**sepTeam-B**

**팀원 : 김민지, 노태욱, 오진욱, 정민재**

* **extern & static**
1. 함수일 때
	1. extern
		1. 다른 함수에서 해당 함수를 호출할 수 있다.(default)
		2. extern 지정은 함수가 정의된 파일 외에 다른 파일 에서 함수의 호출을 허용한다.
	2. static
		1. 함수가 정의된 파일에서만 이 함수를 호출할 수 있다.
		2. 다른 파일에서는 이 함수를 호출할 수 없다.
		3. 여러 파일에 같은 함수 이름이 존재해도 에러가 나지 않는다.
2. 변수일 때
	1. extern
		1. extern 변수는 한번 선언되면 NULL로 기본값이 초기화 된다.
		2. 외부변수(예:extern int a)는 함수밖에서 선언되어 선언된 시점 이후 모든 함수에서 사용가능한 변수이다.(각 함수간 자료 공유)
		3. 프로그램 종료 시 까지 메모리에 할당되어 소멸되지 않는다.

예) a.cpp에 int aa; 라는 외부변수가 선언됨

 b.cpp c.cpp 에 extern int aa; 로 선언하면 a.cpp의 int aa를

사용가능하다.

* + 1. extern 선언(외부파일을 참조)한다는 것은 메모리 할당은 하지 않고, 각 파일 간의 자료를 공유한다는 뜻이다.
		2. extern 선언을 함수 내에서 하게 되면 해당 함수 내에서만 사용가능하다.

예) void AAA() { extern int a; } 를 하면 AAA() 내에서만 int a 외부변수를 사용한다.

* 1. static
		1. static 변수는 한번 선언되면 NULL로 기본값이 초기화 된다.
		2. 내부 static 변수
			1. 함수내부에서 선언되고 사용되며, 다른 함수에서는 사용 불가능.
			2. 자료의 값이 소멸되지 않고, 메모리에 영구적으로 존재하는 변수
			3. 함수 실행 후에 값이 유지되고, 이전 값을 알 수 있음.
			4. 모듈의 독립성 보장을 위해 사용되는 변수.

예) static int sdt = 0;

처음에만 0 값이고 처음에만 초기화 되고, 다음 함수에 들어올 때는

초기화되지 않고 사용 가능하며 값이 유지된다.

* + 1. 외부 static 변수
			1. 함수의 외부에 선언된 static 변수
			2. 해당 파일 내에서만 사용가능하고, 해당 파일의 전체 함수에서 사용될 수 있는 변수
			3. 다른 파일에서는 사용 불가능하다.(외부참조 X)
			4. 다른 파일에 같은 이름이 존재하는 외부 static 변수가 있더라도 서로 영향을 미치지 않는다.

**extern 예)**

// file a.c

int temp=12; // extern 변수 선언 - 실제로 메모리가 할당되는 부분은 이곳이다. temp는 static으로 선언한 것과 동일하다.

void main()

{

printf("%d", temp);

sub();

}

// file b.c

extern int temp; // 실제로 메모리가 할당되는 것이 아니고 함께 link되는 파일 중에 temp가 선언되어 있다고 표시만 하는 것이다.

void sub()

{

printf("%d", temp);

}

**static 예)**

static int a=23; // 외부 static 변수

void main()

{

printf("%d", a); // 이전에 a가 선언되었으므로 사용 가능

printf("%d", b); // 이전에 b가 선언되어 있지 않으므로 사용 불가능

sub1();

sub2();

}

void sub1()
{

static int b=45; // 내부 static 변수

printf("%d", a);

printf("%d", b);

}

void sub2()

{

printf("%d", b); // sub1에서 선언하였기 때문에 사용할 수 없다.

}

단,

void sub1()

{

static int a=3;

void sub2()

{

printf("%d", a); // 접근, 출력 가능

}

}

sub1에서 선언되어 있지만, sub2는 sub1도 포함되어 있기 때문에 sub2에서도 a를 사용할 수 있다.

