표적으로 C언어가 있음

**절차적 프로그래밍**
: 데이터보다 알고리즘(절차)을 중시

**〈C++ 특징〉**
**C와의 호환성**
: ANSI C와의 호환성 제공
: 따라서 대부분 C 프로그램은 유효한 C++ 프로그램
: 새로운 키워드 추가

**C보다 개선된 기능**
: bool 형과 true, false 키워드
: 엄격한 형 검사
: 변수의 문중 선언
: 레퍼런스
: 동적메모리와 new/delete 연산자
: 네임스페이스
: 함수 오버로딩
: 디폴트 인자
: 인라인 함수
: 객체지향

**객체 지향 프로그래밍 기능**
: 프로그램의 기본 단위로 '객체(object)' 사용
: 캡슐화를 통해 최소한의 정보만으로 객체 사용
: 객체 구현을 위한 클래스(class) 사용 및 클래스 간 상속을 통해
기존 클래스의 기능 이어받음
: 서로 다른 클래스는 같은 방법으로 사용하더라도 다형성 기능을 통해
서로 다른 동작을 수행할 수 있음

**제너릭 프로그래밍 기능**
: 데이터 알고리즘으로부터 분리하여 일반화해서 처리하는 프로그래밍 기법
: 템플릿 기능을 지원
: 데이터형에 무관하게 같은 알고리즘을 처리하는 함수나
클래스를 쉽게 정의 가능

**멀티패러다임 프로그래밍**
패러다임(paradigm)
: 프로그래밍 방법론
프로그램하기 위한 접근 방식 (approach)

**멀티패러다임 프로그래밍**
: 여러 가지 프로그래밍 방법론을 동시에 지원한다는 의미

**성능 중심**
: C++은 '사용상의 편의성'보다는 '성능'에 좀 더 무게를 두고 있다.
: C++은 배우기도 어렵고, 사용 시 주의 사항도 많지만 성능이 우수한
프로그램을 만들 수 있다.

**강력한 사용자 정의 형(User-defined type) 지원 기능**
: C++의 내장 형(built-in type)과 거의 동일한 방법으로 사용될 수 있는
사용자 정의 형을 만들 수 있다.
**〈C++ 응용 분야〉**
: 애플 OS X, 심비안 OS, 마이크로 소프트 Windows XP 등의 운영 체제 개발
: CORBA 등의 미들웨어 개발
: 구글 검색 엔진 등의 검색 엔진 개발
: Photoshop, Illustrator 등의 그래픽 편집 프로그램 개발
: 그래픽 엔진이나 CAD 툴 개발
: Doom, StarCraft 등의 게임 개발
: 네트워크, 통신 장비 개발
: 모바일, 임베디드 분야
-〉 성능이나 이식성(portavility) 위주의 응용 분야

**〈C++ 언어의 설계 목적〉**
**C 언어의 문법 체계 계승**
: 소스 레벨 호환성 - 기존에 작성된 C 프로그램을 그대로 가져다 사용
: 링크 레벨 호환성 - C 목적 파일과 라이브러리를 C++ 프로그램에서 링크

**캡슐화, 상속, 다형성**
: 소프트웨어의 재사용을 통해 생산성 향상
: 복잡하고 큰 규모의 소프트웨어의 작성, 관리, 유지보수 용이
: 실행 시간 오류의 가능성을 줄임 / 디버깅 편리

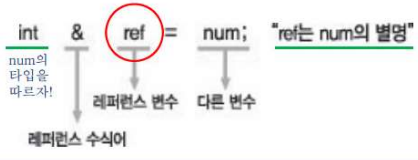
**실행 시간을 저하시키는 요소와 해결**
: 작은 크기의 멤버 함수 잦은 호출 가능성 -〉 인라인 함수로 실행 시간 저하 해소

