2009년 06월 17일

변수의 scope와 그 생애 (Memory Map)

변수의 Scope에 대해서 전형적인 C의 관점에서 그리고, 그것이 실제 Memory에서는 어떻게 운용되는지에 대한 이야기가 시작 됩니다. 간단하게, C coding 관점에서 변수의 Scope에 대해서 먼저 이야기를 늘어 놓고, Memory Map의 관점에서 변수의 Scope에 대해서 알아보도록 보겠습니다.

이런 이야기를 늘어 놓게 될 줄이야. High Level Language 언어를 다루는 측면에서의 변수 Scope와, Memory Map에서의 변수의 특성을 잘 파악하시면, Compile을 하여, Linker를 이용하여 Executable Image를 만든다던가, 하는 작업할 때, 상당히 유용한 지식이 될 수 있습니다.

변수의 유형들을 늘어 놓으면 다음과 같지요.
auto
extern
static
volatile

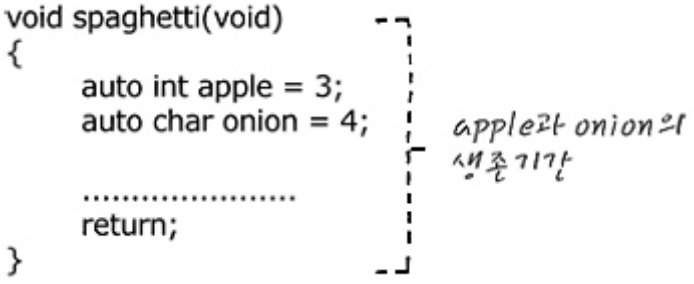
보통 Local, Global variable이라고들 부르는 용어가 있는데, 이는 변수의 scope와 생존기간을 모두 포함한다고 하여, 별로 좋아하지 않는 사람들도 있습니다만, 이 변수의 구분이 상당히 reasonable하면서도 간단하여, 저는 상당히 선호하는 편이에요. 이제부터 scope = 생존기간이라고 생각하셔도 무방 합니다요.

auto 변수 유형 - LOCAL 변수

auto 변수란 흔히들 auto Local변수라고도 부르는데, 이런 변수들의 동작 범위는 자신이 선언된 함수 또는 block - {}으로 둘러 쌓인 영역이욧 - 가 됩니다. 다른 의미로는 존속기간은 이런 auto 변수가 정의된 함수가 수행되는 동안이 됩니다. 이런 변수는 그 함수의 수행이 모두 종료되면 return과 동시에 안녕 사라져 버립니다. 보통은 이런 auto라는 단어를 선언 전에 붙여 쓰지 않습니다.

이런 변수들을 Global 함수의 반대의 의미로 local 변수라고도 부릅니다만. 그 이유로는 c file하나만 보았을 때, 전체적으로 영향을 미치지 않고, 국지적 - Local 적으로 - 으로 함수 하나나, {}로 둘러싸인 block에만 영향을 미치므로 그렇게 부릅니다.

예를 들어, spaghetti() 라는 함수가 있을 때,



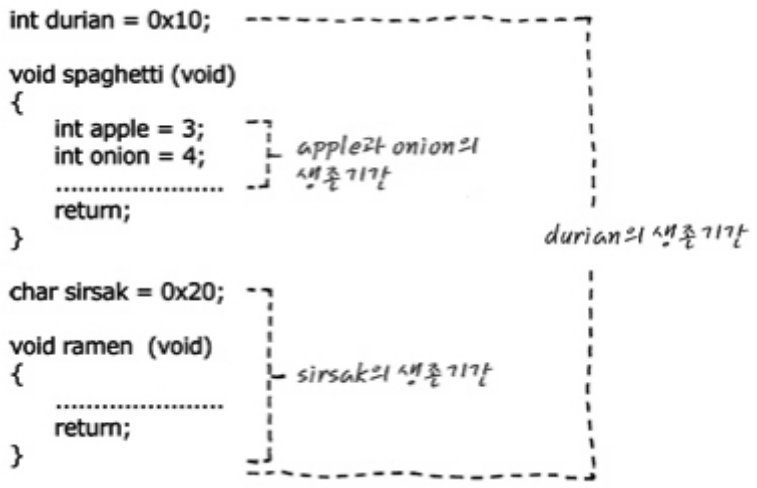
이런 모양의 함수라면, 그림에서 볼 수 있듯이, apple과 onion의 생존기간은 두 변수를 선언한 spaghetti()가 그 흥망성쇠를 다함과

