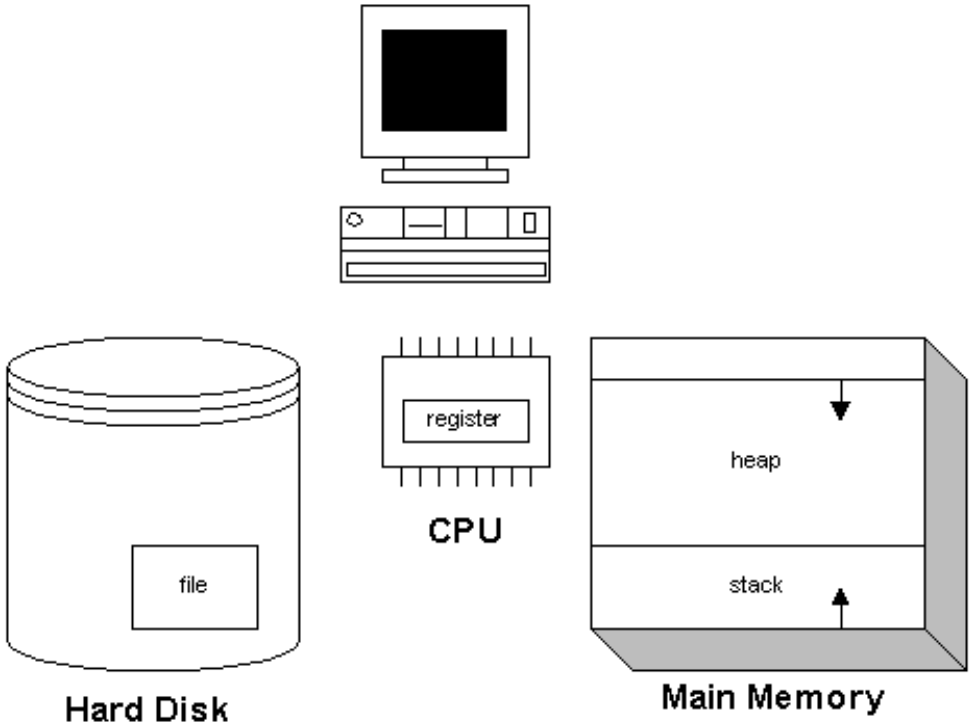
12. 변수의 종류, 범위 규칙(scope rule)

- 기억 부류의 지정은 **기억 부류 지정자(storage class specifier)**에 의해 이루어집니다.
- C언어에서 기억 부류 지정자는 auto, static, register, extern과 mutable이 있습니다.
- auto는 C++11 표준에서 예전의 의미로 사용되지 않고, 변수를 선언할 때 변수의 타입을 컴파일러가 자동으로 하라는 지시자로 변경되었습니다. 그래서 auto의 역할은 소스의 확장자가 .c일때와 .cpp일 때 다릅니다.

- auto
- static
- register
- extern
- mutable



기억 부류: 각각의 변수는 고유의 기억 장치에 할당됩니다. 스택, 힙, 레지스터, 디스크는 모두 메모리입니다.

스택(stack)

• C언어에서 변수 이름 앞에 붙은 auto는 이 변수가 메모리의 스택에 할당됨을 나타냅니다.

```
#include <stdio.h>

void f(int k) {
    int i=3,j=4;

    printf("%d,%d,%d\n",i,j,k);
}//f

void main() {
    int i;

    i=2;
    f(i);
}//main
```

- 위의 소스는 아래와 대등합니다.

```
#include <stdio.h>

void f(auto int k) {
    auto int i=3,j=4;

    printf("%d,%d,%d\n",i,j,k);
}//f

void main() {
    auto int i=2;

    f(i);
}//main
```

C++11에서 auto의 의미

• Visual Studio 2013이상의 버전에서 아래의 코드가 확장자가 .cpp인 소스에 포함되어 있다면 컴파일되지 않습니다.

```
#include <stdio.h>
#include <memory.h>
#include <malloc.h>

void main()
{
    auto int i; // C++11에서 auto는 int와 결합할 수 없습니다.
    i = 3;
    printf("%i\r\n", i);
}//main()
```

• C++11 표준이 정해진 이후 auto의 의미는 변경되었습니다.
• auto는 사용자가 변수의 타입을 명시적으로 지정하지 않아도 컴파일시간의 문맥에 의해서 자동으로 결정되는 타입의 변수를 선언할 때 사용합니다.