**〈C언어에 추가한 기능〉**
**함수 중복(function overloading)**
: 매개 변수의 개수나 타입이 다른
동일한 이름의 함수들 선언
**디폴트 매개 변수(default parameter)**
: 매개 변수에 디폴트 값이 전달되도록 함수 선언
**참조와 참조 변수(reference)**
: 하나의 변수에 별명을 사용하는 참조 변수 도입
**참조에 의한 호출(call-by-reference)**
: 함수 호출 시 참조 전달
**new/delete 연산자**
: 동적 메모리 할당/해제를 위해 new와 delete 연산자 도입
**연산자 재정의**
: 기존 C++ 연산자에 새로운 연산 정의
**제네릭 함수와 클래스**
: 데이터 타입에 의존하지 않고 일반화시킨 함수나 클래스 작성 가능

**〈엄격한 형 검사〉**
: C++에서는 함수 호출 전에 반드시 함수 선언 또는 정의가 필요하다.
  void foo( );  // 인자 없는 함수 선언
: 함수 선언 시 리턴형과 인자형을 모두 명시해야 한다.
: C에서는 **암시적(Implicit)**으로 void* 형이 다른 포인터 형으로 형 변환 가능하다.
  (참고, malloc()은 void* 형)
: 그러나, C++에서는 이런 형 변환을 허용하지 않는다.

**〈bool형〉**
: C 방식의 참, 거짓
: C에는 부울 값(Boolean value)을 표현하기 위해서
 정수형(int, 4 bytes)을 사용하는 것이 일반적이다.
: C++의 bool 형
: bool 형은 참(true) 또는 거짓(false)에
 해당하는 값을 저장할 수 있는 크기의 데이터 형
: C++은 true, false 키워드를 제공한다.
: C99 표준 이후 stdbool.h를 통해 bool 타입 지원

**〈레퍼런스〉**
레퍼런스 형은 한마디로 "다른 변수의 별명"이다.
**레퍼런스의 선언**
: 레퍼런스 수식어인 &를 레퍼런스를 선언할 때 지정한다.
: 레퍼런스가 **어떤 변수의 별명**인지,
  레퍼런스를 초기화하면서 지정한다.




: 레퍼런스를 선언할 때 데이터 형은
  레퍼런스 변수가 참조하는
-〉 데이터 형이 일치하지 않으면 컴파일 에러

**레퍼런스(&)는 자신만의 주소를 갖지 않는다.**
: 포인터 변수는 다른 변수를 주소를 저장할, 자신만의 메모리
  (주소 저장을 위한 메모리)를 갖는다.
: 레퍼런스는 다른 변수에 대한 이름만 존재할 뿐, 레퍼런스를 위한
  실질적인 메모리는 할당되지 않는다.
: 그러므로, 레퍼런스의 크기는 레퍼런스가 참조하는 변수의 크기를 따라 결정된다.
: 즉, 포인터는 주소 값을 할당하고, 레퍼런스는 참조 대상을 그대로 할당한다.
  **레퍼런스는 반드시 선언 시 초기화되어야 한다.**
  **레퍼런스는 반드시 변수로 초기화 해야 한다.**
  **즉, 상수로는 초기화 할 수 없다!**

: 레퍼런스의 데이터 형은 레퍼런스가 참조하는 변수의 데이터 형과 일치해야 한다.
: 레퍼런스가 참조하는 대상을 바꿀 수 없다.
: 포인터는 NULL을 허용하지만 레퍼런스는 NULL을 허용하지 않는다.
: 함수의 매개변수 전달 시 레퍼런스를 사용하면 간결해짐
: 레퍼런스에 의한 전달에서는 코드가 간결해진다.
: 함수를 호출할 때도 변수를 직접 넘겨주고 함수 안에서도 레퍼런스를 사용해서,
  직접 함수를 호출한 곳의 변수에 접근할 수 있기 때문
: 포인터에 의한 전달 방법에서는 의도하지 않는 문제가 발생하는 경우가 있다.

