세상을 먹어 치우는 소프트웨어

- 미국의 대형 서점 BORDERS가 아마존에 대패하면서 문을 닫게 됨

- 대형 비디오 대여 체인 Blockbuster가 넷플릭스에(Neflix)에 대패하면서 문을 닫음

-> 엄청난 소프트웨어 인력을 필요로 하는 세상이 왔음

프로그래밍 언어

기계어(machine language)

- 0과 1의 이진수로 구성된 언어

- 컴퓨터의 CPU는 본질적으로 기계어만 처리 가능

어셈블리어

- 기계어의 명령을 ADD, SUB, MOVE 등 과 같은 상징적인 니모닉 기호(mnemonic symbol)로 일대일 대응시킨 언어

- 어셈블러: 어셈블리어 프로그램을 기계어 코드로 번환

고급언어

- 사람이 이해하기 쉽고 복잡한 작업, 자료 구조, 알고리즘을 표현하기 위해 고안된 언어

- Pascal, Basic, C, C++, Java, C#

- 컴파일러: 고급 언어로 작성된 프로그램을 기계어 코드로 변환

언어에 따라서 인터프리터를 사용하기도 함

CPU 🡨🡪 기계어 🡨(어셈블러)🡪 어셈블리어 🡨(컴파일러)🡪 C++ 🡨🡪 개발자

어셈블리어 🡪 BCPL 🡪 B 🡪 C 🡪 C++ 🡪 Java 🡪 C# 순으로 발전

C++: C언어에 객체지향을 더한 형태

C++ 언어의 설계 목적

엄격한 타입 체크

- 실행 시간 오류의 가능성을 줄임

- 디버깅 편리

실행 시간의 효율성 저하 최소화

- 실행 시간을 저하시키는 요소를 최소화 함

- 인라인 함수로 실행 시간 저하 해소

🡪 작은 크기의 멤버 함수 잦은 호출 가능성이 있는 함수 대상으로 함

C언어와의 호환성

- C 언어의 문법 체계 계승하고자 함

🡪 소스 레벨 호환성

🡪 링크 레벨 호환성

객체 지향 개념 도입

- 캡슐화, 상속, 다형성

C++ 객체 지향 특성

캡슐화

- 데이터에 대한 보안, 보호, 외부 접근 제한 하고자 함

- 클래스: 데이터를 보호하기 위한 틀

- 객체 : 클래스로부터 생성되는 실체

상속성

- 자식이 부모의 유전자를 물려받는 것처럼 객체가 부모 클래스에 선언된 모양을 상속 받음

-클래스간 상속 관계 지정

다형성

- 하나의 기능이 경우에 따라 다르게 보이거나 다르게 작동하는 현상

- 연산자 중복, 함수 중복, 함수 오버라이딩의 방법으로 다형성을 구현

C++ 프로그램 개발 과정

1. C++ 프로그램 작성 및 편집

- C++ 소스 프로그램은 텍스트 파일

🡪 아무 텍스트 편집기로 편집 가능

- C++ 소스 프로그램의 표준 확장자는 .cpp

- C++ 통합 개발 이용 소프트웨어 이용 추천

🡪 C++ 소스 편집, 컴파일, 링킹, 실행, 디버깅 등 모든 단계 통합 지원

🡪 대표적인 소프트웨어 – Visual Studio C++ 10.0

2. 컴파일

- C++ 소스 프로 그램을 기계어를 가진 목적 파일로 변환

🡪 cpp 파일을 obj 파일로 변환

3. 링킹

- 목적 파일 끼리 합쳐 실행 파일을 만드는 과정

🡪 목적 파일은 바로 실행할 수 없음

- 목적 파일과 C++ 표준라이브러리의 함수 연결, 실행 파일을 만드는 과정

4. 실행

- 실행 파일은 독립적으로 바로 실행 가능

- 실행 중에 발생하는 오류

🡪 원하는 결과가 나오지 않거나 실행 중에 프로그램의 비정상 종료

5. 디버깅

- 실행 중에 발생한 오류를 찾는 과정

- 디버거

🡪디버깅을 도와주는 프로그램

🡪컴파일러를 만드는 회사에서 함께 공급

- 소스 레벨 디버깅

🡪 C++ 소스를 한 라인씩 실행하고 변수 값의 변화를 보면서 오류 발견

🡪 Visual Studio는 소스 레벨 디버깅 지원

C++ 기본 요소

기본 구성

|  |
| --- |
| #include <iostream> 🡪 실행을 위한 헤더  Int main() { 🡪 실행 시작점의 기본 형식  std::cout << ”Hello\n” 🡪 출력  return 0;  } |

주석문

- 개발자가 자유롭게 붙인 특이 사항의 메모 또는 프로그램에 대한 설명

- 프로그램의 실행에 영향을 미치지 않음

- 작성 방법

🡪 여러 줄 주석문: /\* … \*/

🡪 한 줄 주석문: //

main()함수

- C++ 프로그램의 실행을 시작하는 함수

🡪 main() 함수가 종료하면 C++ 프로그램 종료

- main함수의 C++ 표준 모양

표준 표준 아님

|  |  |
| --- | --- |
| int main() {  …………..  return 0;  } | void main() {  ……………..  } |

- return 생략 가능

#include 문

- 전처리기(C ++ Preprocessor)에게 내리는 지시

- #include <헤더파일>또는 #include ”헤더파일”

🡪 “”을 사용하는 형식은 프로젝트 내에 헤더가 있을 시에 사용

🡪 <>을 사용하는 형식은 표준을 사용할 때 사용

- 지정한 헤더 파일을 컴파일 하기 전에 소스에 확장하도록 지시함

<iostream> 헤더 파일

- 표준 입출력을 위한 클래스와 객체, 변수 등이 선언되어 있음

🡪 ios, istream, ostream, iostream 클래스의 객체 선언

🡪 cout, cin, <<, >> 등 연산자 선언

화면 출력

cout 객체 🡪 출력 스트림 버퍼 내장

- 스크린 출력 장치에 연결된 표준 C++ 출력 스트링 객체

🡪 <iostream> 헤더 파일에 선언

🡪 std 이름 공간에 선언되어 있음 (std::cout으로 사용해야 함)