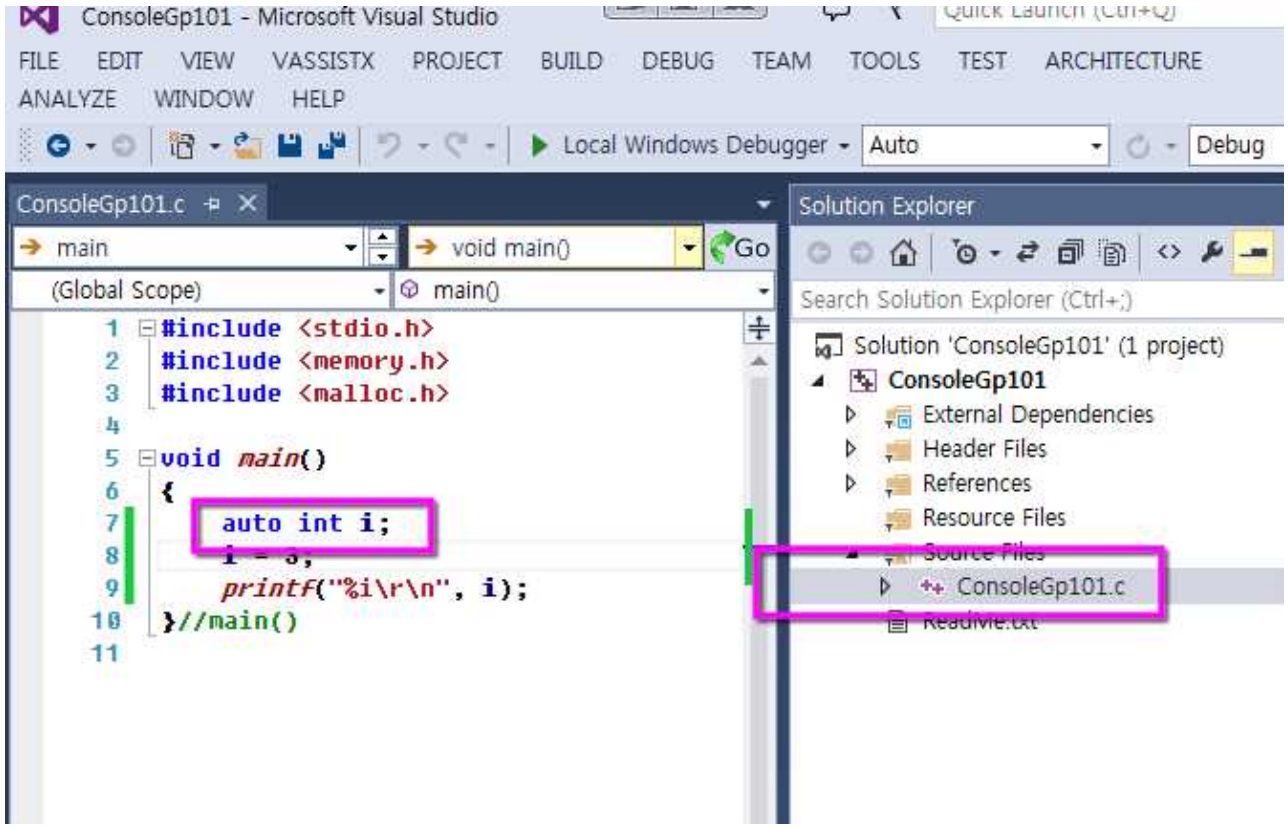
```
int i = 1;
```

• 이제 위 문장은 C++11에서 다음과 같이 작성할 수 있습니다.