**〈네임스페이스〉**
: 네임스페이스란 식별자가 정의되는 공간

**특징**
: 네임스페이스를 이용하면 같은 이름의 식별자를 여러 번 정의하고
  구분해서 사용할 수 있다.
: 네임스페이스 안에 정의된 식별자를 사용할 때는
  **'네임스페이스 이름::식별자'**의 형식으로 사용한다.

네임스페이스의 사용
: using 네임스페이스이름 :: 식별자;
: using namespace 네임스페이스 이름;
: using namespace std;

**〈동적 메모리와 new/delete〉**
: 동적 메모리란 프로그램 실행 중에 메모리의 할당과 해제가 결정되는 메모리

**특징**
: 동적 메모리를 사용하면 실행 중에 꼭 필요한 만큼 메모리를 할당 받아서 사용하므로
  메모리 낭비가 일어나지 않으며, 미리 정해진 크기가 아니라 원하는 크기만큼
  할당 받는 것도 가능
: 동적 메모리를 사용하면 메모리의 할당과 해제 시점을 전적으로 프로그래머가 제어 가능

**동적 메모리를 할당하기 위해서는 new 연산자를 사용한다.**
: new 연산자는 메모리 할당 실패시 널 포인터를 리턴한다.

**동적 메모리를 사용이 끝나면 delete 연산자로 해제한다.**
: delete 연산자는 **포인터가 가리키는 동적 메모리를 해제**하는데,
  이때 주의할 점은 delete 연산자 다음에
  지정된 포인터 변수가 없어지는 것은 아니라는 점이다.
: delete 연산자로 동적 메모리를 해제한 다음에는 동적 메모리의 주소를 저장하는
  **포인터 변수를 널 포인터로 지정하는 것이 안전**하다.
  **동적으로 할당한 메모리를 자동으로 해제되지 않으므로 반드시 프로그래머가**
  **명시적으로 해제해야 한다.**

: new로 할당한 메모리는 delete로 해제하고, new [ ]로 할당한 메모리는 delete [ ]로 해제한다.
: delete 연산자로 이미 해제된 메모리를 다시 해제해서는 안된다.
 해제된 메모리: 이미 반환해버려서 다시 해제가 불가능

**〈C++ 객체 지향 특성〉**
**캡슐화(Encapsulation)**
: 데이터를 캡슐로 싸서 외부의 접근으로부터 보호
: C++에서 클래스(class 키워드)로 캡슐 표현

**클래스와 객체**
: 클래스 - 객체를 만드는 틀 (= 템플릿, 인스턴스화)
: 객체 - 클래스라는 틀에서 생겨난 실체
객체(object), 실체(instance)는 같은 뜻

**객체 지향 상속성(Inheritance)**
: 객체가 자식 클래스의 멤버와 부모 클래스에
선언된 모양 그대로 멤버들을 가지고 탄생

**다형성(Polymorphism)**
: 하나의 기능이 경우에 따라 다르게 보이거나
다르게 작동하는 현상
: 연산자 중복, 함수 중복, 함수 재정의(overriding)

**〈C++언어에서 객체 지향을 도입한 목적〉**
**소프트웨어 생산성 향상**
: 소프트웨어의 생명 주기 단축 문제 해결 필요
: 기 작성된 코드의 재사용 필요
: C++ 클래스 상속 및 객체 재사용으로 해결

**실세계에 대한 쉬운 모델링**
과거의 소프트웨어
: 수학 계산이나 통계 처리에 편리한 절차 지향 언어가 적합
현대의 소프트웨어
: 물체 혹은 객체의 상호 작용에 대한 묘사가 필요
: 실세계는 객체로 구성된 세계
: 객체를 중심으로 하는 객체 지향 언어 적합
**〈링킹〉**
: 목적 파일끼리 합쳐 실행 파일을 만드는 과정
목적 파일은 바로 실행할 수 없음
: 목적 파일과 C++ 표준 라이브러리의 함수 연결, 실행 파일을 만드는 과정
hello.obj + cout 객체 + 〈〈 연산자 함수 => hello.exe를 만듦