<< 연산자

- 스트림 삽입 연산자

🡪 ostream 클래스에 구현됨

🡪 오른쪽 피연산자 왼쪽 스트림 객체에 삽입

🡪 cout 객체에 연결된 화면에 출력

🡪 여러 개의 << 연산자로 여러 값 출력

문자열 및 기본 타입의 데이터 출력

- bool, char, short. int, long, float, double 타입 값 출력

- 연산식, 함수 호출가능

|  |
| --- |
| int n = 3;  char c =’#’;  std::cout << c << 5.5 << ‘-’ << n << “hello” << true;  std::cout << “n + 5 = ” << n+5;  std::cout < f(); 🡨 함수 f()의 값을 리턴 |

다음줄로 넘어가기

- ‘\n’ 사용

- std::endl 조작자 사용

namespace 개념

이름(identifier) 충돌 문제

- 발생 경우

🡪 여러 명이 서로 나누어 프로젝트를 개발하는 경우

🡪 오픈 소스 혹은 다른 사람이 작성한 소스나 목적 파일을 가져 와서 사용하는 경우 컴파일 하거나 링크하는 경우

- 이름 충돌 문제를 해결하기 위해 많은 시간과 노력이 필요함

namespace 키워드

- 이름 충돌 문제를 해결하기 위해 사용

- 개발자가 자신만의 이름 공간을 생성할 수 있도록 함

🡪 이름공간 안에 선언된 이름은 다른 이름공간과 별도 구분

이름공간 생성과 사용

- namespace 키워드를 사용하고 이름공간::이름으로 사용

|  |
| --- |
| namespace test {  int a;  }  test::a = 100; |

std

- ANSI C++ 표준에서 정의한 이름공간(namespace)

- 모든 C++ 표준 라이브러리는 std 이름 공간에 포함됨

🡪 <iostream> 헤더 파일에 선언된 모든 이름 (cout, cin, endl)등

- std 이름 공간에 선언된 이름을 접근 하기위해 std접두어 사용

std 생략

- using 지시어 사용

|  |
| --- |
| using std::cout; 🡪 std에서 cout만 생략  ………………………………..  cout << “Hello” << std::endl; |
| using namespace std; 🡪 std전부 생략  ………………………………..  cout << “Hello” << endl; |

C++ 변수 선언

변수 선언은 아무 곳이나 가능

C++ 변수 선언 방식의 장점

- C에서와 같이 변수 선언부와 실행 문 사이를 왔다 갔다 하는 번거로움 해소

- 변수를 사용하기 직전 선언함으로써 변수 이름에 대한 타이핑 오류 줄임

C++ 변수 선언 방식의 단점

- 선언된 변수를 일괄적으로 읽기 어려움

- 코드 사이에 있는 변수 찾기 어려움

C++ 문자열

C-스트링 방식

- ‘\0’으로 끝나는 문자 배열 (한자리당 1바이트 한글은 2바이트)

|  |
| --- |
| char name1[6] = {‘G’, ‘r’, ‘a’, ‘c’, ‘e’, ‘\0’} 🡪 c-스트링  char name2[5] = {‘G’, ‘r’, ‘a’, ‘c’, ‘e’} 🡪 일반 문자 배열 |

- 미리 크기를 지정해야 하고 끝은 null(\0)로 끝나야함

string 클래스 이용

- <cstring>(표준) or <string.h> (<string>도 가능)헤더 파일 선언 필요 🡪 #include <cstring>

- 문자열 관련 함수 사용 가능 srtlen() - 크기, strcmp() – 비교(-1 - 작다, 0 - 같다, 1 - 크다),

strcpy() – 복사

- 장점

🡪 C-string보다 다루기 쉽고 문자열 크기에 제약이 없으며 객체 지향적이다.

키 입력 받기

cin 객체

- 표준 입력 장치인 키보드를 연결하는 C++입력 스트림 객체

- 입력 버퍼를 내장하고 입력된 키를 립력 버퍼에 저장

🡪 <Enter>키를 입력 해야 입력 버퍼에서 키 값을 읽어 변수에 전달

🡪 도중애 <Backspace> 키를 입력하면 입력된 키 삭제

>> 연산자

- 스트림 추출 연산자

🡪 입력 스트림에서 값을 읽어 변수에 저장

🡪 >> 연산자를 연속 사용하여 여러 값 입력 가능

🡪 <Enter> 키가 입력 되면, cin의 입력 버퍼에서 키 값을 읽어 변수에 전달

- C++ 산술 시프트 연산자(>>)가 <iostream> 헤더 파일에 스트림 추출 연산자로 재정의됨

공백이 낀 문자열을 입력 받는 방법

- cin.getline(char buf[], int size, char delimitChar); 🡪 배열 이름, 배열 사이즈, 구분자(디폴트: \n)

- getline(cin, string name); 🡪getline 전역 함수 안에 cin인자와 스트링 변수명 입력

객체의 특성

객체의 일부분 공개

- 외부와의 인터페이스(정보 교환 및 통신)을 위해 객체의 일부분 공개

- C++ 객체는 멤버 변수(상태)와 멤버 함수(행동)로 구성된다.

C++ 에서의 클래스 & 객체

클래스

- 객체를 만들어 내기 위해 정의된 틀

- 클래스는 객체, 실체가 아님

- 클래스에 멤버 변수&멤버 함수 선언

객체

- 객체는 클래스의 모양을 그대로 가지고 생성됨

- 맴버 변수&맴버 함수로 구성

- 메모리에 할당되므로, 실체(instance)라고도 부름

- 하나의 클래스(틀)에서 여러 개의 객체 생성 가능

- 동일한 클래스의 각 객체는 상호 별도의 공간에 생성

클래스 작성 방법

클래스 선언 부와 클래스 구현부로 구성

|  |
| --- |
| class Circle{ 🡪 클래스 선언부  public:  int radius; 🡪 멤버 변수  double getArea(); 🡪 멤버 함수  }  double Circle::getArea() { 🡪 클래스 구현부  return 3.14\*radius\*radius;  } |

클래스 선언부(class declartion)

- class 키워드를 이용하여 멤버 변수와 멤버 함수 선언하는 부분

🡪 멤버 변수는 클래스 선언 내에서 초기화 할 수 없음

🡪 멤버 함수는 원형(prototype) 형태로 선언

- 멤버에 대한 접근 권한을 지정함

🡪 접근지정자 private, public, protected 중 하나로 지정 가능

private: 멤버에 대해 다른 클래스나 객체에서 접근을 차단을 지정(디폴트 값)

public: 멤버에 대해 다른 클래스나 객체에서 접근 허용을 표시

클래스 구현부(class implementation)

- 클래스에 정의된 멤버 함수의 구현 부분

- 컴파일러에 소속된 클래스를 명시해야 한다

생성자

생성자(constructor)