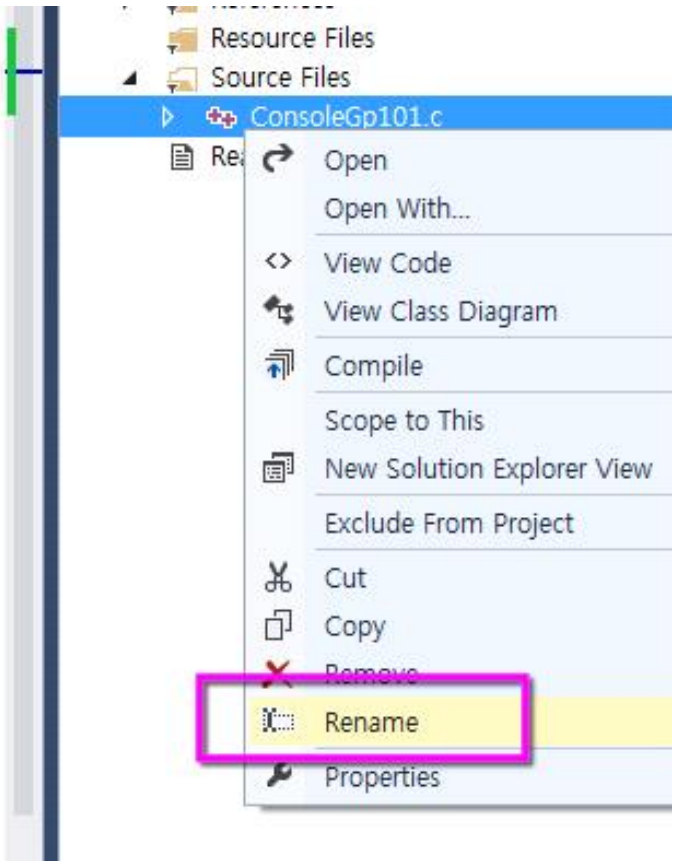
```
auto i = 1;
```

• 컴파일러는 초기화 문장으로 i의 타입을 유추합니다. 1에 해당하는 적절한 값은 int이므로 컴파일 시간에 i의 타입을 int로 결정하는 것입니다.
• 아래의 문장은 C++11을 지원하는 Visual Studio 2013에서는 에러가 발생합니다. 초기화 값이 없으므로 i의 값을 유추할 수 없기 때문입니다.