**〈C++ 표준 라이브러리〉**
**C라이브러리**
: 기존 C표준 라이브러리를 수용 -〉 c로 시작하는 헤더 파일에 선언됨

**C++ 입출력 라이브러리**
: 콘솔 및 파일 입출력을 위한 라이브러리

**C++ STL 라이브러리**
: 제네릭 프로그래밍을 지원하기 위한 템플릿 라이브러리

**〈주석문과 main( ) 함수〉**
**주석문**
: 여러 줄 주석문 /* … */
: 한 줄 주석문 //

**main( ) 함수**
: int main( ) 표준 모양
: return 0; 생략 가능
**〈#include 〈iostream〉〉**
**#include 〈iostream〉**
: 전처리기 (C++ Preprocessor)에게 내리는 지시
: 〈iostream〉 헤더 파일을 컴파일 전에
소스에 확장하도록 지시

**〈iostream〉 헤더 파일**
: 표준 입출력을 위한 클래스와 객체, 변수 등이 선언됨
: ios, istream, ostream, iostream 클래스 선언
: cout, cin, 〈〈, 〉〉 등 연산자 선언

**〈화면 출력〉**
**std :: cout 〈〈 "Hello";**

**cout 객체**
: 스크린 출력 장치에 연결된 표준 C++ 출력 스트림 객체
: 〈iostream〉 헤더 파일에 선언
: std 이름 공간에 선언 -〉 std :: cout로 사용

**〈〈 연산자**
: 스트림 삽입 연산자
산술 시프트 연산자( 〈〈 )가 재정의됨
: iostream 클래스에 구현됨
: 오른쪽 피연산자를 왼쪽 스트림 객체에 삽입
: cout 객체에 연결된 화면에 출력
: 여러 개의 〈〈 연산자로 여러 값 출력
: 연산식 뿐 아니라 함수 호출도 가능
: '₩n' 이나 std :: endl; 로 다음 줄 이동

**〈네임스페이스〉**
이름(identifier) 충돌이 발생하는 경우
: 여러 명이 서로 나누어 프로젝트 개발하는 경우
: 오픈 소스 / 다른 사람이 작성한 소스나 파일 가져와서 컴파일 하거나 링크하는 경우

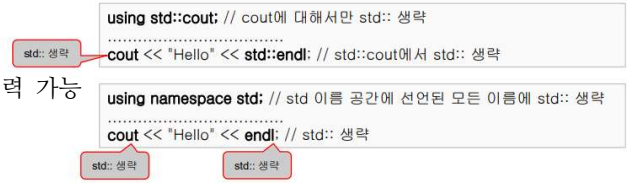
이름 공간 안에 선언된 이름은 다른 이름공간과 별도로 구분됨
```
namespace kitae {              // kitae라는 이름 공간 생성
        // 이 곳에 선언된 모든 이름은 kitae 이름 공간에 생성된 이름
}
```

이름 공간 사용
: 이름 공간 :: 이름

**〈std ::〉**
: C++ 표준에서 정의한 이름 공간(namespace) 중 하나
: std :: 생략 -〉 using 지시어 사용

**〈cin과 〉〉연산자를 이용한 키 입력〉**
**cin**
: 표준 입력 장치인 키보드를 연결하는 입력 스트림 객체

**〉〉 연산자**
: 스트림 추출 연산자
: 연속된 〉〉 연산자를 사용하여 여러 값 입력 가능
```
cout 〈〈 "너비와 높이를 입력하세요";
cin 〉〉 width 〉〉 height;
```




**〈〈Enter〉 키를 칠 때 변수에 값 전달〉**
**cin의 특징**
: 입력 버퍼를 내장
: 〈Enter〉키가 입력될 때까지 입력된 키를 입력 버퍼에 저장
: 도중에 〈Backspace〉 키를 입력하면 입력된 키 삭제