- 객체가 생성되는 시점에서 자동으로 호출되는 멤버 함수

🡪 이름 : 클래스 이름과 동일

🡪 리턴 타입을 명기하지 않음

🡪 매개변수 형식이 다른 생성자의 중복 선언 가능

|  |
| --- |
| class Circle{  ……………….  Circle();  Circle(int r);  ………………………………………..  };  Circle::Circle(){  ……………….  }  Circle::Circle(int r){  ……………….  } |

생성자의 목적

- 객체가 생성될 떄 객체가 필요한 초기화를 위해 호출

- 객체를 사용하기 전에 필요한 조치 수행하기 위함

🡪 멤버 변수 값 초기화, 메모리 할당, 파일 열기, 네트워크 연결 등

생성자 이름

- 반드시 클래스 이름과 동일

생성자 리턴 타입

- 리턴 타입 없음

🡪 void 타입으로 지정해도 안됨

🡪 생성자에서 return 구문은 사용 가능

각 객체의 생성시 생성자 함수가 자동으로 호출됨

- 객체 생성시 자동으로 호출됨

- 임의로 호출할 수 없음

- 오직 한 번만 호출됨

생성자는 중복 가능

- 생성자는 한 클래스 내에 여러 개 선언 가능

🡪 매개변수 개수 또는 타입이 다르면 OK

- 중복된 생성자 중 하나만 실행됨

생성자가 선언되어 있지 않으면 기본 생성자가 자동으로 생성됨

- 기본 생성자 : 매개 변수가 없는 생성자

- 컴파일러에 의해 자동 생성

여러 생성자에 중복 작성된 코드의 간소화

- 타겟 생성자와 이를 호출하는 위임 생성자로 나누어 작성

🡪 타겟 생성자 : 객체 초기화를 전담하는 생성자

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

|  |
| --- |
| Circle::Circle() : Circle(1); 🡪 위임 생성자  Circle::Circle(int r) { 🡪 타겟 생성자  radius = r;  cout << “반지름 “ << radius << “ 원 생성” << 두이;  } |

생성자의 멤버 변수 초기화 방법

- 생성자 코드에서 초기화

- 생성자 서두에 초기값으로 초기화

|  |
| --- |
| Point::Point(): x(0),y(0);  Point::Point(int a, int b)  : x(a), y(b){  } |

- 클래스 선언부에서 직접 초기화

기본 생성자

- C++ 컴파일러에서 객체가 생성될 때, 생성자를 반드시 호출한다(없을 시 자동으로 기본 생성자 생성)

- 매개 변수 없는 생성자

- 디폴트 생성자라고도 부름

|  |
| --- |
| Class Circle {  Circle(); 🡪 기본 생성자  } |

- 클래스에 하나라도 생성자가 선언되어 있다면 기본 생성자는 자동으로 생성되지 않음

소멸자

소멸자

- 객체가 소멸되는 시점에서 자동으로 호출되는 함수

🡪 오직 한번만 자동 호출

🡪 임의로 호출할 수 없음

🡪 객체 메모리 소멸 직전 호출됨

소멸자의 목적

- 객체가 사라질 때 마무리 작업을 위함

- 실행 도중 동적으로 할당 받은 메모리 해제, 파일 저장 및 닫기, 네트워크 닫기 등

소멸자 선언

- 소멸자 함수의 이름은 클래스 이름 앞에 ~를 붙인다.

소멸자 특징

- 소멸자는 리턴 타입이 없고, 어떤 값 리턴하면 안됨

🡪 리턴 타입 선언 불가

- 중복 불가능

🡪 소멸자는 한 클래스 내에 오직 한 개만 작성 가능

- 소멸자는 매개 변수 없는 함수

- 소멸자가 선언 되어 있지 않으면 기본 소멸자가 자동 생성됨

🡪 컴파일러애 의해 기본 생성자 코드 생성

🡪 컴파일러가 생성한 기본 소멸자 : 아무 것도 하지 않고 단순 리턴

프로그램 실행 순서

객체가 선언된 위치에 따른 분류

지역 객체

- 함수 내에 선언된 객체로서, 함수가 종료하면 소멸된다.

지역 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 순서를 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

프로그램 실행 명령

프로그램 로딩

main() 함수 실행

함수 실행

함수 종료

main() 함수 종료

프로그램 종료

접근 지정자

캡술화의 목적

- 객체 보호, 보안

- C++에서 객체의 캡슐화 전략

🡪 객체의 상태를 나타내는 데이터 멤버(멤버 변수)에 대한 보호

🡪 중요한 멤버는 다른 클래스나 객체에서 접근할 수 없도록 보호

🡪 외부와의 인터페이스를 위해서 일부 멤버는 외부에서 접근 허용

접근 지정자

private – 동일한 클래스의 멤버 함수에만 접근 허용 (보호)

public – 모든 다른 클래스에 접근 허용 (공개)

protected – 클래스 자신과 상속 받은 자식 클래스에만 허용

멤버 변수는 private 지정이 바람직함

인라인 함수

함수 호출에 따른 시간 오버 헤드

함수 호출 -> 돌아올 리턴 주소 저장 -> CPU 레지스터 값 저장 -> 함수의 매개 변수를 스택에 저장 -> 함수 실행

함수 실행 -> 함수의 리턴 값을 임시 저장소에 저장 -> 저장한 레지스터 값 CPU에 복귀 -> 돌아올 주소를 알아 내어 리턴 -> 함수 호출

작은 크기의 함수

함수 실행 시간 < 함수 호출을 위해 소요되는 부가적인 시간 오버헤드

인라인 함수

- inline 키워드로 선언된 함수

- 인라인 함수를 호출하는 곳에 인라인 함수의 코드를 그대로 삽입

🡪 함수 호출이 일어나지 않음

- C++ 프로그램의 실행 속도 향상

🡪 자주 호출되는 짧은 코드의 함수 호출에 대한 시산 소모를 줄임

🡪 C++에는 짧은 코드의 멤버 함수가 많기 때문.