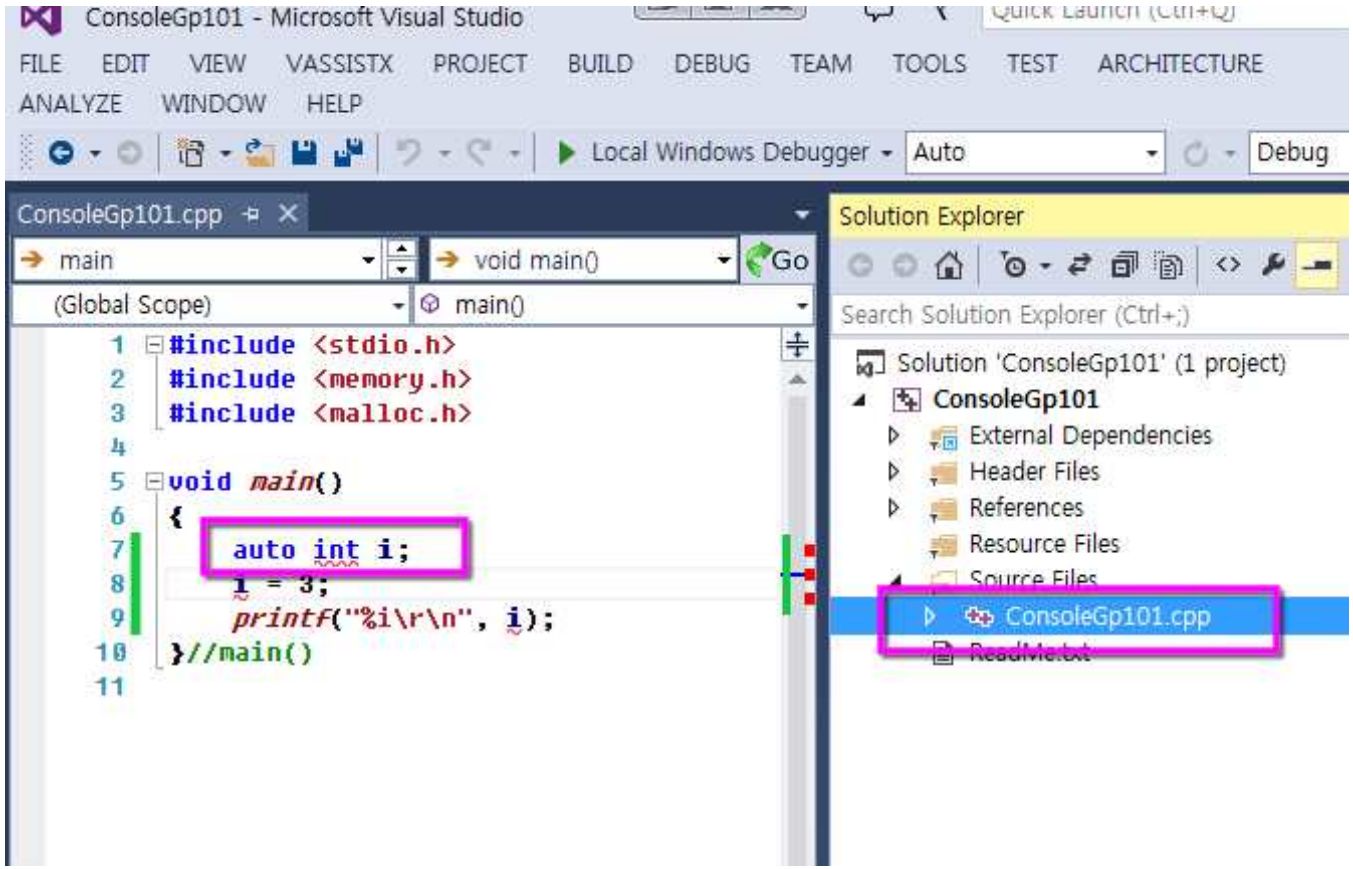
```
auto i;
```



C에서 auto의 의미: 이 그림에서 auto는 확장자 .c인 소스의 내부에서 사용되었으므로, 변수가 stack에 할당되어야 함을 지시하는 역할을 합니다.



Visual Studio 2013 IDE에서 파일 이름 변경: 마우스 오른쪽 버튼을 눌러서 소스 파일의 이름을 변경할 수 있습니다.



C++11에서의 auto의 의미: 이 그림에서 auto는 확장자 .cpp인 소스에서 사용되었으므로 변수의 타입을 자동으로 결정하라는 지시자 역할을 합니다.

힙(heap)

- 힙은 동적 메모리 할당(dynamic memory allocation)시에 사용됩니다.

```
#include <stdio.h>

int i;

void main() {
    printf("%d\n",i);
    //       0
}//main
```

• 전역 변수를 static이라고 선언하는 것과, 그렇지 않은 것은 다릅니다. 전역 변수를 static으로 선언하면, 그 변수는 파일 범위를 가집니다. 하지만, static 변경자를 사용하지 않으면 전역 범위를 가집니다.

```
#include <stdio.h>

static int i;//i의 범위는 파일로 제한됩니다.

void main() {
    printf("%d\n",i);
}//main
```

