Scope and storage class of variables

The *scope*of a variable refers to those portions of a program wherein it may be accessed.

Failure to understand scoping rules can lead to two problems:

(1) Syntax errors (easy to find and fix)

(2) Accidentally using the wrong instance of a variable (sometimes very hard one to find).

Two general rules apply

(1) The *declaration* of a variable must *precede* any *use* of it.

(2) If a particular line of code is in the scope of *multiple variables of the same name the innermost declaration of the variable is the one that is used.*

Specific refinements of these rule include:

(1) the scope of any variables declared outside any function is all code in the source module that appears *after* the definition.

(2) the scope of any variable declared inside a basic block is all code *in that block and any blocks nested within that block that appears after*the definition.

Improper definition location

```
 1 /* p13.c */
 2
 3 /* this program demonstrates some of the characteristics */
 4 /* of variable scoping in C.                             */
 5
 6 /* The scope of y and z is all lines that follow their   */
 7 /* definitions.  Thus z may be used in f1 and f2         */
 8 /* but y may be used only in f2                          */
 9
10 int z = 12;
11
12 int f1(
13 int x)
14 {
15    x = x + y + z;
16    return(x);
17 }
18
19 int y = 11;
20
21 int f2(
22 int x)
23 {
24    x = x + y + z;
25
26 }
27
```

```
class/215/examples ==> gcc p13.c
p13.c: In function `f1':
p13.c:15: `y' undeclared (first use in this function)
p13.c:15: (Each undeclared identifier is reported only once
p13.c:15: for each function it appears in.)
p13.c: At top level:
p13.c:19: `y' used prior to declaration
```

Overlapping scope

Example program p14.c illustrates that multiple declarations of a variable having a single name is legal and results in overlapping scope.

In this program there do exist *three different variables named y*. When the program accesses *y* which *y* is used is governed by the innermost definition rule.

```
/* p14.c */

/* This program illustrates that multiple different */
/* declarations of a variable having the same name   */
/* may have overlapping scope.                       */

int y = 11;

int main( )
{
   int y = 12;

   if (1)
   {
      int y, z;
      y = 92;
     printf(“inner y = %d \n”, y);
   }
   printf("middle y = %d \n", y);
}
```

```
class/215/examples ==> p14
inner y = 92
middle y = 12
```

For sane debugging *never* use multiple variables wit h the *same name*and *overlapping scope*.

Note that if you always call your loop counter variable *i* and you always declare it at the *start of each function,* you are not violating this guidelines. In this case there are multiple *i'*but their scopes *don't overlap.*

Storage class

The *storage class* of a variable the area of memory in which a variable is stored.

The two available areas are commonly referred to as the *heap*and the *stack*.

Stack resident variables include:

> parameters passed to functions
> variables declared inside basic blocks that are *not* declared *static*
> memory areas dynamically allocated with *alloca()*

Heap resident variables include:

> variables declared outside all functions
> variables declared inside basic blocks that are declared *static*.
> memory areas dynamically allocated with *malloc()*

Storage for heap resident variables is assigned at the time a program is loaded and remain assigned for the life of a program.

Stack resident variables are created at entry to the basic block that contains them and deleted at exit from the block.

Non-Persistence of stack resident variables

The example below appears to contradict the claim that storage is assigned to a stack resident variable only for the time in which the block is active.

> At first entry to *f1()* the variable *x* is unintialized.
> The variable *x* is set to 55 before returning from *f1()*
> The variable is still 55 at entry on the second call.

```
class/215/examples ==> cat p15.c
/* p15.c */

void f1(void)
{
   int x;

   printf("At entry to f1 x = %d \n", x);

   x = 55;
}

main()
{
   f1();
   f1();
}
```

```
class/215/examples ==> p15
At entry to f1 x = -1073743532
At entry to f1 x = 55
```

Example p16.c shows that the claim was indeed true and it was only *bad* luck that made it appear otherwise.

```
/* p16.c */

void f1(void)
{
   int x;

   printf("At entry to f1 x = %d \n", x);
   x = 55;
}

void f2(void)
{
   int z;

   printf("At entry to f2 z = %d \n", z);
   z = 102;
}

main()
{
   f1();
   f2();
   f1();
}
```

```
class/215/examples ==> p16
At entry to f1 x = -1073743644
At entry to f2 z = 55
At entry to f1 x = 102
```

It can be observed from the output above that in this particular case the variable *y* in *f1()* and *z* in *f2()* are in fact occupying the *same physical storage*.

