Articles from Veda Solutions

Extern keyword C

2012-11-21 11:11:30 techveda

Most newbies are often not clear about the of why and when extern keyword should be used. In this article I will attempt to give your more clarity on extern keyword and its importance.

It's very important to know the difference between definition statements and declaration statements before we understand what extern keyword is.

First we will try to understand what definition statement means and what compiler does with the definition statements.

Let's look into a sample code snippet

```
#include <stdio.h>

int g=10;

int main()

{

	int l=4;

	printf("%dn",l);

	printf("%dn",g);

	return 0;

}
```

In the above code snippet we can see there are two declaration statements, one declaring a global object "g" and other declaring a local object "l". Most of us tend to call them as declarations, that's true for the local object "l", but for global object "g" we cannot call it as a declaration, rather it's a definition.

C compilers will allocate memory for the variable definition statements and also initializes its contents with the given value.

Let's look into the assembly code.

```
.globl  g

	.data

	.align 4

	.type   g, @object

	.size   g, 4

g:

	.long   10
```

In the above assembly code snippet we can see that global variable "g" is allocated with 4 bytes and aligned with 4 bytes as part of .data section.

We know that all functions in C language are global symbols, and keeping this context, let's look at how main() information is stored.

```
.globl  main

	.type   main, @function

main:

.LFB0:

	.cfi_startproc

	pushq   %rbp

	.cfi_def_cfa_offset 16

	.cfi_offset 6, -16

	movq    %rsp, %rbp

	.cfi_def_cfa_register 6

	subq    $16, %rsp

	movl    $4, -4(%rbp)
```

```
        movl    g(%rip), %eax
        addl    $1, %eax
        movl    %eax, g(%rip)
        addl    $1, -4(%rbp)
        movl    $.LC0, %eax
        movl    -4(%rbp), %edx
        movl    %edx, %esi
        movq    %rax, %rdi
        movl    $0, %eax
        call    printf
        movl    g(%rip), %edx
        movl    $.LC0, %eax
        movl    %edx, %esi
        movq    %rax, %rdi
        movl    $0, %eax
        call    printf
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
```

From this it's very clear that compiler will allocate memory for global symbols and their information is stored under separate labels. This is done by compiler whenever it finds definition statements.

But local variable "l" is allocated in main stack frame and memory is allocated only after main starts.

Let's look at the relocatable file to understand it more clearly

```
objdump -t extern.o

extern.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*        0                    000000000000000 extern.c
0000000000000000 l    d  .text                                0000000000000000 .text
0000000000000000 l    d  .data                               0000000000000000 .data
0000000000000000 l    d  .bss                                 0000000000000000 .bss
0000000000000000 l    d  .rodata                             0000000000000000 .rodata
0000000000000000 l    d  .note.GNU-stack                0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame                          0000000000000000 .eh_frame
0000000000000000 l    d  .comment                           0000000000000000 .comment
0000000000000000 g     O .data                                0000000000000004 g
0000000000000000 g     F .text                                  000000000000005a main
0000000000000000         *UND*                                0000000000000000 printf
```

Here we can see that " g " is stored as part of data segment and its size is 4 bytes and it's a global object. "main" is a global function part of .text segment.

But there is no information about local variable "l". So for global symbols, memory is allocated as and when the program is loaded but main stack frame is allocated only when main starts.

From the above explanations it's very clear that for definition statements memory is

allocated by compiler.

Now let's try to understand what declaration statements are.

As we know that global symbols defined in one source file can be accessed from another source file. Let's look into one example to understand declaration statements.

Let's write three source files add.c, sub.c, and main.c

add.c

```
int r = -1;

int add(int x,int y)

{

            r = x+y;

            return r;

}
```

sub.c

```
int sub(int x,int y)

{

r = x-y;

return r;

}
```

main.c

```
int main()

{

            r = add(10,20);

            printf("%dn",r);

            r = add(20,10);

            printf("%dn",r);


            return 0;

}
```

In the above example we are accessing "add and r" symbols, defined in add.c, from another source file, main.c .

If we try to compile the above source files using the below command

```
 gcc add.c sub.c main.c
```

Compilers will throw the below errors

```
sub.c: In function 'sub':

sub.c:3:2: error: 'r' undeclared (first use in this function)

sub.c:3:2: note: each undeclared identifier is reported only once for each function it appears in

main.c: In function 'main':

main.c:6:2: error: 'r' undeclared (first use in this function)

main.c:6:2: note: each undeclared identifier is reported only once for each function it appears in
```

Now the problem is compiler can compile only one translation unit at a time.