• C에서 함수의 상태를 유지하는 방법은 전역 변수 혹은 정적 변수를 사용하는 것입니다.

```
#include <stdio.h>

void f() {
    int i=0;

    printf("%d\n",i);
    ++i;
}//f

void main() {
    f();
    f();
    f();
}//main
```

• f()의 변수 i를 정적(static)으로 선언하면, 이것은 별도의 공간에 유지되므로 함수가 호출될 때마다, 초기화가 되지 않습니다.

```
#include <stdio.h>

void f() {
    static int i;//이 문장은 컴파일 시간에 단 한 번 실행됩니다.

    printf("%d\n",i);
    ++i;
}//f

void main() {
    f();
    f();
    f();
}//main
```

• 함수 f()가 호출될 때마다 3으로 초기화하라는 문장이 아니라, 프로그램 시작 시에 i를 3으로 초기화하라는 의미입니다.

```
#include <stdio.h>

void f() {
    static int i=3;

    printf("%d\n",i);
    ++i;
}//f

void main() {
    f();
    f();
    f();
}//main
```

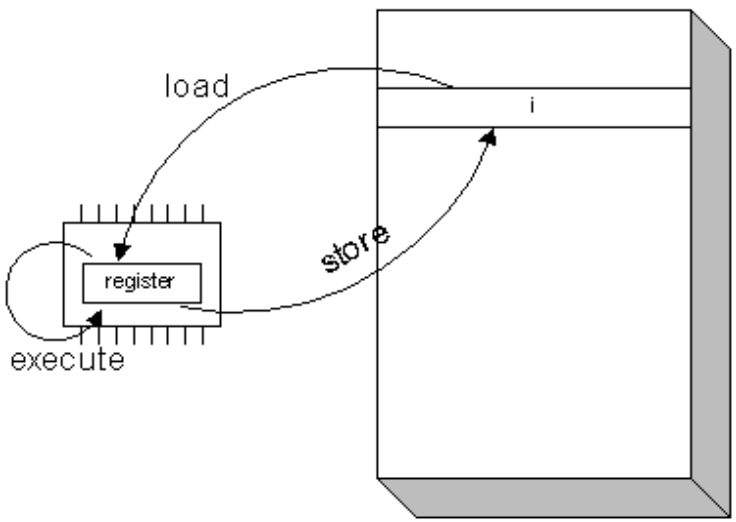
```
#include <stdio.h>

void f() {
    static int i;

    i=3;//이 문장은 매번 실행됩니다.
    printf("%d\n",i);
    ++i;
}//f

void main() {
    f();
    f();
    f();
}//main
```

레지스터(register)



기계 사이클(machine cycle): 대부분의 명령은 다음과 같은 순서로 수행됩니다. 명령어 가지고 오기(fetch) → 명령어의 해독(decode) → 명령 실행(execute) → 결과 저장(store). 기계 사이클 중 가장 많은 시간을 차지하는 부분은 load와 store일 것입니다. 이것은 메모리의 성능에 따라 다르지만, 일반적으로 CPU의 레지스터에서 작업하는 것에 비해 수 십배 이상 느립니다.

• 변수를 참조하기 위해 무조건 메모리의 내용을 읽으라고 지시하는 경우에는 **volatile** 키워드를 사용해야 합니다. volatile은 반드시 그 변수의 메모리 값이 읽힌다는 보장을 하는 코드를 생성합니다.

```
volatile int ticks;

void timer( ) {
   ticks++;
}
void wait (int interval) {
   ticks = 0;
   while (ticks < interval);  // Do nothing
}
```

파일(file)

• 마지막으로 살펴 볼 변수의 기억 부류는, 이 변수를 가리키는 소스 코드가 미리 컴파일 되어서 디스크에 다른 오브젝트 파일로 존재하는 경우에 발생합니다.

