Variable Scope,
Address & Storage

Outline

1. Variable scope
2. Concept of Address
3. Storage class

1. Variable Scope (範圍,領域)

- The *scope* of an variable determines where the variable is accessible or useable in a program.
- In C language, scope rules depend on the notion of *blocks*
  - Each **{ … }** defines a block
- Basic scope rule:

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

1.1. Local Scope (or Block Scope)

```
void foo(int p)
{
  int q;
  ...
}

int main(void)
{
    int x;
    ...
    if (...)
    {
        int y;
        ...
    }
    return 0;
}
```

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

x is only accessible inside `main()`.

1.1. Local Scope (or Block Scope)

```
void foo(int p)
{
  int q;
  ...
}

int main(void)
{
    int x;
    ...
    if (...)
    {
        int y;
        ...
    }
    return 0;
}
```

y is only accessible within the `if`-block

1.1. Local Scope (or Block Scope)

```
void foo(int p)
{
  int q;
  printf("%d", x);  // Error!
}

int main(void)
{
    int x;
    if (...)
    {
        int y;
       printf("%d", x);  // OK!
    }
    printf("%d", y);      // Error!
    return 0;
}
```

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

1.1.1. How to make good use of local scope

```
int A, B;

...

// When we need a variable temporarily (e.g., to
// swap the value between two A and B), we can introduce
// a block and declare the "temp" variable inside.
{
  int tmp;    // This way, we make sure "tmp" only exists
  tmp = A;    // in this block and won't introduce
  A = B;      // a conflicting name by accident.
  B = tmp;
}
```

1.2. Global Scope (File Scope)

```c
int universe;

void foo() {
  printf("%d\n", universe);
  universe++;
}

int main(void) {

    universe = 1;
    foo();
    printf("%d\n", universe);

    return 0;
}
```

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

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

1.3. Masking

```c
int bar = 0;

void foo() {
    bar = 1;    // Refer to the
}               // global "bar"

int main(void) {
    int bar = 2;
    bar++;      // Refer to the "bar" declared in main()
    {
        int bar = 3;
        printf("%d\n", bar);  // Refer to the "bar" declared in the
                              // current block
    }
    bar--;      // Refer to the "bar" declared in main()
    return 0;
}
```

An identifier declared inside a block *masks* or *overshadows* the same identifier declared outside the block.

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

1.4. Why you should not use global variables

```c
#include <stdio.h>
int  universe = -9;
void fcn() {
  int  f;
  universe *= 3;
  f = 99;
}
void fcn2() {
  double g;
  universe -= 40;
  fcn();
  g = universe;
}
```

What’s the value of f here?

Can you tell the value of universe here (right after calling fcn3())?

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

1.4. Why you should not use global variables

- Global variable is a powerful tool available 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 scope*.
- The value of a *global variable* is hard to tell and predict because it can be *modified anywhere in any order*!
- Instead, we should *use parameters and return values to exchange information* between functions.

2. Address

- Identifiers are human friendly names to identify variables or other entities (such as functions) in C.
- The computer, however, access variables via their unique locations in the memory, i.e. their <u>addresses</u>.

2. Address

- The operator **&** allow us to access the address of a variable in C during run-time.
  - Does that look familiar to you?
- We can also use **%p** to help us print out addresses as a hexadecimal number.

e.g.
```
int x = 0;
printf("%p\n",&x);
```

2. Address

```c
#include <stdio.h>

int x = 0;

void foo() {
   printf("Address 2: %p\n",&x);
}

int main(void) {
   int x = 0;
   printf("Address 1: %p\n",&x);
   if (x == 0) {
      int x = 10;
      foo();
      printf("Address 3: %p\n",&x);
   }
}
```

You have just learned about scope and masking. Notice here we have three different **x** (green, orange and blue). What do you expect the output to be?

Also, have you tried running the program multiple times?

Note: For illustration only. Avoid introducing identifiers that mask other identifiers in real programs.

2.1. Address of Arrays

- We can also use the operator **&** to print out the addresses of array elements.
- If you recall, an array in C was introduced to you as a continuous block of memory. What will be their address be like?

2.1. Addresses of Arrays

```c
#include <stdio.h>

int main(void) {
   int i, myArray[10] = {0};
   for (i=0;i<10;i++) {
      printf("%p\n",&myArray[i]);
   }
   return 0;
}
```

Do you see a pattern in the output?

The addresses will differ by 4 (on repl.it). Why do you think it's 4?

Possible output:

```
0x7fffa4d1ac60
0x7fffa4d1ac64
0x7fffa4d1ac68
0x7fffa4d1ac6c
0x7fffa4d1ac70
0x7fffa4d1ac74
0x7fffa4d1ac78
0x7fffa4d1ac7c
0x7fffa4d1ac80
0x7fffa4d1ac84
```

3. Storage Class

- The storage class of a variable determines how the variable's storage (in the computer memory) are managed during program execution.

- Two common types:
  – Automatic
  – Static

3.1. The 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.

3.1. The 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?

3.2. The Storage Class `static`

- A variable with static storage class has the following characteristics
  - It is created and initialized to zero right before the program execution begins.
  - It stays in the memory until the program terminates.

- All global variables have static storage class.

3.2.1. Local `static` Variable

```c
#include <stdio.h>

void foo() {
  static  int  static_var = 0;
          int  auto_var = 0;
  printf("static = %d, auto = %d\n", static_var, auto_var);
  static_var++;
  auto_var++;
}

int main(void) {
  int i;
  for (i = 0; i < 5; i++)
    foo();

  return 0;
}
```

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

3.2.1. Local `static` Variable

```c
#include <stdio.h>

void foo() {
  static  int  static_var = 0;
          int  auto_var = 0;
  printf("static = %d, auto = %d\n", static_var, auto_var);
  static_var++;
  auto_var++;
}

int main(void) {
  int i;
  for (i = 0; i < 5; i++)
    foo();

  return 0;
}
```

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

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

Summary

1. Variable scope (Local vs. Global)

2. Address

3. Storage class (Automatic vs. Static)

Reading Assignment

- C: How to Program, 8th ed, Deitel and Deitel
- Chapter 5 C Functions
  - Section 5.12: Storage Classes
  - Section 5.13: Scope Rules