**〉〉 연산자**
: 〈Enter〉 키가 입력되면 비로소 cin의 입력 버퍼에서 키 값을 읽어 변수에 전달

**〈실행문 중간에 변수 선언〉**
장점
: 변수를 사용하기 직전 선언함으로써 변수 이름에 대한 타이핑 오류 줄임

단점
: 선언된 변수는 일괄적으로 보기 힘듦
: 코드 사이에 있는 변수 찾기 어려움

**〈C++ 문자열 표현 방식〉**
C-스트링 문자열 : char name1[6] = { 'G', 'r', 'a', 'c', 'e', '₩0' };
// name1은 문자열 "Grace" -〉 string
단순 문자 배열 : char name1[6] = { 'G', 'r', 'a', 'c', 'e' };
// name2는 단순 문자 배열
=〉 char는 ₩0 없고, string은 ₩0로 초기화

**〈cin.getline( )으로 공백이 낀 문자열 입력〉**
공백이 낀 문자열을 입력 받는 방법
-〉 cin.getline ( char buf[ ], int size, char delimitChar )
: buf에 최대 size -1개의 문자 입력 가능. 끝에 '₩0' 붙임
: delimitChar를 만나면 입력 중단. 끝에 '₩0' 붙임 (디폴트 값은 〈Enter〉키)

**〈string 클래스〉**
: C++ 표준 클래스
: 문자열의 크기에 따른 제약 없음
(string 클래스가 스스로 문자열 크기에 맞게 내부 버퍼 조절)
: 문자열 복사, 비교, 수정 등을 위한 다양한 함수, 연산자 제공
: 객체 지향적
: 〈string〉 헤더 파일에 선언 (#include 〈string〉 필요)

- 문자열 대입 : '=' 연산자 이용
```
string s1;
s1 = "hello world";
```

- 문자열의 길이 구하기 : size / length 멤버 함수 이용
```
string s1 = "abcde";
cout 〈〈 s1.size( );
```
- 문자열 비교 : == 연산자
```
string s1 = "abcde";
string s2 = s1;
if (s1 == s2)
        cout 〈〈 "s1과 s2는 같습니다.₩n";
```
- 공백 문자를 포함한 문자열 입력 : getline 함수 이용
```
string s1;
getline(cin, s1);
```

**〈C++ 헤더 파일 확장자〉**
: 헤더 파일 확장자 없고, std 이름 공간 적시
```
#include 〈iostream〉
using namespace std;
```

| 언어 | 헤더 파일 확장자 | 사례 | 설명 |
|---|---|---|---|
| C | .h | 〈string.h〉 | C/C++ 프로그램에서 사용 가능 |
| C++ | 확장자 없음 | 〈cstrnig〉 | using namespace std; 와 함께 사용해야 함 |

**〈#include 〈헤더파일〉과 #include "헤어파일"의 차이〉**
**#include 〈헤더파일〉**
: '헤더 파일'을 찾는 위치
: 컴파일러가 설치된 폴더에서 찾으라는 지시

**#include "헤더파일"**
: '헤더 파일'을 찾는 위치
: 개발자가 컴파일 옵션으로 지정한
include 폴더에서 찾도록 지시

Q. 〈cstring〉파일에 strcpy() 함수의 코드가 들어 있을까?
(1) strcpy() 함수의 코드가 들어 있다(O, X).
A. 답은 X
(2) strcpy() 함수의 원형이 선언되어 있다(O, X).
A. 답은 O

Q. 그러면 strcpy() 함수의 코드는 어디에 있는가?
A. strcpy() 함수의 코드는 컴파일된 바이너리 코드로,
비주얼 스튜디오가 설치된 lib 폴더에 libcmt.lib 파일에 들어 있고
링크 시에 strcpy() 함수의 코드가 exe에 들어간다.