인라인 함수 처리

- 인라인 함수를 호출하는 곳에 인라인 함수코드를 확장 삽입

🡪 컴파일러에 의해 이루어짐

🡪 매크로와 유사

🡪 코드 확장 후 인라인 함수는 사라짐

인라인 함수 호출

- 함수 호출에 따른 오버헤드 존재하지 않음

- 프로그램의 실행 속도 개선

|  |
| --- |
| inline int odd(int x){  return (x%2); }  int main(){  if(odd(5)) 🡪 if(5%2)  cout << “홀수”; |

장점 - 프로그램의 실행 시간이 빨라짐

단점 – 인라인 함수 코드의 삽입으로 컴파일된 전체 크기 증가(통상적으로 최대 30% 증가)

인라인 함수 제약

- inline은 컴파일러에게 주는 요구 메시지

- 컴파일러는 inline 요구를 수용할지 여부를 결정함

🡪 recursion, 긴 함수, static, 반복문, goto 문 등을 가진 인라인 함수는 수용하지 않음

자동 인라인 함수

- 클래스 선언부에 구현된 멤버 함수(선언과 동시에 구현)

🡪 inline으로 선언할 필요 없음

🡪 컴파일러에 의해 자동으로 인라인 처리

🡪 생성자 포함, 모든 함수가 자동 인라인 함수 가능

구조체

구조체(struct)

- 상속, 멤버, 접근 지정 등 모든 것이 클래스와 동일

- 클래스와 유일하게 다른점

🡪 구조체의 디폴트 접근 지정 – public

🡪 클래스의 디폴트 접근 지정 – private

- 구조체 수용의 이유

🡪 C언어와의 호환성 때문(C언어의 구조체와 100% 호환하여 C의 소스 사용 가능)

구조체 선언

|  |
| --- |
| Struct StructName{  private:  public:  protected:  } |

구조체 호출

struct StructName stObj;

C++프로그래밍 원칙

c++ 소스의 2가지 원칙

- 클래스 헤더 파일과 cpp 파일로 분리하여 작성

🡪 클래스마다 선언부와 구현부를 분리 저장

🡪 전체 프로그램 관리 및 클래스 재사용 용이 목적

- main() 등 전역 변수 함수나 변수는 다른 cpp파일에 분산 저장

🡪 필요하면 클래스가 선언된 헤더파일을 include함

- 헤더 파일을 중복 include할 때 생기는 문제

|  |
| --- |
| #include <iostream>  using namespace std;  #inculde “Circle.h”  #inculde “Circle.h” //컴파일러 오류  #inculde “Circle.h” |

🡪 중복으로 헤더 파일을 include할 경우 C2011 오류 발생

조건 컴파일문

|  |
| --- |
| 헤더 파일  #ifndef CLRCLE\_H 🡪 해당 헤더 파일이 include 되지 않았다면 헤더 파일 include  #define CLRCLE-H  ……  #endif |

🡪 조건 컴파일러 문의 상수는 클래스 이름으로 하는 것이 좋다. (꼬임 방지)

- 조건 컴파일러문을 이용해여 cpp파일에 여러 번 헤더 파일이 include되더라도 컴파일러 오류를 막을 수 있다.

헤더파일 호출

#include <헤더파일>

- 헤더 파일을 찾는 위치 : 컴파일러가 설치된 폴더에서 찾으라는 지시

#include “헤더파일”

- 헤더 파일을 찾는 위치 : 개발자의 프로젝트 폴더, 개발자가 컴파일 옵션으로 지정한 include 폴더에서 찾도록 지시

- 개발자 자신이 만든 헤더 파일을 사용하는 경우

포인터

포인터(pointer)

- 주소를 가지고 있는 변수

메모리 구조

- 변수는 메모리에 저장된다.

- 메모리는 바이트 단위로 엑세스된다.

- 첫번째 바이트의 주소는 0, 두번째 바이트는 1 (index)

- 변수의 크기에 다라서 차지하는 메모리 공간이 달라진다.

🡪 변수의 메모리상 위치는 사용하고 있는 컴퓨터 환경에 따라 달라진다, 변수의 실제 주소 값이 얼마인지는 중요하지 않다.

주소 계산 연산자 &

- 변수의 저장 시작 주소를 반환한다.

|  |
| --- |
| int i = 10  cout << hex << &i; //hex : 16진수로 출력해달라는 변수 조작자  🡪 1245024 |

포인터 선언

|  |
| --- |
| int \* p; //int형 기억 공간의 시작 주소를 저장 |

간접 참조 연산자 \*

- 포인터가 가리키는 값을 가져오는 연산자

|  |
| --- |
| int i = 10;  int \*p = &i;  cout << \*p; //p가 담고 있는 주소에 위치에 저장된 데이터를 엑세스 출력: 10 |

- 포인터는 다른 변수처럼 변경 가능

- 초기화가 안된 포인터는 사용 불가

- 포인터 타입과 변수의 타입을 일치 해야함

포인터 연산

- 가능한 연산 ++, --, +, -

증가 연산

- 증가되는 값(주소)은 포인터가 가리키는 객체의 크기

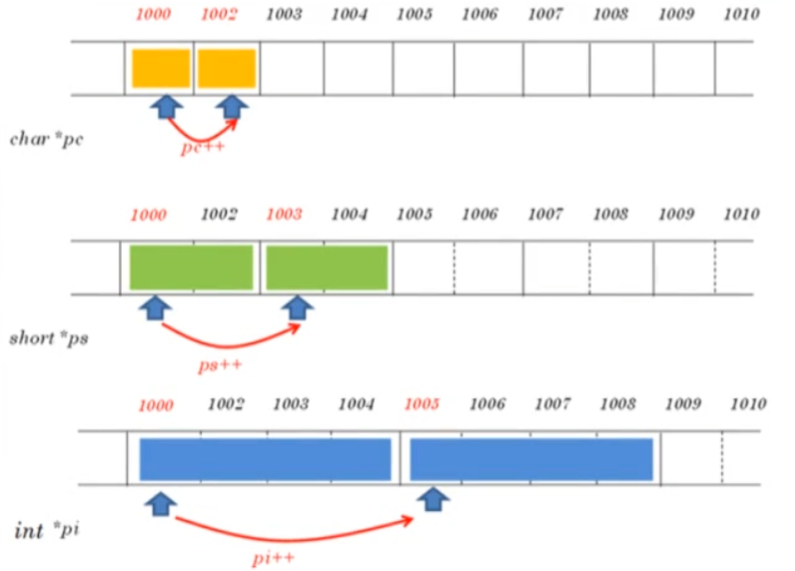
char 🡪 1

short 🡪 2

int 🡪 4

float 🡪 4

double 🡪 8



객체 포인터

- C언어의 포인터와 동일

- 객체의 주소 값을 가지는 변수

- 포인터를 이용한 멤버 접근

🡪 객체 포인터 -> 멤버 ex) p -> function();

🡪 (\*객체 포인터).멤버 ex) (\*p).function();