* **싱글톤(Singleton)**
1. 전역 객체가 필요한 경우에 유용하게 사용할 수 있다.
2. 구현 예제

#include <iostream>

using namespace std;

class CSingleton

{

private:

static CSingleton \*pInstance;

CSingleton() : m\_iValue(10) {}; // 일반적인 방법으로 객체를 생성하는 것을 막기위해

~CSingleton() {} // 외부에서 싱글톤 객체를 강제 delete 하는 것을 막기 위해 필요.

int m\_iValue;

public:

static CSingleton& getInstance()

{

if(pInstance == NULL) pInstance = new CSingleton;

 return \*pInstance;

}

static CSingleton\* getInstancePtr()

{

if(pInstance == NULL) pInstance = new CSingleton;

 return pInstance;

}

static void releaseInstance()

{

if(pInstance != NULL) delete pInstance;

}

void setValue(int in) { cout << m\_iValue << "-->" ; m\_iValue = in; cout << m\_iValue << endl; }

 int getValue() { return m\_iValue; }

};

CSingleton \*CSingleton::pInstance = NULL;

int main(){

CSingleton::getInstance().setValue(3);

CSingleton::getInstance().setValue(5);

CSingleton::releaseInstance();

return 0;

}

1. 구현의 핵심

3.1. 생성자를 private으로 감추어서 외부에서 새로운 객체를 여러 개를 생성하지 못하게 막는다.

3.2. 유일하게 생성된 객체에 접근하는 방법을 제공한다.

 (getInstance()가 여기에 해당)

1. 늦은 초기화(동적 할당을 이용)를 하는 이유

4.1. 싱글톤 객체가 필요한 시점에 동적 할당을 하여 메모리 낭비를 없애기 위해

4.2. 다른 전역 객체의 생성자에서 참고하고 싶은 경우 문제가 발생할 수 있음.

1. new/delete

5.1. 프로그램이 종료되는 시점에 new로 생성한 메모리까지 해제되기 때문에

 굳이 명시적인 메모리 해제를 할 필요는 없음.

5.2. 다만, 외부 시스템과 정리해야 하는 것이 있다면 명시적인 해제 함수(releaseInstance)

를 만들어서 사용

* **싱글톤의 다른 구현 예제**
* class Printer
* {
* public:
* void SubmitPrint();
* ..
* **friend Printer& thePrinter();**
* private:
* **Printer();**
* ...
* };
*
* Printer& thePrinter()
* {
* static Printer p;
* return p;
* }
*
* // 호출
* thePrinter().SubmitPrint();

by “More Effective C++”

1.     Printer 클래스는 생성자가 private으로 선언되어 있습니다. 하나의 인스턴스만 만들어야 하기 때문에 아무나 이 클래스를 만들지 못하게 하기 위해 이와 같은 강력한 제한을 두는 거지요.

2.     Factory 로 쓰이는 thePrinter() 함수는 Printer 의 friend 로 되어 있습니다. 즉 이 함수는 언제든 Printer 의 모든 멤버 함수에 접근할 길을 터 놓았습니다.

3.     thePrinter() 구현 함수에서 Printer 를 static 으로 선언했습니다. 정적 객체로 만들어진 변수의 장점은 잘 아시다시피

A.     하나의 인스턴스만 생성된다는 것입니다.

B.     함수 thePrinter() 가 호출되기 전에는 Printer 가 만들어 질 일이 없습니다. 최소한 한번이라도 호출되어야 생성된다고 보장하는 셈입니다. 바로 이 점이 Printer 클래스를 단순히 전역으로 선언하는 것보다 훨씬 나은 방법이지요.

이 정도만으로도 아주 훌륭한 싱글톤 클래스라고 생각합니다. 여기서 조금 더 나아간다면

class Printer

의 헤더와 구현 부분을 실제 이 클래스를 사용하는 클라이언트와 완전히 분리하기 위해 interface IPrinter 와 같은 virtual 로만 이뤄진 인터페이스만 제공하는 방법이 있습니다. 이는 당연히 Printer 클래스의 수정으로부터 클라이언트를 보호해서 대규모 개발에 적합한 설계로 나아가기 위함입니다.