Variable Scope & Storage

Outline

1. Variable scope
   - This is spatial (space)

2. Storage class
   - This is temporal (time)

1. Variable scope (範圍, 領域)

- The *scope* of a variable determines
  where the variable is accessible or usable in a program.

- In C language, scope depends on the notion of *blocks*
  – Each { … } defines a block

- Basic scope rule (in C language):

Variables are accessible only within the block in which they are declared

1.1. Local scope (or Block scope)

```
1   void foo( int p )
2   {
3       int q ;
4       ...
5   }
6
7   int main( void )
8   {
9       int x ;
10      ...
11      if ( ... )
12      {
13          int y ;
14          ...
15      }
16      return 0 ;
17  }
```

p and q are accessible only inside `foo()`.

Local variables

x is accessible only inside `main()`.

1.1. Local scope (or Block scope)

```
1   void foo( int p )
2   {
3       int q ;
4       ...
5   }
6
7   int main( void )
8   {
9       int x ;
10      ...
11      if ( ... )
12      {
13          int y ;
14          ...
15      }
16      return 0 ;
17  }
```

y is accessible only within the `if`-block

1.1. Local scope (or Block scope)

```
1   void foo( int p )
2   {
3       int q ;
4       printf( "%d" , x );  // Error!
5   }
6
7   int main( void )
8   {
9       int x ;
10      ...
11      if ( ... )
12      {
13          int y ;
14          printf( "%d" , x );  // OK!
15      }
16      printf( "%d" , y );  // Error!
17      return 0 ;
18  }
```

Accessing an identifier outside its scope will result in a compile-time error.

1.1.1. How to make good use of local scope

```
1  ...
2    int a , b ;
3
4    ...
5
6    // When we need a variable temporarily (e.g., to swap
7    // the value between "a" and "b"), we may introduce
8    // a block and declare the "tmp" variable inside.
9    {
10     int tmp ;     // In this way, we make sure "tmp" only
11     tmp = a   ;   // exists in this block and won't introduce
12     a   = b   ;   // a conflicting name by accident in future
13     b   = tmp ;
14   }
15 ...
```

1.2. Global scope (or **File** scope)

```
1   int universe ;
2
3   void foo()
4   {
5      printf( "%d\n" , universe );
6      universe++ ;
7   }
8
9   int main( void )
10  {
11     universe = 1 ;
12     foo() ;
13     printf( "%d\n" , universe );
14     return 0 ;
15  }
```

Variables that are not declared in any function are commonly known as *global variables*. They are accessible anywhere in the same file.

In this example, `universe` is a global variable.

1.3. Masking

```
1   int bar = 0 ;
2
3   void foo()
4   {
5       bar = 1 ;  // which bar?
6   }
7   int main( void )
8   {
9       int bar = 2 ;
10      bar++ ;    // which bar?
11      {
12          int bar = 3 ;
13          printf( "%d\n" , bar );   // which bar?
14
15      }
16      bar-- ;    // which bar?
17      return 0 ;
18  }
```

1.3. Masking

An identifier declared inside a block *masks* or *overshadows* the same identifier (with same name) declared outside the block.

```
1   int bar = 0 ;
2
3   void foo()
4   {
5       bar = 1 ;  // Refer to the global "bar"
6   }
7   int main( void )
8   {
9       int bar = 2 ;
10      bar++ ;    // Refer to the "bar" declared in main()
11      {
12          int bar = 3 ;
13          printf( "%d\n" , bar );   // Refer to the "bar" declared
14                                    // in the current local block
15      }
16      bar-- ;    // Refer to the "bar" declared in main()
17      return 0 ;
18  }
```

1.3. Masking

Since the system will not color each variable for you, **avoid using same variable names**, since it is error-prone and the code will have poor readability

```
1   int bar = 0 ;
2
3   void foo()
4   {
5       bar = 1 ;  // Refer to the global "bar"
6   }
7   int main( void )
8   {
9       int bar = 2 ;
10      bar++ ;    // Refer to the "bar" declared in main()
11      {
12          int bar = 3 ;
13          printf( "%d\n" , bar );   // Refer to the "bar" declared
14                                    // in the current local block
15      }
16      bar-- ;    // Refer to the "bar" declared in main()
17      return 0 ;
18  }
```

Note: You should avoid introducing identifiers that mask other identifiers.