Q. 그러면 헤더 파일은 왜 사용되는가?
A. 사용자 프로그램에서 strcpy() 함수를 호출하는 구문이 정확한지
확인하기 위해 컴파일러에 의해 필요

**〈객체〉**
사전적 의미 : 물건 또는 대상
객체지향 프로그래밍 : 객체 중심의 프로그래밍

**객체 지향 프로그래밍**
: 관련된 함수와 변수를 묶어서 '객체(object)'를 만들고, 객체 단위로 프로그래밍하기 위한 방법

**인터페이스(interface)**
: 객체의 서비스에 대해서 미리 약속된 부분

**객체는 멤버 변수와 멤버 함수로 구성**
: 객체는 상태(state)와 행동(behavior)으로 구성

TV 객체 예시)
상태
: on/off 속성 : 현재 작동 중인지 표시
: 채널 : 현재 방송중인 채널
: 음량 : 현재 출력되는 소리 크기

행동
: 켜기 (power on)
: 끄기 (power off)
: 채널 증가

**〈클래스 만들기〉**
: 멤버 변수와 멤버 함수로 구성 멤버 변수 = 객체의 속성을 나타낸다.
: 클래스 선언부와 구현부로 구성 멤버 함수 = 객체의 동작을 나타낸다.

**클래스 선언부**
: class 키워드를 이용해 선언
: 멤버 변수는 클래스 선언 내에서 초기화할 수 없음
: 멤버 함수는 원형 형태로 선언
  멤버에 대한 접근 권한 지정 =〉 private, public, protected 중의 하나
: 디폴트는 private
: public : 다른 모든 클래스나 객체에서 멤버의 접근 가능

**클래스 구현부**
: 클래스에 정의된 모든 멤버 함수 구현

**〈구조체와 클래스의 유일한 차이점〉**
: 디폴트 액세스 지정자 뿐인데 구조체는 public이고 클래스는 private이다.

**〈생성자〉**
: 객체가 생성되는 시점에서 자동으로 호출되는 멤버 함수
: 클래스 이름과 동일한 멤버 함수
: 같은 이름으로 함수 선언 가능!
  달라야 하는 점 -〉 인자의 개수, 인자의 데이터 타입
  =〉 생성자의 특징 X, 함수의 특징으로 들어감

**특징**
**생성자의 목적**
: 객체가 생성될 때 객체가 필요한 초기화를 위해
- 멤버 변수 값 초기화, 메모리 할당,
  파일 열기, 네트워크 연결 등

**생성자 이름**
: 반드시 클래스 이름과 동일 (대소문자까지)

**생성자는 리턴 타입**
: 리턴 타입 없음. void 타입도 X

**객체 생성 시 호출은 오직 한 번만 = 함수 오버로딩이 가능**
: 자동으로 호출됨. 임의로 호출할 수 없음.
  각 객체마다 생성자 실행

**생성자 중복**
: 생성자는 한 클래스 내에 여러 개 가능
: 중복된 생성자 중 하나만 실행

**생성자가 선언되어 있지 않으면 기본 생성자가 자동으로 생성됨**
• 기본 생성자 - 매개 변수 없는 생성자
: 컴파일러에 의해 자동 생성

**기본 생성자**
: 클래스에 생성자가 하나도 선언되어 있지 않은 경우, 컴파일러가 대신 삽입해주는 생성자
: 매개 변수 없는 생성자
: 디폴트 생성자

** 함수와 변수는 쓰는 공간이 다르다.
변수는 선언과 동시에 메모리 공간이 생기지만,
함수는 선언은 되도 메모리 공간이 생기지 않는다.
**〈생성자가 다른 생성자 호출(위임 생성자)〉**
**여러 생성자에 중복 작성된 코드의 간소화**
타겟 생성자
: 객체 초기화를 전담하는 생성자

위임 생성자
: 타겟 생성자를 호출하는 생성자,
객체 초기화를 타겟 생성자에 위임

**〈소멸자 (파괴자)〉**
: 객체가 소멸되는 시점에서 자동으로 호출되는 함수
: 객체가 생성되기 전의 상태로 정리하는 함수
- 오직 한 번만 자동 호출, 임의로 호출할 수 없음 - 객체 메모리 소멸 직전 호출됨

```
class Circle {
   Circle();
   Circle(int r);
   ..............
   ~Circle();
};
Circle::~Circle() {
   ..............
}
```