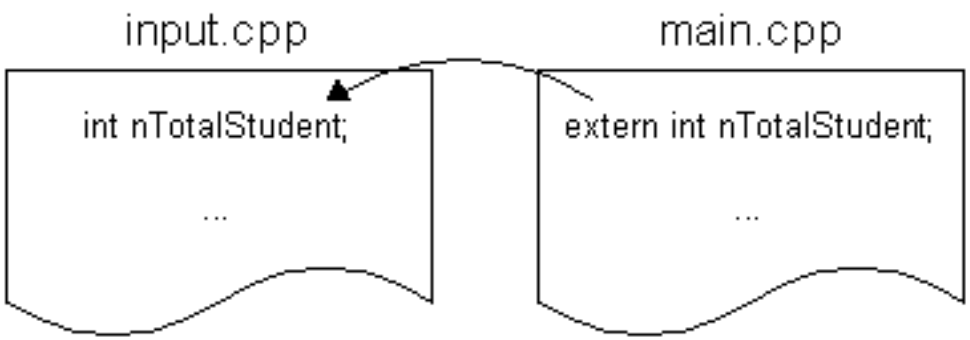
```
//input.cpp
...
int nTotalStudent;//이 변수에 학생 전체 수가 관리된다고 가정합시다.
...
void ReadDataFromFile() {
        ...
}
...
```

• input.cpp의 변수 nTotalStudent는 아마도 main.cpp와 output.cpp에서 모두 사용될 것입니다. 그렇다면, main.cpp에서 input.cpp에서 사용하는 변수 nTotalStudent를 어떻게 선언하는가요? 해답은 바로 extern입니다.

//main.cpp

extern int nTotalStudent;

...



extern은 변수가 다른 파일에 이미 존재한다는 선언이다.

• 어떤 파일에서 선언된 전역 변수가 반드시 다른 파일에서 볼 수 있는 것이 아니다는 - extern을 사용하더라도 - 것을 주의해야 합니다. 만약 input.cpp에서 nTotalStudent가 static 변경자로 선언되었다면,

static int nTotalStudent;

• 이 변수는 input.cpp에서만 볼 수 있습니다. 즉 main.cpp에서 이 변수를 참조하기 위해, 아래처럼 사용할 수 없습니다.

extern static int nTotalStudent;

- **파일 범위(file scope)**
- **전역 범위(global scope)**

• extern 키워드와 관계된 다른 사항들을 살펴봅시다. 이것은 C언어로 컴파일된 함수를 C++에서 사용하려고 하는 경우 발생합니다. C와 C++가 함수의 실행코드(execution code)를 생성하는 방식은 조금 다릅니다. C++는 유일한(unique) 함수를 호출하기 위해 오버로딩된 함수들에 대해 **이름 바꾸기(name mangling)**를 수행합니다.

• C++에서 C의 함수를 호출하기 위해서 선언할 때, C 함수의 헤더 앞에 extern "C"를 붙여주어야 합니다.

extern "C" void cfunc(int);

• 이러한 C함수가 굉장히 많다면, 함수마다 extern "C"를 붙여주는 것은 번거로운 일입니다. 이 때에는 extern "C" { ... } 블록안에 C함수의 선언을 적으면 됩니다.

```
extern "C" {
    void cfunc(int);
    ...
}
```

가시범위(visibility)에 의한 구분

• 변수가 선언되었을 때, 이 변수를 어디에서 사용할 수 있는가요? 또한 어디에서 사용할 수 없는가요?

• 이러한 규칙은 **범위 규칙(scope rule)**이라고 알려져 있으며, 특정 변수가 그 범위에서 사용 가능하면, **볼(visible) 수 있다**고 합니다.

• 변수는 다음과 같은 5가지 범위(scope)를 가질 수 있습니다■.