When compiler compiles add.i (translation unit of add.c) "r" was defined, so there is no problem. But when sub.i and main.i was compiled, compiler could not find any information regarding "r". We cannot define "r " in sub.c and main.c as definition statements allocate memory for symbols, and memory should be allocated only for onces. So how can we give the information to compiler that "r" is defined in add.c and being used in sub.c and main.c.

This is where declaration comes. Before we access any global symbol, defined in one source file from another we need to tell to the compiler about the existence of the symbol, by declaring the symbol with extern keyword.

add.c

```
int r = -1;

int add(int x,int y)

{

       r = x+y;

           return r;

}
```

sub.c

```
extern int r;

int sub(int x,int y)

{

    r = x-y;

return r;

}
```

main.c

```
extern int r;

extern int add(int,int);

extern int sub(int,int);

int main()

{

           r = add(10,20);

           printf("%dn",r);

           r = add(20,10);

           printf("%dn",r);


           return 0;

}
```

Now in sub.c and main.c we declared “r” to tell to the compiler about the existence of the “r”.

Let’s look into relocated files to understand what extern keyword will tell to the compiler.

gcc -c add.c

objdump -t add.o

add.o: file format elf64-x86-64

SYMBOL TABLE:

0000000000000000 l df *ABS* 0000000000000000 add.c

0000000000000000 l d .text 0000000000000000 .text

0000000000000000 l d .data 0000000000000000 .data

0000000000000000 l d .bss 0000000000000000 .bss

0000000000000000 l d .note.GNU-stack 0000000000000000 .note.GNU-stack

0000000000000000 l d .eh_frame 0000000000000000 .eh_frame

0000000000000000 l d .comment 0000000000000000 .comment

0000000000000000 g O .data 0000000000000004 r

0000000000000000 g F .text 0000000000000020 add

As add.c contains the definitions of “add” function and “r” object we can see their information as part of SYMBOL TABLE.
But if we look into the relocatable file of sum.c

```
#gcc -c sub.c

#objdump -t sub.o
```

```
sub.o:     file format elf64-x86-64

SYMBOL TABLE:

0000000000000000 l    df *ABS*                                  0000000000000000 sub.c

0000000000000000 l    d  .text                                    0000000000000000 .text

0000000000000000 l    d  .data                                    0000000000000000 .data

0000000000000000 l    d  .bss                                     0000000000000000 .bss

0000000000000000 l    d  .note.GNU-stack                0000000000000000 .note.GNU-stack

0000000000000000 l    d  .eh_frame                        0000000000000000 .eh_frame

0000000000000000 l    d  .comment                         0000000000000000 .comment

0000000000000000 g     F .text                                    0000000000000024 sub

0000000000000000         *UND*                                   0000000000000000 r
```

In sub.c, “r” was declared but not defined, so we can see that r entry in Symbol Table as undefined (*UND*).

Same is true in main.o

```
#gcc -c main.c

#objdump -t main.o

main.o:     file format elf64-x86-64

SYMBOL TABLE:

0000000000000000 l    df *ABS*                                  0000000000000000 main.c

0000000000000000 l    d  .text                                    0000000000000000 .text

0000000000000000 l    d  .data                                    0000000000000000 .data

0000000000000000 l    d  .bss                                     0000000000000000 .bss

0000000000000000 l    d  .rodata                                  0000000000000000 .rodata

0000000000000000 l    d  .note.GNU-stack                0000000000000000 .note.GNU-stack

0000000000000000 l    d  .eh_frame                        0000000000000000 .eh_frame

0000000000000000 l    d  .comment                         0000000000000000 .comment

0000000000000000 g     F .text                                    0000000000000069 main

0000000000000000         *UND*                                   0000000000000000 add

0000000000000000         *UND*                                   0000000000000000 r

0000000000000000         *UND*                                   0000000000000000 printf

0000000000000000         *UND*                                   0000000000000000 sub
```

As we can see add, r, printf, sub symbols are not defined but declared, memory is not allocated for these symbols. So extern keyword is used to declare symbols to tell the compiler that it is defined is in some source file.

-
-
-
-
- More
-