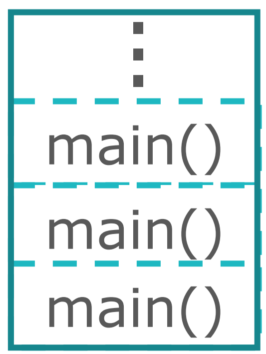
Lecture 6 - Function and Scope

Meng-Hsun Tsai
CSIE, NCKU

```
int main ( )
{

  return main();
}
```



6.1 Defining and Calling Functions

Function

- A function is a series of statements that have been grouped together and given a name.
- Each function is essentially a small program, with its own declarations and statements.
- Advantages of functions:
  - A program can be divided into small pieces that are easier to understand and modify.
  - We can avoid duplicating code that's used more than once.
  - A function that was originally part of one program can be reused in other programs.

Function Definitions

- General form of a ***function definition:***

return-type function-name (*parameters*)
{
    declarations
    statements
}

No parameter

```
int main ( )
{
    return 0;
}
```

No declaration

Function Definitions (cont.)

- The return type of a function is the type of value that the function returns.
- Rules governing the return type:
  - Functions may not return arrays.
  - Specifying that the return type is `void` indicates that the function doesn't return a value.

```
void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
}
```

Function Definitions (cont.)

- After the function name comes a list of parameters.
- Each parameter is preceded by a specification of its type; parameters are separated by commas.

```
double average(double a, double b, double c ) { … }
```

- If the function has no parameters, the word `void` should appear between the parentheses.

```
int main(void) { … }
```

```
void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
}
```

IMSLab Since 2010

Function Definitions (cont.)

- The body of a function may include both declarations and statements.
- Variables declared in the body of a function can't be examined or modified by other functions.
- Variable declarations and statements can be mixed, as long as each variable is declared prior to the first statement that uses the variable.

```
double average(double a, double b)
{
  double sum;         /* declaration */

  sum = a + b;        /* statement */
  return sum / 2;     /* statement */
}
```

Function Definitions (cont.)

- The body of a function whose return type is `void` (a "`void` function") can be empty:

```
void print_pun(void)
{
}
```

- Leaving the body empty may make sense as a temporary step during program development.

Function Calls

- A function call consists of a function name followed by a list of arguments, enclosed in parentheses:

```
average(x, y)
print_count(i)
print_pun()
```

- If the parentheses are missing, the function won't be called:

```
print_pun;    /*** WRONG ***/
```

This statement is legal but has no effect.

Function Calls (cont.)

- A call of a `void` function is always followed by a semicolon to turn it into a statement:

```
print_count(i);
print_pun();
```

- A call of a non-`void` function produces a value that can be stored in a variable, tested, printed, or used in some other way:

```
avg = average(x, y);
if (average(x, y) > 0)
  printf("Average is positive\n");
printf("The average is %g\n", average(x, y));
```

IMSLab since 2010

Function Calls (cont.)

- The value returned by a non-`void` function can always be discarded if it's not needed:

```
average(x, y);  /* discards return value */
```

- Ignoring the return value of `average` is an odd thing to do, but for some functions it makes sense.
- `printf` returns the number of characters that it prints.
- After the following call, `num_chars` will have the value 9:

```
num_chars = printf("Hi, Mom!\n");
```

- We normally discard `printf`'s return value:

```
printf("Hi, Mom!\n"); /* discards return value */
```

Program #1: Computing Averages

- A function named `average` that computes the average of two `double` values:

```
double average(double a, double b)
{
  return (a + b) / 2;
}
```

- The word `double` at the beginning is the ***return type*** of `average`.
- The identifiers `a` and `b` (the function's ***parameters***) represent the numbers that will be supplied when `average` is called.

Program #1: Computing Averages (cont.)

- Every function has an executable part, called the ***body***, which is enclosed in braces.
- The body of `average` consists of a single `return` statement.
- Executing this statement causes the function to "return" to the place from which it was called; the value of `(a + b) / 2` will be the value returned by the function.

Program #1: Computing Averages (cont.)

- A function call consists of a function name followed by a list of ***arguments***.
  - `average(x, y)` is a call of the `average` function.
- Arguments are used to supply information to a function.
  - The call `average(x, y)` causes the values of `x` and `y` to be copied into the parameters `a` and `b`.
- An argument doesn't have to be a variable; any expression of a compatible type will do.
  - `average(5.1, 8.9)` and `average(x/2, y/3)` are legal.

Program #1: Computing Averages (cont.)

- We'll put the call of `average` in the place where we need to use the return value.
- A statement that prints the average of `x` and `y`:

  ```
  printf("Average: %g\n", average(x, y));
  ```

  The return value of `average` isn't saved; the program prints it and then discards it.

- If we had needed the return value later in the program, we could have captured it in a variable:

  ```
  avg = average(x, y);
  ```

Program #1: Computing Averages (cont.)