객체 배열

객체 배열 선언

- 선언 방식은 기존 배열과 동일

1. 객체 배열을 위한 공간 할당

2. 배열의 각 원소 객체마다 생성자 실행

🡪 c[0]의 생성자, c[1]의 생성자, c[2]의 생성자 실행

🡪 매개변수 없는 생성자를 호출 -> 매개 변수가 있는 생성자 호출 불가

- 객체에 생성자가 없다면 컴파일러에서 자동으로 기본 생성자를 생성한다

🡪 만약 생성자가 매개변수가 있는 생성자만 있다면 오류

객체 배열 소멸

- 배열의 각 객체마다 소멸자 호출

- 객체 생성의 역순으로 소멸

🡪 c[2]의 소멸자, c[1]의 소멸자, c[0]의 소멸자 실행

|  |
| --- |
| Circle circleArray[3];  circleArray[0].setRadius(10);  circleArray[1].setRadius(20);  circleArray[2].setRadius(30);  for(int i = 0; i < 3; i++)  cout << circleArray[i].getArea(); //배열의 각원소로 접근  Circle \*p = circleArray;  for(int i = 0; i < 3; i++){  cout << p->getArea(); //객체 포인터로 접근  p++; } |

객체 배열 초기화

- 객체 배열명[배열 크기] = {객체(생성자 메게 변수), 객체(생성자 메게 변수)}

ex) Circle circleArray[3] = {Circle(10), Circle(20), Circle(30)}

2차원 객체 배열

- 객체 배열명[배열 크기][배열 크기]

ex) Circle circleAray[2][3]

- 객체 배열명[배열 크기][배열 크기] = {{객체(생성자 메게 변수), 객체(생성자 메게 변수)}, {객체(생성자 메게 변수), 객체(생성자 메게 변수)}}

Circle circleAray[2][3] **= {**{Circle(10), Circle(20), Circle(30)}, {Circle(40), Circle(50), Circle(60)}**}**

동적 메모리 할당/반환

정적 할당

- 변수 선언을 통해 필요한 메모리를 할당

🡪 많은 양의 메모리는 배열 선언을 통해 할당

동적 할당

- 필요한 양이 예측되지 않는 경우, 프로그램 작성시 할당 받을 수 없음

- 실행 중에 운영체제로부터 할당 받음

🡪 힙(heep)으로부터 할당

힙(heep)

- 힙은 운영 체제가 송유하고 관리하는 메모리

- 모든 프로세스가 공유할 수 있는 메모리

C언어의 동적 메모리 할당

- malloc()/free() 라이브러리 함수 사용

C++의 동적 메모리 할당/반환

new 연산자 – 기본 타입, 배열, 객체, 객채 배열을 위한 동적 메모리 할당

- 객체의 동적 생성

🡪 힙 메모리부터 객체를 위한 메모리 할당 요청

- 객체 할당 시 생성자 호출

delate 연산자

- new 연산자로 받은 메모리 반환

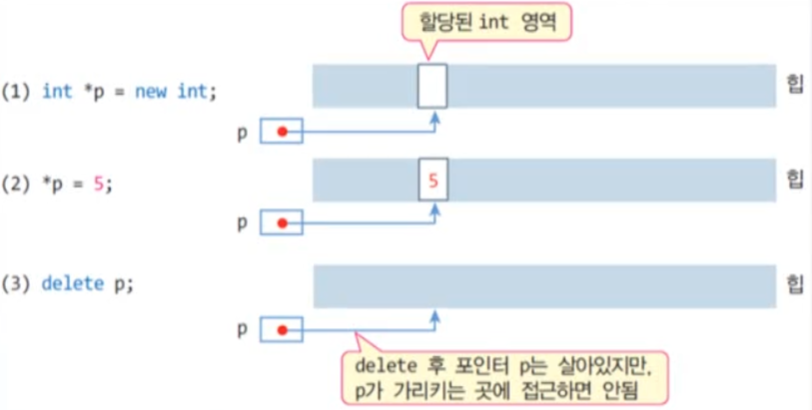
- 객체의 동적 소멸

🡪 소멸자 호출 뒤 객체를 힙에 반환

데이터 타입 \*포인터 변수 = new 데이터 타입;

delate 포인터 변수;

|  |
| --- |
| int \*pInt = new int;  char \*pChar = new char;  Circle \*pCrcle = new Circle();  delate pInt;  delate pChar;  delate pCrcle; |



delate 사용 주의 사항

- 적절치 못한 포인터로 delate하면 런타임 오류 발생

🡪 동적으로 할당 받지 않은 메모리 반환 – 오류

🡪 동일한 메모리 두 번 반환 – 오류

배열의 동적 할당/반환

데이터 타입 \*포인터 변수 = new 데이터 타입 [배열 크기];

delate [] 포인터 변수;

유의 사항

- delate시 [] 생략

🡪 컴파일 오류 발생 안 됨

🡪 동적 할당된 힙 기억공간은 비정상적인 반환

동적 할당 메모리 초기화

데이터 타입 \*포인터 변수 = new 데이터 타입(초기값);

- 배열은 동적 할당 시 초기화 불가능

객체의 동적 할당/반환

클래스 이름 \*포인터 변수 = new 클래스 이름;

클래스 이름 \*포인터 변수 = new 클래스 이름(생성자 매개 변수 리스트);

delate 포인트 변수;

객체 배열의 동적 할당/반환

클래스 이름 \*포인터 변수 = new 클래스 이름 [배열 크기];

delate [] 포인터 변수;

- 일반 객체 배열과 같이 소멸자 별도 실행, 생성의 반대순으로 반환

포인터를 이용한 접근 방식

|  |
| --- |
| pArray -> setRadius(20);  (pArray + 1) -> setRadius(30); //포인터 변수에 1을 더해서 다음 인덱스의 배열 객체 지정가능 |

this 포인터

- 객체 자신을 가르키는 포인터

- this 포인터 변수 선언

🡪 컴파일러가 선언한 변수

🡪 개발자가 선언하는 변수 아님

- this 포인터 사용

🡪 클래스이 멤버 함수 내에서만 사용 가능

🡪 컴파일러에 의해 멤버 함수내에 묵시적으로 삽입되는 매개 변수

🡪 생성자 함수에서는 생략 가능

🡪 static 멤버 변수에서는 사용 불가

this가 필요한 경우

- 매개 변수의 이름과 멤버 면수의 이름이 같은 경우

|  |
| --- |
| Circle(int radius){  this->radius = radius; } |

- 멤버 함수가 객체 자신의 주소를 리턴 할 때

🡪 연산자 중복시에 매우 필요

|  |
| --- |
| Sample \*f() {  …….  return this;  } |

this 연산자 제약 사항

- 멤버 함수만 사용 가능

🡪 멤버 함수 외의 함수에서는 this사용 불가(객체와의 연관성이 없기 때문)