**특징**
**소멸자의 목적**
: 객체가 사라질 때 마무리 작업을 위함
: 실행 도중 동적으로 할당 받은 메모리 해제, 파일 저장 닫기, 네트워크 닫기 등

**소멸자 함수의 이름은 클래스 이름 앞에 ~를 붙인다.**
예) Circle::~Circle() { ... }

**소멸자는 리턴 타입이 없고, 어떤 값도 리턴하면 안됨**
: 리턴 타입 선언 불가
**중복 불가능**
: 소멸자는 한 클래스 내에 오직 한 개만 작성 가능
: 소멸자는 매개 변수 없는 함수

**소멸자가 선언되어 있지 않으면 기본 소멸자가 자동 생성**
: 컴파일러에 의해 기본 소멸자 코드 생성
: 컴파일러가 생성한 기본 소멸자 : 아무 것도 하지 않고 단순 리턴

**〈생성자/소멸자 실행 순서〉**
**객체가 선언된 위치에 따른 분류**
지역 객체
: 함수 내에 선언된 객체로서, 함수가 종료하면 소멸된다.
전역 객체
: 함수의 바깥에 선언된 객체로서, 프로그램이 종료할 때 소멸된다.

**객체 생성 순서**
: 전역 객체는 프로그램에 선언된 순서로 생성
: 지역 객체는 함수가 호출되는 순간에 순서대로 생성

**객체 소멸 순서**
: 함수가 종료하면, 지역 객체가 생성된 순서의
  역순으로 소멸
: 프로그램이 종료하면, 전역 객체가 생성된 순서의 역순으로 소멸

**new를 이용하여 동적으로 생성된 객체의 경우**
: new를 실행하는 순간 객체 생성
: delete 연산자를 실행할 때 객체 소멸
**〈생성자/소멸자 특징〉**
: 컴파일러에 의해 자동 호출되므로 이름이 정해져 있다.
: 생성과 소멸은 동작일 뿐 리턴값이 없고, void라고 적지도 않는다.
: 생성자는 오버로딩 할 수 있지만, 소멸자는 오버로딩이 불가능하다.
: 개발자가 정의하지 않으면 컴파일러가 아무것도 하지 않는 생성자와 소멸자를 만든다.

**〈정보 은폐〉**
캡슐화 방법
: 멤버값을 대신 읽거나 쓰는 액세서 제공

액세서(접근자) 함수는 Get, Set으로 시작하며
대상 멤버 변수를 대신 읽고 쓰는 역할

**〈접근 지정자〉**
멤버에 대한 3가지 접근 지정자
public
: 모든 다른 클래스에 허용

private
: 동일한 클래스의
  멤버 함수에만 제한함

protected
: 클래스 자신과 상속받은
  자식 클래스에만 허용

| 접근 지정자 | 해당 클래스의<br>멤버 함수에서의 접근 | 파생 클래스의<br>멤버 함수에서의 접근 | 클래스 밖에서의 접근 |
|---|---|---|---|
| private 멤버 | O | X | X |
| protected 멤버 | O | O | X |
| public 멤버 | O | O | O |

**〈접근자 함수의 장점〉**
접근자 함수를 사용하면 멤버 변수의 값을 변경하거나
읽어 올 때 값의 유효성을 검사할 수 있다.

**〈멤버의 접근 권한〉**
: 클래스의 멤버 변수는 보통 : private / protected 멤버로 정의
  -〉 잘못된 변경으로부터 멤버 변수를 보호하기 위해 지정
: 클래스의 멤버 함수는 보통 : public 멤버로 정의
  -〉 멤버 함수는 객체가 할 수 있는 일이므로 클래스 밖에서 접근할 수 있도록
  public으로 지정