- The `average.c` program reads three numbers and uses the `average` function to compute their averages, one pair at a time:

```
Enter three numbers: 3.5 9.6 10.2
Average of 3.5 and 9.6: 6.55
Average of 9.6 and 10.2: 9.9
Average of 3.5 and 10.2: 6.85
```

Program #1: Computing Averages (cont.)

average.c

```
#include <stdio.h>

double average(double a, double b)
{
  return (a + b) / 2;
}

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}
```

Program #2: Printing a Countdown

- To indicate that a function has no return value, we specify that its return type is `void`:

```
void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}
```

- `void` is a type with no values.
- A call of `print_count` appears in a statement by itself:

```
print_count(i);
```

- The `countdown.c` program calls `print_count` 10 times inside a loop.

Program #2: Printing a Countdown (cont.)

countdown.c

```
#include <stdio.h>

void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}

int main(void)
{
  int i;

  for (i = 10; i > 0; --i)
    print_count(i);

  return 0;
}
```

Program #3: Testing Whether a Number Is Prime

- The `prime.c` program tests whether a number is prime:

```
Enter a number: 34
Not prime
```

- The program uses a function named `is_prime` that returns `true` if its parameter is a prime number and `false` if it isn't.

- `is_prime` divides its parameter `n` by each of the numbers between 2 and the square root of `n`; if the remainder is ever 0, `n` isn't prime.

Program #3: Testing Whether a Number Is Prime (cont.)

prime.c

```
#include <stdbool.h>
#include <stdio.h>

bool is_prime(int n)
{
  int divisor;

  if (n <= 1)
    return false;

  for (divisor = 2; divisor *
       divisor <= n; divisor++)
    if (n % divisor == 0)
      return false;
  return true;
}
```

```
int main(void)
{
  int n;

  printf("Enter a number: ");
  scanf("%d", &n);
  if (is_prime(n))
    printf("Prime\n");
  else
    printf("Not prime\n");
  return 0;
}
```

Function Declarations

- Either a declaration or a definition of a function must be present prior to any call of the function.
- A ***function declaration*** provides the compiler with a brief glimpse at a function whose full definition will appear later.
- General form of a function declaration:

  return-type function-name (*parameters*) ;
- The declaration of a function must be consistent with the function's definition.
- Here's the `average.c` program with a declaration of `average` added.

Function Declarations (cont.)

```c
#include <stdio.h>

double average(double a, double b);    /* DECLARATION */

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x,
  y));
  printf("Average of %g and %g: %g\n", y, z, average(y,
  z));
  printf("Average of %g and %g: %g\n", x, z, average(x,
  z));

  return 0;
}

double average(double a, double b)     /* DEFINITION */
{
  return (a + b) / 2;
}
```

Function Declarations (cont.)

- Function declarations of the kind we're discussing are known as ***function prototypes*.**
- A function prototype doesn't have to specify the names of the function's parameters, as long as their types are present:

```
double average(double, double);
```

6.2 Arguments

Arguments

- In C, arguments are ***passed by value:*** when a function is called, each argument is evaluated and its value assigned to the corresponding parameter.
- Since the parameter contains a copy of the argument's value, any changes made to the parameter during the execution of the function don't affect the argument.

Arguments (cont.)

- Consider the following function, which raises a number `x` to a power `n`:

```
int power(int x, int n)
{
  int i, result = 1;

  for (i = 1; i <= n; i++)
    result = result * x;

  return result;
}
```

Arguments (cont.)

- Since `n` is a *copy* of the original exponent, the function can safely modify it, removing the need for `i`:

```
int power(int x, int n)
{
  int result = 1;

  while (n-- > 0)
    result = result * x;

  return result;
}
```

Arguments (cont.)

- C's requirement that arguments be passed by value makes it difficult to write certain kinds of functions.
- Suppose that we need a function that will decompose a `double` value into an integer part and a fractional part.
- Since a function can't *return* two numbers, we might try passing a pair of variables to the function and having it modify them:

```
void decompose(double x, long int_part,
               double frac_part)
{
  int_part = (long) x;
  frac_part = x - int_part;
}
```

Arguments (cont.)

- A call of the function:

```
decompose(3.14159, i, d);
```

- Unfortunately, `i` and `d` won't be affected by the assignments to `int_part` and `frac_part`.

Argument Conversions

- C allows function calls in which the types of the arguments don't match the types of the parameters.
- ***Note that the compiler has encountered a prototype prior to the call.***
- The value of each argument is implicitly converted to the type of the corresponding parameter as if by assignment.
- Example: If an `int` argument is passed to a function that was expecting a `double`, the argument is converted to `double` automatically.

Array Arguments

- When a function parameter is a one-dimensional array, the length of the array can be left unspecified:

```
int f(int a[])  /* no length specified */
{
   …
}
```

- C doesn't provide any easy way for a function to determine the length of an array passed to it.
- Instead, we'll have to supply the length—if the function needs it—as an additional argument.

Array Arguments (cont.)