동시에 automatically 삼천 궁녀처럼 사라진다고 볼 수 있겠지요. 그러니까, 이 함수 이외의 다른 곳에서는 갑자기 혜성처럼 나타나 건드릴 수 없습니다.
뭐, 물론 pointer등으로 다른 함수를 호출한다면, 다른 얘기가 되겠지만.

extern 변수 유형 - GLOBAL 변수

extern 변수 유형은 보통은 Global 변수라고 불리 우는 유형으로서, auto local 변수 유형과 다른 점은 변수가 정의 된 위치가 다른데, 이런 Global 변수는 함수의 안쪽이 아닌 함수 바깥쪽에 정의됩니다. 이런 Global 변수들은 이 변수가 선언된 된 파일의 전체에 영향을 끼치며 살아 갑니다. 결국 자신이 정의된 위치부터 파일의 끝까지 전체에서 사용될 수 있습니다.

recipies.c



그림에서 볼 수 있듯이, extern 변수 유형은 어떤 file에서의 선언위치에 따라 그 생존기간이 정해지게 됩니다. 이런 긴 생존기간 덕분에 Global 변수라고도 불리지만, extern 변수 유형이라고 따로 불리 우는 이유는 변수 선언만 해주면 프로그램 어느 부분에서도 사용할 수 있으며, 심지어는 다른 file에서도 불러다 쓸 수 있기 때문 입죠.

그 방법으로는 다른 file에서 만약 durian 변수를 쓰고 싶다면, 다른 파일의 쓰고자 하는 위치에
extern int durian; 이라고 선언만 해준다면, compiler에게 다른 파일의 어딘가에 global로 durian이 선언되어 있을 것이고, link 할 때 그 변수를 가져다 쓸 테니 지금은 error를 내지 말고 그 자리를 비워 두거라 하는 의미가 되지요.

static 변수유형

static 유형의 변수들은 static이 변수 선언 앞에 붙어서 선언되는데, 이런 static 유형은 local static 유형과 global static 유형이 있어요. 뭐 어려운 건 아니고 위의 auto와 extern 유형의 변수 앞에 static이라고 하나만 덧붙여서 선언하면 되는데, 이 static이라는 말을 붙임으로써 local 변수와 global 변수는 약간 다른 의미를 갖게 됩니다.

일단 local static의 예를 들어 다음과 같은 함수가 있다고 칩시다. - 뭐 정말 어디에도 쓸데 없는 Hello, World code보다도 의미 없지만, 이런 상황에는 이런 임기응변 code가 좋을 것 같습니다만 -

```
void spaghetti (void)
{
  static int apple = 3;
  static int onion = 4;

  apple ++;
  onion ++;
```

```
    return;
}
```

여전히 apple과 onion은 spaghetti() 함수 안에서만 존재하고 생존할 수 있어요. 이때 spaghetti() 함수가 호출 될 때 마다 무슨 일이 벌어질 지 상상해 보셨나요? 만일 static이라는 말이 안 붙어 있다면 spaghetti()가 불리 울 때마다 apple = 4, onion = 5가 된 후, 결국 각각 4와 5가 된후 사라 집니다. 결국 spaghetti()가 불릴 때마다 apple과 onion은 증가 하지만, 함수가 끝나는 순간 영원히 안녕~! 인 것이죠.

하지만, static으로 apple과 onion이 선언된 경우에는 다른 이야기가 되지요. 무슨 이야기냐 하면, static으로 무장한 local 변수는 생존 기간은 자신을 생성한 함수 내부에서만 가능하지만, 그 함수가 끝이 나더라도 그 값은 유지 하고 있습니다. 그러니까, 다시 말해, spaghetti()가 불릴 때 마다 apple과 onion은 계속 증가하게 되어요. spaghetti()가 한번 불리우면 apple = 4, onion =5, 두 번 불리우면 apple = 5, onion = 6 이런 식으로 자신의 값을 굳건하게 보존하고 있지요. 끝까지 살아 남는다 구요.

그러면, Global static은 어떻게 다를까요? Global 함수는 어차피 자신의 값을 유지 할 수 있을 텐데, 무엇이 다른 걸까요?

recipes.c

```
static int durian = 0x10;

void spaghetti (void)
{
    int apple = 3;
    int onion = 4;
    ..................
    return;
}

static char sirsak = 0x20;

void ramen (void)
{
    ..................
    return;
}
```

Global static은 여전히 생존기간은 recipes.c 전체이지만, 일반 Global 변수와 다르게, 다른 file에서 extern을 선언하여 가져갈 수 없습니다. Global static은 다른 file로 부터 감추는 역할을 합니다. 이런 Global static의 이용은 보통 여러 사람이 같이 큰 project를 진행 할 때, 자신의 file에서만 사용하고, 다른 사람이 가져다 쓰지 못하게 하는데 사용 하는 예가 많습니다 .C++로 따지자면, protect와 같은 원리라고 보면 되겠습니다요.