- static 멤버 함수에서 this 사용 불가

🡪 객체가 생기기 전에 static 함수 호출이 있을 수 있기 때문

string 클래스

- C++ 표준 라이브러리, <string> 헤더 파일에 선언

- 가변 크기의 문자열 처리 가능

- 다양한 문자열 연산을 실행하는 연산자와 멤버 함수를 포함

🡪 문자열 복사, 문자열 비교, 문자열 길이 등

- 문자열, 스트링, 문자열 객체, string 객체 등으로 혼용 사용 가능

문자열 생성

|  |
| --- |
| string str; // 빈 문자열 스트링 객체  string address(“서울시 성복구 삼선동 389”); //문자열 리터럴로 초기화  string copyAdress(address); //address 복사  char text[] = {‘a’, ‘b’, ‘c’}  string title(text) //배열 text를 문자열로 변환 |

new/delate를 사용하여 동석 생성/반환 가능

|  |
| --- |
| string \*p = new string”C++”  delate c++; |

문자열 출력 - cout << 문자열;

문자열 입력 – cin >> 문자열;

공백 문자가 포함된 문자열 입력

- getline() 전역함수 사용

🡪 <string> 헤더 파일 include 필요

|  |
| --- |
| string name;  getline(cin, name, ‘\n’); |

string 클래스 추가 기능

문자열.length() - 문자열 길이 반환

문자열.substr(인덱스1, 숫자) – 인덱스1부터 숫자 만큼의 문자열 분리

문자열.find(찾을 문자, 시작 인덱스) – 시작 인덱스부터 찾을 문자 검색, 없을 때 -1 반환

stoi(문자열) – 문자열을 수로 변환

문자열.replace(인덱스, 길이, 문자열) – 인덱스부터 길이까지 문자열로 변경

cin.ignore() – getline을 통해 문자를 입력할 때 입력 버퍼 제거



함수의 참조

인자 전달 방식

- 값에 의한 호출 (call by value)

🡪 호출하는 코드에서 넘어온 값이 매개 변수에 복사됨

- 주소에 의한 호출 (call by address)

🡪 호출하는 코드에서 넘어온 주소 값이 매개 변수에 저장됨

값에 의한 호출 : call by value

- 함수가 호출되면 매개 변수가 스택에 생성됨

- 호출하는 코드에서 값을 넘겨줌

- 호출하는 코드에서 넘어온 값이 매개 변수에 복사됨

주소에 의한 호출 : call by address

- 함수의 매개 변수는 포인터 타입

🡪 함수가 호출되면 포인터 타입의 매개 변수가 스택에 생성됨

- 호출하는 코드에서는 명시적으로 주소를 넘겨줌

🡪 기본 타입 변수나 객체 주소, 배열 이름 전달

함수에 객체 전달

값에 의한 호출

- 함수를 호출하는 쪽에서 객체 전달

🡪 객체 이름만 사용

- 함수의 매개 변수 객체 생성

🡪 매개 변수 객체의 공간이 스택에 할당

🡪 호출하는 쪽의 객체가 매개 변수 객체에 그대로 복사됨

🡪 매개 변수 객체의 생성자는 호출되지 않음

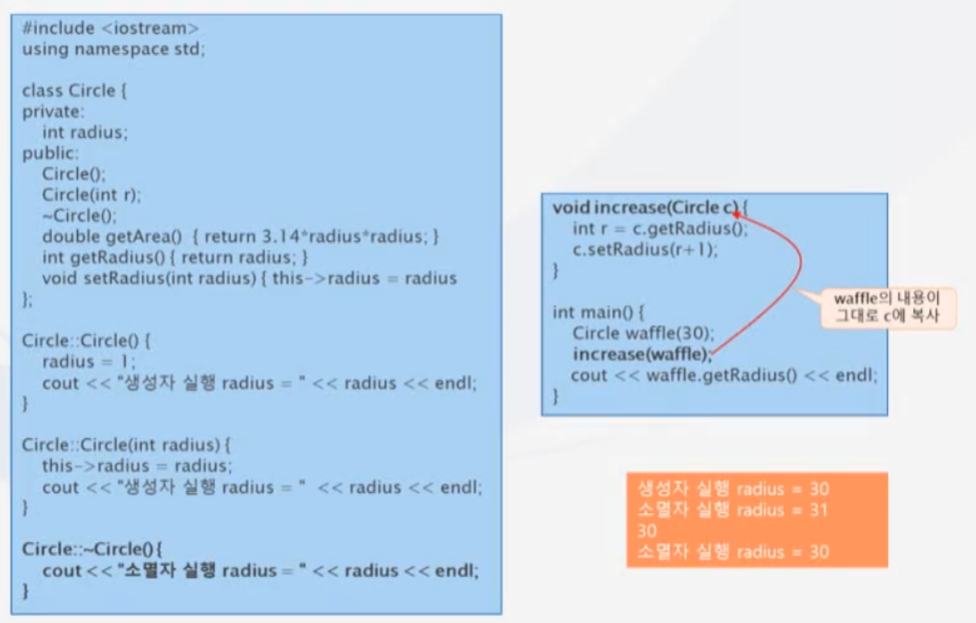
- 함수 종료

🡪 매개 변수 객체의 기억 공간에서 삭제

🡪 매개 변수 객체의 소멸자 호출

- 매개 변수 객체의 생성자 & 소멸자의 비대칭 실행 구조

- 호출되는 순간 실인자의 객체 상태를 매개 변수에 전달하기 위해 생성자 실행 X



주소에 의한 호출

- 함수 호출시 객체의 주소만 전달

🡪 함수의 매개 변수는 객체에 대한 포인터 변수로 선언

🡪 함수 호출 시 생성자 소멸자가 실행되지 않는 구조

값에의한 호출 vs 주소에 의한 호출

- 원본에 객체 복사 시간 필요 없음

- 생성자 소멸자 비대칭 문제 발생 없음

- 원본 객체 훼손 가능성 있음

객체 치환/리턴

객체 치환

- 동일한 클래스 타입의 객체끼리 치환 가능

- 객체의 모든 데이터가 비트 단위로 복사

- 치환된 두 객체는 현재 내용물만 같을 뿐 독립적인 공간 유지

객체 리턴

|  |
| --- |
| Circle getCircle() {  Circle tmp(30);  return tmp;  }  Circle c;  c = getCircle(); |