1.4. Why shouldn't use global variables

```
1  #include <stdio.h>
2
3  int universe = -9 ;
4
5  void fcn()
6  {
7     int f ;
8     universe *= 3  ;
9     f          = 99 ;
10 }
11 void fcn2()
12 {
13    double g ;
14    universe -= 40 ;
15    fcn() ;
16    g = universe ;
17 }
```

What's the value of f here?

```
void fcn3()
{
   double h ;
   fcn() ;
   h = universe = 9 ;
   fcn2() ;
}
int main( void )
{
  int m ;
  universe = m = 10 ;
  fcn()  ;
  fcn2() ;
  fcn3() ;
  fcn()  ;
  return 0 ;
}
```

What is the value of universe right after calling fcn3()?

1.4. Why shouldn't use global variables

- Global variable is powerful and handy in C.
- However, we should NOT use it in general.
- When there is something wrong with the value of a *local variable*, we can easily *look for the bug in its local scope*.
- The value of a *global variable* is hard to tell and predict because it can be *modified anywhere in a (large) program in any order*!
- Instead, we should *use parameters and return values to exchange information* between functions.

2. Storage class

- The storage class of a variable can define the

  Life-time of a variable in the computer memory during the program execution.

- Two common types:
  - Automatic
  - Static

2.1. Storage class auto

```
int main( void )
{
   int    a , b , c ;
   double f ;
...
}
```

Equivalent

```
int main( void )
{
   auto int    a , b , c ;
   auto double f ;
...
}
```

- Variables declared within function bodies are *by default automatic*.
- The keyword `auto` is seldom used and can be omitted.

2.1. Storage class `auto`

Automatic Creation / Destruction of **`auto`** variables

- *** When entering a block, memory is allocated for the automatic local variables (*Creation*).
- When exiting a block, the memory set aside for the automatic variables are released (*Destruction*).
  - Thus the values of these variables are lost.
- If the block is *re-entered*, the whole process repeats.
  - But the *values of the variables are unknown*. Why?

2.2. Storage class `static`

- Variables with “static” storage class have the following characteristics
  - **By default**, they are **assigned to zero** (if you don’t initialize them)
  - **Created and initialized only once**
  - **It stays in the memory** until the program terminates.

```
static int number_hours = 50 ;
```

- All global variables have static storage class.

2.2.1. Local `static` variable

```
1  #include <stdio.h>
2
3  void foo()
4  {
5    static int static_var = 0 ;
6           int auto_var   = 0 ;
7    printf( "static = %d, auto = %d\n" , static_var , auto_var );
8    static_var++ ;
9    auto_var++   ;
10 }
11
12 int main( void )
13 {
14    for ( int i = 0 ; i < 5 ; i++ )
15       foo() ;
16    return 0 ;
17 }
```

```
static = 0, auto = 0
static = 1, auto = 0
static = 2, auto = 0
static = 3, auto = 0
static = 4, auto = 0
```

2.2.1. Local `static` variable

```
1  #include <stdio.h>
2
3  void foo()
4  {
5    static int static_var = 0 ;
6           int auto_var   = 0 ;
7    printf( "static = %d, auto = %d\n" , static_var , auto_var );
8    static_var++ ;
9    auto_var++   ;
10 }
11
12 int main( void )
13 {
14    for ( int i = 0 ; i < 5 ; i++ )
15       foo() ;
16    return 0 ;
17 }
```

`static_var` is created and initialized once per program execution. It can retain its value over function calls.

`auto_var` is created, initialized, and eventually destroyed in each function call to `foo()`.

Think about their life-time!!!

Summary

1. Variable scope (Local vs. Global) – Accessible region

   Note *masking and avoid global variables

2. Storage class (Automatic vs. Static) – Life time
   - auto
   - Static – any application?
     - Count how many times you've called a function: `static int count = 0 ;`
     - In a function, check if this is the first time (or not) it is called: `count == 0`?

   * Note: there are two other keywords for storage class:
   - register and extern

Note: **scope** is about **where we can access** (read + write) a variable and **storage class** is about the variable **memory** and **life-time**