결국 static으로 선언하게 되면 값은 항상 유지 하되, 그 변수를 선언한 함수 또는 파일에서만 사용하도록 국한 시키는 속성을 지니게 합니다. 참 재미 있습니다. 다시 말해, 변수를 static을 붙여 넣어 선언을 하면 Global 변수처럼 항상 자기 값을 기억할 수 있게 되지요. 대신 접근성은 선언한 곳에서만 가능합니다. 함수 안에서 선언하면 함수 안에서만, file안에서 선언하면 file안에서만 가능하구요, extern으로 가져다 쓰지 못하죠. 다시 말하지만 protect된 것과 마찬가지입니다.

Volatile 유형

Compiler의 Smart함 때문에 생겨난 불운의 유형입니다. 보통 Compiler는 compile을 할 때, 이런 저런 이유를 들어가며, 할 일들을 최대한 줄여 나갑니다. 이런 것을 optimization이라고 하는데, 어찌된 셈인지 code를 만드는 사람의 의도와 상관없이 이상한 story로 흘러갈 때가 있습니다.

예를 들어, 아래와 같이 하릴없이 integer type의 dummy에 1부터 10까지 집어 넣는 코드를 만들었다고 칩시다.

```
int strawberry(int arg)
{
   int *dummy;

   dummy = &arg;

   *dummy = 1;
   *dummy = 2;
   *dummy = 3;
   *dummy = 4;
   *dummy = 5;
   *dummy = 6;
   *dummy = 7;
   *dummy = 8;
   *dummy = 9;
   *dummy = 10;

    return *dummy;
}
```

이 딸기 함수가 하는 일은 dummy가 가리키는 주소에 1부터 10까지 넣는 일을 하고 있는데, code creator는 무슨 일이 있더라도 꼭 dummy가 가리키는 주소에 1부터 10까지 순차적으로 집어 넣어야 직성이 풀린다는 의지를 표현하고 있습니다.

만일 우리가 이 함수를 compile하게 된다면, compiler는 optimization을 시도해서 아래와 같이 의도 하지 않는 방향으로 code를 만들어 버리죠.

```
int strawberry(int arg)
{
   int *dummy;

   dummy = &arg;

   *dummy = 10;

    return *dummy;
}
```

어랏, compiler는 1~9까지 집어 넣는 작업을 모두 없애 버리고, 결국 dummy가 가리키는 주소에 10만 넣으면 되지 하고 멋대로 판단해 버립니다. 1~9를 열심히 넣어도, 결국에 10만 제대로 들어가 있으면 되지 하면서 게으름을 피워 버리지요.

이럴 때를 대비해서 volatile int *dummy; 로 volatile이라는 예약어를 사용하여 선언하게 된다면, compiler가 오호, 이건 optimization하지 말라는 의미로구나! 하고 알아 듣게 되어, *dummy에 1~10까지 순차적으로 넣어주게 됩니다. 그러니까, 어느 면에서는 compiler에게 "너 이건 중요한 sequence니까 optimization하지 마." 라든가, "똑똑한척 하면서 게으름 피우지 말도록"하는 의미로 사용하게 됩니다.

보통은 Device Driver를 Design할 때 이런 volatile을 많이 쓰게 되는데, 왜냐하면 Memory mapped I/O의 경우 같은 주소에 다른 값들을 연속해서 쓰게 되는 일들이 비일 비재 하게 되기 때문입니다. Memory Mapped I/O에 관련해서는 해당 section에서 또 자세히 보시지요.

이런 얘기 잘 안 합니다만, 다음 section의 trailer를 조금 하자면, 이제 이런 Global 변수와 Local 변수들이 메모리에 어떻게 위치하는가에 대한 흥미진진한 이야기를 계속 하려고 합니다.
다음 편을 기대해 주세요. 짜잔.

 Life span 즉, 수명에 대해서 말한다면
다음과 같이 구분할 수 있습니다요.

방식	생성방법	제거방법	다른용어로
자동	함수에 진입할 때 마다	return할때	Local 변수
정적	시스템시작할 때	시스템이 죽을 때 까지.	Global 변수
	(프로그램이 시작할때)	(프로그램 종료할때)	윈도우의 경우. PC등

변수, scope, memory, map, 생애, local변수, global변수

Linked at at 2009/10/01 12:32

... ⓙ 변수의 scope와 그 생애 (Memory Map) ... more

Commented by daniel at 2009/07/13 20:05

^^ 잘 읽고 갑니다~

Commented by 히언 at 2009/07/14 22:39

얏호~ 감사합니다~~