Chapter 10

Program Organization

Local Variables

- A variable declared in the body of a function is said to be ***local*** to the function:

```
int sum_digits(int n)
{
  int sum = 0;    /* local variable */

  while (n > 0) {
    sum += n % 10;
    n /= 10;
  }

  return sum;
}
```

Local Variables

- Default properties of local variables:
  - ***Automatic storage duration.*** Storage is "automatically" allocated when the enclosing function is called and deallocated when the function returns.
  - ***Block scope.*** A local variable is visible from its point of declaration to the end of the enclosing function body.

Static Local Variables

- Including `static` in the declaration of a local variable causes it to have ***static storage duration.***
- A variable with static storage duration has a permanent storage location, so it retains its value throughout the execution of the program.
- Example:

```
void f(void)
{
  static int i;    /* static local variable */
  …
}
```

- A static local variable still has block scope, so it's not visible to other functions.

Static Local Variables: Example

- What will be the value of `f(10)` if `f` has never been called before? What will be the value of `f(10)` if `f` has been called five times previously?

```
int f(int i)
{
    static int j = 0;
    return i * j++;
}
```

Answers: 0, 50

Parameters

- Parameters have the same properties—automatic storage duration and block scope—as local variables.
- Each parameter is initialized automatically when a function is called (by being assigned the value of the corresponding argument).

External Variables

- Passing arguments is one way to transmit information to a function.
- Functions can also communicate through ***external variables***—variables that are declared outside the body of any function.
- External variables are sometimes known as ***global variables.***

External Variables

- Properties of external variables:
  - Static storage duration
  - File scope
- Having ***file scope*** means that an external variable is visible from its point of declaration to the end of the enclosing file.

Pros and Cons of External Variables

- External variables are convenient when many functions must share a variable or when a few functions share a large number of variables.
- In most cases, it's better for functions to communicate through parameters rather than by sharing variables:
  - If we change an external variable during program maintenance (by altering its type, say), we'll need to check every function in the same file to see how the change affects it.
  - If an external variable is assigned an incorrect value, it may be difficult to identify the guilty function.
  - Functions that rely on external variables are hard to reuse in other programs.

Pros and Cons of External Variables

- Don't use the same external variable for different purposes in different functions.
- Suppose that several functions need a variable named `i` to control a `for` statement.
- Instead of declaring `i` in each function that uses it, some programmers declare it just once at the top of the program.
- This practice is misleading; someone reading the program later may think that the uses of `i` are related, when in fact they're not.

Pros and Cons of External Variables

- Make sure that external variables have meaningful names.
- Local variables don't always need meaningful names: it's often hard to think of a better name than `i` for the control variable in a `for` loop.

Pros and Cons of External Variables

- Making variables external when they should be local can lead to some rather frustrating bugs.
- Code that is supposed to display a 10 × 10 arrangement of asterisks:

```
int i;

void print_one_row(void)
{
  for (i = 1; i <= 10; i++)
    printf("*");
}

void print_all_rows(void)
{
  for (i = 1; i <= 10; i++) {
    print_one_row();
    printf("\n");
  }
}
```

- Instead of printing 10 rows, `print_all_rows` prints only one.

Blocks

- In Section 5.2, we encountered compound statements of the form

  { *statements* }

- C allows compound statements to contain declarations as well as statements:

  { *declarations* *statements* }

- This kind of compound statement is called a ***block*.**

Blocks

- Example of a block:

```
if (i > j) {
  /* swap values of i and j */
  int temp = i;
  i = j;
  j = temp;
}
```

Blocks

- By default, the storage duration of a variable declared in a block is automatic: storage for the variable is allocated when the block is entered and deallocated when the block is exited.
- The variable has block scope; it can’t be referenced outside the block.
- A variable that belongs to a block can be declared `static` to give it static storage duration.

Blocks

- The body of a function is a block.
- Blocks are also useful inside a function body when we need variables for temporary use.
- Advantages of declaring temporary variables in blocks:
  - Avoids cluttering declarations at the beginning of the function body with variables that are used only briefly.
  - Reduces name conflicts.

Scope

- In a C program, the same identifier may have several different meanings.
- C's scope rules enable the programmer (and the compiler) to determine which meaning is relevant at a given point in the program.
- The most important scope rule: When a declaration inside a block names an identifier that's already visible, the new declaration temporarily "hides" the old one, and the identifier takes on a new meaning.
- At the end of the block, the identifier regains its old meaning.

Scope

- In the example on the next slide, the identifier `i` has four different meanings:
  - In Declaration 1, `i` is a variable with static storage duration and file scope.
  - In Declaration 2, `i` is a parameter with block scope.
  - In Declaration 3, `i` is an automatic variable with block scope.
  - In Declaration 4, `i` is also automatic and has block scope.
- C's scope rules allow us to determine the meaning of `i` each time it's used (indicated by arrows).

```
int i ;                  /* Declaration 1 */

void f(int i )           /* Declaration 2 */
{
  i = 1;
}

void g(void)
{
  int i = 2;             /* Declaration 3 */

  if (i > 0) {
    int i ;              /* Declaration 4 */

    i = 3;
  }

  i = 4;
}

void h(void)
{
  i = 5;
}
```