**〈함수의 장점〉**
: 코드가 중복되지 않고 간결해진다.
: 코드의 재사용성이 증가된다.
: 프로그램의 기능을 함수 단위로 묶었기 때문에 코드 수정이 쉽다.

**〈인라인 함수〉**
: 함수 호출 시 발생하는 오버헤드를 줄이기 위해서 함수를 호출하는 대신
함수가 호출되는 곳마다 함수의 코드를 복사하여 넣어주는 방법

**인라인 함수의 정의**
: 함수의 선언이나 정의에 inline 키워드를 지정하면 된다.
: 헤더파일에 포함되어야 한다.

**인라인 함수의 목적**
: C++ 프로그램 실행 속도 향상
: 자주 호출되는 짧은 코드의 함수 호출에 대한 시간 소모 ↓
: 짧은 코드의 멤버 함수가 많기 때문

**매크로 함수와 인라인 함수의 차이점**
: 매크로 함수 -〉 선행처리기에 의한 문자열 대치 방식
: 인라인 함수 -〉 컴파일러에 의한 코드 대치 방식

**매크로 함수의 문제점**
: 연산자 우선 순위 문제
: 인자의 형 검사를 하지 않는 문제

**인라인 함수, 자동 인라인 함수 장단점**
**장점**
: 프로그램의 실행 시간 빨라짐
**단점**
: 인라인 함수 코드의 삽입으로 컴파일된 전체 코드 크기 증가
: 통계적으로 30% 증가
: 짧은 코드의 함수를 인라인으로 선언하는 것이 좋음

**자동 인라인 함수**
: 클래스 선언부에 구현된 멤버 함수
: 컴파일러에 의해 자동으로 인라인 처리

**〈디폴트 인자〉**
: 함수를 호출할 때 인자를 생략하면 디폴트 값이 자동적으로 사용되도록 하는 방법
: =과 함께 디폴트 값을 지정 -〉 모든 처리는 컴파일러가 해 줌
void foo ( char c, int I, double d **= 0.5** )

**디폴트 인자를 지정할 때 규칙**
: 디폴트 인자를 함수의 가장 오른쪽 인자부터 지정

**디폴트 인자를 사용할 때 규칙**
: 함수의 가장 오른쪽 인자부터 생략해야 한다.

**〈함수 오버로딩〉**
: 이름은 같지만 인자의 개수나 데이터 형이 다른 함수를 여러 번 정의할 수 있는 기능
: 같은 이름을 갖는 함수를 여러 번 정의할 수 있다.
- 각각의 함수는 인자의 시그니처(인자의 개수나 인자의 데이터형)가
반드시 달라야 한다.
- 함수 호출 시 전달된 인자의 데이터 형으로 컴파일러가
어떤 함수를 호출할 지를 결정한다

**특징**
: 인자의 이름만 다른 경우 오버로딩할 수 없다.
: 함수의 리턴형만 다른 경우 오버로딩할 수 없다.
: 데이터 형과 해당 형의 레퍼런스 형으로 오버로딩된 경우 오버로딩할 수 없다.
: typedef로 정의된 데이터 형에 대해 오버로딩된 경우 오버로딩할 수 없다.
: 디폴트 인자에 의해서 인자의 개수가 같은 경우 오버로딩할 수 없다.

**〈함수 템플릿〉**
: 특정 데이터 형을 사용하는 대신 어떤 데이터 형이든 될 수 있는
범용형으로 정의된 함수

**특징**
: 함수 템플릿은 직접 함수를 정의하는 대신 컴파일러에게
함수를 정의하는 방법을 알려준다.
: 컴파일러는 함수가 호출될 때 사용된 인자의 데이터 형을 검사하여
함수 템플릿으로부터 함수의 정의를 자동으로 생성한다.
-〉 함수 템플릿의 인스턴스화
: 함수 템플릿은 처리할 데이터의 데이터 형은 다양하지만,
처리 알고리즘은 동일한 함수를 만들 때 유용

**〈C++ 구조체〉**
**클래스와 유일하게 다른 점**
구조체의 디폴트 접근 지정 - public
클래스의 디폴트 접근 지정 - private