① 블록 범위(block scope) - 지역 변수(local variable)
② 파일 범위(file scope) - 전역 변수(global variable)
③ 전체 범위(global scope) - 전역 변수(global variable)
④ 프로토타입 범위(prototype scope)
⑤ 이름공간 범위(namespace scope)

블록 범위(block scope)

• 블록은 여는 중괄호(open brace: {)와 닫는 중괄호(close brace: })로 표현되며 다음과 같은 특징을 가집니다.

① 변수 선언을 임의의 위치에 가질 수 있습니다.
② 하나의 문장 취급됩니다. 또한, 겹쳐짐(nesting)이 가능합니다. 즉, 블록 안에 또 다른 블록이 올 수 있습니다.

• 블록은 변수 선언을 가질 수 있는데, 이처럼 블록 안에서 선언된 변수를 지역 변수라 합니다.

```
#include <stdio.h>

void main() {                   //main 블록
    int i=1,k=100;
    {                           //1번 블록
        int j=2;
        {                   //2번 블록
            int i=3;
            printf("%d,%d\n",i,j);
        }
        printf("%d,%d\n",i,j);
        int i=4;
        printf("%d,%d,%d\n",i,j,k);
    }
    printf("%d\n",i);
}//main
```

전역 범위(global scope)

- 전역 변수는 블록 안에서(클래스를 포함하여) 선언되지 않은 변수를 말합니다.

```
#include <stdio.h>

int i=1;

void f(int j) {
    printf("%d,%d\n",i,j);
}

int j=2;

void g() {
    printf("%d,%d\n",i,j);
}
```

```
void main() {
    int i=3;

    f(i);
    g();
}//main
```

• 전역 변수 i는 f(), g()와 main()에서 모두 볼 수 있습니다.
• 전역 변수 j는 f()이후에 선언되었으므로, g()와 main()에서만 볼 수 있습니다.
• 지역 변수와 전역 변수가 모두 보이는 경우는 어떻게 할 것인가요? 이 때의 규칙은 아래와 같습니다.

"**같은 이름의 지역 변수와 전역 변수가 존재하는 경우, 가까운 쪽, 즉 지역 변수를 사용합니다.**"

프로토타입 범위(prototype scope)

```
#include <stdio.h>

int f(int i,int j);//함수도 먼저 선언되어야 합니다.

void main() {
    printf("%d\n",f(2,3));
}//main

int f(int i,int j) {
    return i*j;
}
```

• f()는 main()위에 선언되었습니다. 함수 f()의 선언은 함수의 이름과 파라미터의 형과 개수를 검사하므로, i와 j라는 명칭은 더미(dummy)입니다.

• 이것은 i와 j는 컴파일러가 무시함을 의미합니다. 그러므로 다음과 같은 선언도 타당합니다.

```
int f(int k, int l);
```

• 이때 k나 l등은 **프로토타입 범위(prototype scope)**를 가진다고 합니다.

실습문제

1. 이미 만들어져서 .obj형태로 존재하는 C함수들을 C++에서 사용하기 위해서, C++의 소스코드에서 해 주어야 하는 것은 무엇인가요?(Hint: extern "C" 블록)

2. 다음 이름이 같은 두 함수 f()의 실행 결과는 어떻게 다른가요? 이유를 설명하세요.

```
int f() {
    static int i=0;
    return ++i
}

int f() {
    static int i;
    i=0;
    return ++i;
}
```

3. a.cpp에 전역 변수가 i가 static으로 선언되어 있습니다.

static int i=1;

이 변수를 b.cpp에서 참조하기 위해 적당하게 프로젝트를 설정한 다음, 다음과 같은 문장을 사용하였습니다.

extern int i=1;

그랬더니 컴파일 시간 에러가 발생하였습니다. 이 장에서 설명하였듯이, a.cpp의 i 선언에서 static을 생략해야 됩니다. 그래서 a.cpp의 i 선언을 다음과 같이 바꾸었습니다.

int i=1;

그랬더니 역시 에러가 발생하였습니다. 무엇이 잘못인가요?

@