Details of stack allocation

```
/* p27.c */

int adder(
int a,
int b)
{
   int d;
   int e;
   d = a + b;
   e = d - a;
   return(d);
}
```

At entry to the function *adder* the stack is organized as follows

```
Parm - 2 (b)
Parm - 1 (a)
Return address   <- SP
```

The compiler produces a prologue to the body of the function which looks like. The *ebp* register is known as the *base pointer* or the *frame pointer*. All stack resident variables are addressed using base/displacement addressing with *ebp* serving as the base.

```
adder:
        pushl   %ebp                ;save caller's frame ptr
        movl    %esp, %ebp          ;set up my frame pointer
        subl    $8, %esp            ;allocate local vars
```

After the prolog completes the stack looks as follows

```
(ebp + 12) Parm - 2 (b)
(ebp +  8) Parm - 1 (a)
(ebp +  4) Return address
(ebp +  0) Saved ebp         <- BP
(ebp -  4) local var (d)
(ebp -  8) local var (e)     <- SP
```

d = a + b;

```
movl    12(%ebp), %eax    ;load b into eax
addl    8(%ebp), %eax     ;add a  to eax
movl    %eax, -4(%ebp)    ;store sum at d
```

e = d - a;

```
movl    8(%ebp), %edx     ;load a into edx
movl    -4(%ebp), %eax    ;load d into eax
subl    %edx, %eax        ;subtract a from d
movl    %eax, -8(%ebp)    ;save result in e
```

return(d);

```
movl    -4(%ebp), %eax    ;copy d to return reg
leave                     ;Copies EBP to ESP then POPS
EBP
```

After the *leave* executes

```
(ebp + 12) Parm - 2 (b)
(ebp +  8) Parm - 1 (a)
(ebp +  4) Return address   <- SP

ret                         ;POPS EIP
```

The *static* and *extern* modifiers

The *static* and *extern* modifiers are used as follows:

```
static int num;
extern int extnum;
```

The action of the *static* modifier is dependent upon the location of the declaration. When used inside the body of a function, *static* forces the variable to

1 - reside on the *heap* instead of the *stack* and thus

2 - safely retain its value across function calls

```
int public_val;
int adder(int a)
{
    static int sum;
    sum += a;
    return(sum);
}
```

The *extern* modifier can be used to access a public variable that is declared in another *source module.* If, in another module, I need to access the variable *public_val* that is declared in the module above I can declare.

```
extern int public_val;
main()
{
    public_val = 15;
}
```

Use of *static* on variables declared outside function bodies

The action of the *static* modifier is dependent upon the location of the declaration. When used outside the body of a function as in declaration of *private_val*, *static*

1- limits the scope of the variable to *this* source module.

2 - but the variable still remains on the *heap.*

```
static int private_val;
int adder(int a)
{
    static int sum;
    sum += a;
    return(sum);
}
```

This will defeat the ability of *extern* modifier to access the variable.

```
extern int private_val;

main()
{
    private_val = 15;
}
```

The two source files *will* compile correctly but the linker *ld* will fail because p34.c no longer publishes the address of *private_val;*

```
class/215/examples ==> gcc p34.c p35.c
/tmp/ccNrTn6K.o(.text+0x12): In function `main':
: undefined reference to `private_val'
collect2: ld returned 1 exit status
class/215/examples ==>
```