- 리턴을 통해 값 전달

참조 함수

참조의 의미

- 가르킨다는 의미

- 이미 존재하는 객체나 변수에 대한 별명

- 참조변수, 참조에 의한호출, 참조 리턴에서 활용

참조 변수

참조 변수 선언

- 이미 존재하는 변수에 대한 다른 이름(별명)을 선언

🡪 참조자 &의 도입

🡪 참조 변수는 이름만 생김

🡪 참조 변수에 새로운 공간을 할당하지 않음

🡪 초기화함으로써 지정된 원본 변수를 공유함

|  |
| --- |
| int n = 2;  int &reft = n;  Circle circle;  Circle &refc = circle; |

참조 변수 사용

- 보통 변수와 동일하게 사용 (포인터 아님)

주의 사항

- 참조 변수 선언 시, 초기화가 없다면 컴파일 오류 발생

- 참조 변수 선언 시의 참조자 &의 위치는 타입과 변수명 사이 존재

- 참조 변수의 배열을 만들 수 없음

- 참조 변수에 대한 참조 선언 가능

참조에 의한 호출 (call by reference)

- 함수의 매개 변수를 참조 변수 타입으로 선언

- 함수에서 참조 매개 변수 사용

참조 매개 변수

- 참조를 가장 많이 활용하는 사례

- 참조 매개 변수는 실인자 변수를 참조함

🡪 참조 매개 변수의 이름만 생기고 공간이 생기지 않음

🡪 참조 매개 변수는 실인자의 공간 공유

🡪 참조 매개 변수에 대한 조작 🡪 실인자 변수 조작 효과

|  |
| --- |
| void swap(int &a, int &b); // 함수에서 매개변수를 조작할 시 실인자에도 영향이 감  int m=2, n=9;  swap(m, n); |

참조에 의한 호출로 객체 전달

- 함수에서 매개 변수 객체가 생성되지 않음

🡪 참조 매개 변수는 이름만 전달됨

🡪 생성자와 소멸자가 실행되지 않음

- 참조 매개 변수의 모든 연산은 원본 객체에 대한 연산임

참조 리턴

C언어 함수 리턴

- 함수는 반드시 값만 리턴

🡪 기본 타입 값 : int, char, double 등

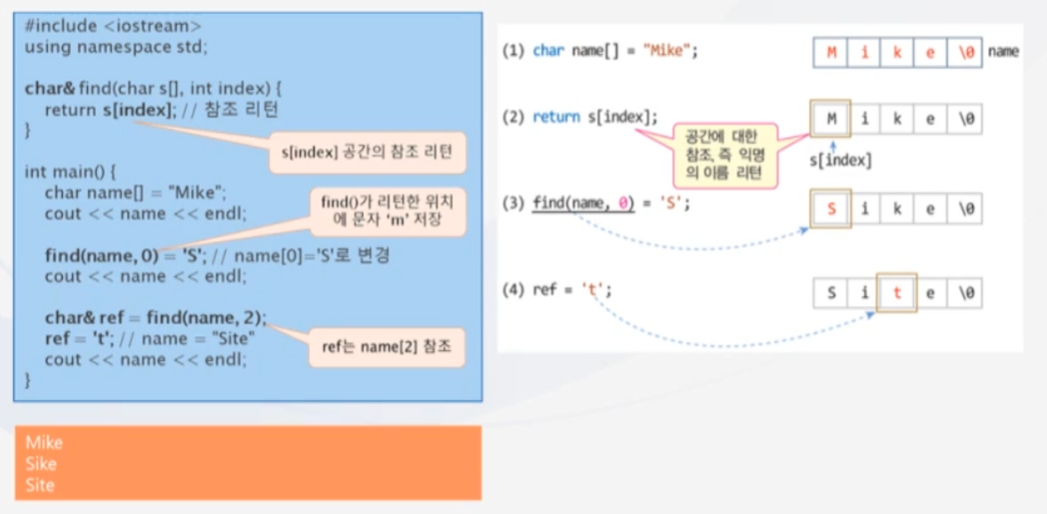
🡪 포인터 값

C++의 함수 리턴

- 함수는 값 외에 참조 리턴 가능

- 참조 리턴

🡪 변수 등과 같이 현존하는 공간에 대한 참조 리턴(변수의 값을 리턴 하는 것이 아님)



복사 생성자

얕은 복사(shallow copy)

- 객체 복사 시, 객체의 멤버를 1:1로 복사

- 객체의 멤버 변수에 동적 메모리가 할당된 경우

🡪 사본은 원본 객체가 할당 받은 메모리를 공유하는 문제 발생

|  |
| --- |
| int \*a = new int(10);  int \*b = new int(20);  a = b; //a는 b가 가지고 있는 주소를 복사함 |

깊은 복사(deep copy)

- 객체 복사 시, 객체의 멤버를 1:1대로 복사

- 객체의 멤버 변수에 동적 메모리가 할당된 경우

🡪 사본은 원본이 가진 메모리 크기만큼 별도로 동적 할당

🡪 원본의 동적 메모리에 있는 내용을 사본에 복사

- 완전한 형태의 복사

🡪 사본과 원본은 메모리를 공유하지 않음

|  |
| --- |
| int \*a = new int(10);  int \*b = new int(20);  \*a = \*b; //a가 가리키는 주소에 b가 가리키는 주소의 값을 복사함 |

복사 생성자(copy constructor)

- 객체의 복사 생성시 호출되는 특별한 생성자

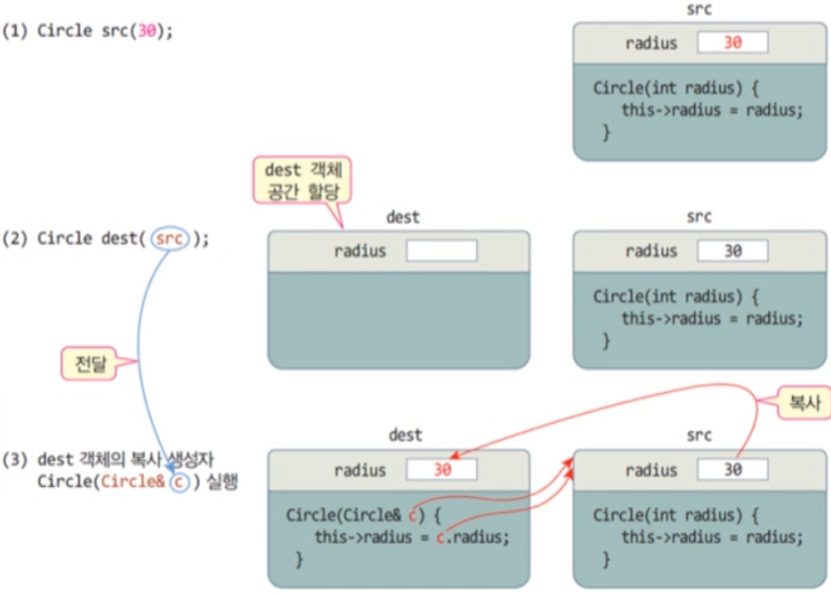
특징

- 한 클래스에 오직 한 개만 선언 가능

- 복사 생성자는 보통 생성자와 클래스 내에 중복 선언 가능

- 클래스에 대한 참조 매개 변수를 가지는 독특한 생성자

|  |
| --- |
| Circle (Circle& c); //복사 생성자 생성 |



디폴트 복사 생성자

- 복사 생성자가 선언되어 있지 않은 경우 자동으로 복사 생성자 삽입

|  |
| --- |
| 클래스이름::클래스이름(클래스이름& 매개변수){  this->변수 = 매개변수.변수; } |

함수 중복

함수 중복(function overloading)

- 동일한 이름의 함수가 공존

🡪 다형성

🡪 C 언어에서는 불가능

- 함수 중복이 가능한 범위

🡪 일반 함수들 간의 중복

🡪 한 클래스의 멤버 함수들 간의 중복

🡪 상속 관계에 있는 기본 클래스와 파생 클래스의 멤버 함수들 간의 중복

- 함수 중복의 성립 조건

🡪 중복 함수들의 이름이 동일할 것

🡪 중복된 함수들의 매개 변수 타입이 다르거나 개수가 다를 것(함수 구분)

🡪 리턴 타입은 함수 중복과 무관함

|  |
| --- |
| int sum(int a, int b, int c){  return a+b+c; }  int sum(int a, int b){  return a+b; }  double sum(double a, double b){  return a+b; } |

- 동일한 이름사용으로 인한 이점

🡪 함수 이름을 구분하여 기억할 필요 없음

🡪 함수 호출을 잘못하는 실수를 줄일 수 있음

생성자 함수 중복

- 생성자 함수 중복 목적

🡪 객체 생성시, 매개 변수를 통해 다양한 형태의 초기값 전달

- 소멸자 함수는 중복 불가능(매개 변수를 가지지 않고 클래스 내에서 하나만 존재하기 때문)

디폴트 매개 변수(default parameter)

- 매개 변수에 값이 넘어오지 않는 경우, 디폴트 값을 받도록 선언된 매개 변수

🡪 매개 변수 = 디폴트 값

|  |
| --- |
| void star(int a, int b = 5); |

- 디폴트 매개 변수는 일반 매개변수와 혼용 가능

🡪 일반 매개 변수 앞에 선언 불가

- 디폴트 매개변수는 함수의 중복을 간소화해준다

🡪 중복 함수와 같이 사용 불가

함수 중복의 모호성

- 함수 중복이 모호 하여 컴파일러가 어떤 함수를 호출하는지 판단하지 못하는 경우

🡪 형 변환으로 인한 모호성

|  |
| --- |
| float function(float a);  double function(double a);  function(3); // int타입인 3을 어떤 형으로 변환할지 모호함 |

🡪 참조 매개 변수로 인한 모호성

|  |
| --- |
| float function(float a);  float function(float &a);  function(3); // 원초적으로 참조 매개 변수로인한 중복 불가 |

🡪 디폴트 매개 변수로 인한 모호성

|  |
| --- |
| float function(float a);  float function(float a, string b = “”);  function(3); // 디폴트 매개변수로 인해 두 함수 모두 가리킴 |

static 멤버

static

- 변수와 함수에 대한 기억 부류의 한 종류

🡪 생명 주기 – 프로 그램이 시작될 때 생성, 프로그램 종료 시 소멸

🡪 사용 범위 – 선언된 범위, 접근 지정에 따름

클래스의 멤버

- static 멤버

🡪 프로그램이 시작할 때 생성

🡪 클래스 당 하나만 생성, 클래스 멤버라고 불림

🡪 클래스의 모든 인스턴스(객체) 들이 공유하는 멤버

- non-static 멤버

🡪 객체가 생성될 때 함께 생성

🡪 객체마다 객체 내에 생성

🡪 인스턴스 멤버라고 불림

static 멤버 변수 생성

- 전역 변수로 선언

🡪 전체 프로그램 내에 한 번만 생성

🡪 외부 선언이 없으면 링크 오류 발생

|  |
| --- |
| class Person{  public :  double money;  void addMoney(int money){  this->money += money; }  static int sharedMoney;  static void addSherd(int n) {  shardMoney += n; }  };  int person::sharedMoney = 0; |

- static 멤버는 하나만 생성해서 모든 객체와 공유함

테이블이(가) 표시된 사진

자동 생성된 설명

static 멤버 접근

- static 멤버는 객체 이름이나 객체 포인터로 접근

🡪 객체.static멤버

🡪 객체포인터->static멤버

- 클래스 이름과 범위 지정 연산자(::)로 접근 가능

🡪 클래스명::static멤버

static의 주요 활용

- 전역 변수나 전역 함수를 클래스에 캡슐화

🡪 전역 변수나 전역 함수를 가능한 사용하지 않도록

🡪 전역 변수나 전역 함수를 static으로 선언하여 클래스 멤버로 선언

- 객체 사이에 공유 변수를 만들고자 할 때

🡪 static 멤버를 선언하여 모든 객체들이 공유

static 멤버 함수

- static 멤버 함수는 static 멤버만 접근 가능

🡪 static 멤버 함수가 접근할 수 있는 것 : static 멤버 함수, static 멤버 변수, 함수 지역 변수

- static 멤버 함수는 non- static 멤버에 접근 불가

🡪 객체가 생성되지 않은 시점에서 static 멤버 함수가 호출될 수 있기 때문

- non-static 멤버 함수는 static 멤버에 접근 가능

- static 멤버 함수는 this 사용 불가

🡪 static 멤버 함수는 객체가 생기기 전부터 호출 가능

🡪 static 멤버 함수에서 this 사용 불가