- Example:

```
int sum_array(int a[], int n)
{
  int i, sum = 0;

  for (i = 0; i < n; i++)
    sum += a[i];

  return sum;
}
```

- Since `sum_array` needs to know the length of `a`, we must supply it as a second argument.

Array Arguments (cont.)

- The prototype for `sum_array` has the following appearance:

```
int sum_array(int a[], int n);
```

- As usual, we can omit the parameter names if we wish:

```
int sum_array(int [], int);
```

Array Arguments (cont.)

- When `sum_array` is called, the first argument will be the name of an array, and the second will be its length:

```
#define LEN 100

int main(void)
{
  int b[LEN], total;
  …
  total = sum_array(b, LEN);
  …
}
```

- Notice that we don't put brackets after an array name when passing it to a function:

```
total = sum_array(b[], LEN);   /*** WRONG ***/
```

Array Arguments (cont.)

- A function has no way to check that we've passed it the correct array length.
- Suppose that we've only stored 50 numbers in the `b` array, even though it can hold 100.
- We can sum just the first 50 elements by writing

```
total = sum_array(b, 50);
```

- Be careful not to tell a function that an array argument is *larger* than it really is:

```
total = sum_array(b, 150);    /*** WRONG ***/
```

`sum_array` will go past the end of the array, causing undefined behavior.

Array Arguments (cont.)

- A function is allowed to change the elements of an array parameter, and the change is reflected in the corresponding argument.
- A function that modifies an array by storing zero into each of its elements:

```
void store_zeros(int a[], int n)
{
  int i;
  for (i = 0; i < n; i++)
    a[i] = 0;
}
```

```
store_zeros(b, 100);
```

Array Arguments (cont.)

- If a parameter is a multidimensional array, only the length of the first dimension may be omitted.
- If we revise `sum_array` so that `a` is a two-dimensional array, we must specify the number of columns in `a`:

```
#define LEN 10

int sum_two_dimensional_array(int a[][LEN], int n)
{
  int i, j, sum = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < LEN; j++)
      sum += a[i][j];

  return sum;
}
```

6.3 Function and Program Termination

The `return` Statement

- A non-`void` function must use the `return` statement to specify what value it will return.
- The `return` statement has the form

  `return` *expression* `;`

- The expression is often just a constant or variable:

```
return 0;
return status;
```

- More complex expressions are possible:

```
return n >= 0 ? n : 0;
```

The `return` Statement (cont.)

- If the type of the expression in a `return` statement doesn't match the function's return type, the expression will be implicitly converted to the return type.
  - If a function returns an `int`, but the `return` statement contains a `double` expression, the value of the expression is converted to `int`.

```
int average(double a, double b)
{
  return (a + b)/2;
}
```

double

The **`return`** Statement (cont.)

- `return` statements may appear in functions whose return type is `void`, provided that no expression is given:

```
return;   /* return in a void function */
```

- Example:

```
void print_int(int i)
{
  if (i < 0)
    return;
  printf("%d", i);
}
```

The `return` Statement (cont.)

- A `return` statement may appear at the end of a `void` function:

```
void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
  return;    /* OK, but not needed */
}
```

  Using `return` here is unnecessary.

- If a non-`void` function fails to execute a `return` statement, the behavior of the program is undefined if it attempts to use the function's return value.

Program Termination

- Normally, the return type of `main` is `int`:

```
int main(void)
{
   …
}
```

- Omitting the word `void` in `main`'s parameter list remains legal, but—as a matter of style—it's best to include it.

Program Termination (cont.)

- The value returned by `main` is a status code that can be tested when the program terminates.
- `main` should return 0 if the program terminates normally.
- To indicate abnormal termination, `main` should return a value other than 0.
- It's good practice to make sure that every C program returns a status code.

The `exit` Function

- Executing a `return` statement in `main` is one way to terminate a program.
- Another is calling the `exit` function, which belongs to `<stdlib.h>`.
- The argument passed to `exit` has the same meaning as `main`'s return value: both indicate the program's status at termination.
- To indicate normal termination, we pass 0:

```
exit(0);    /* normal termination */
```

The `exit` Function (cont.)

- Since 0 is a bit cryptic, C allows us to pass `EXIT_SUCCESS` instead (the effect is the same):

```
exit(EXIT_SUCCESS);
```

- Passing `EXIT_FAILURE` indicates abnormal termination:

```
exit(EXIT_FAILURE);
```

- `EXIT_SUCCESS` and `EXIT_FAILURE` are macros defined in `<stdlib.h>`.
- The values of `EXIT_SUCCESS` and `EXIT_FAILURE` are implementation-defined; typical values are 0 and 1, respectively.

The `exit` Function (cont.)

- The statement

  `return` *expression*;

  in `main` is equivalent to

  `exit(`*expression*`);`

- The difference between `return` and `exit` is that `exit` causes program termination regardless of which function calls it.
- The `return` statement causes program termination only when it appears in the `main` function.

6.4 Scope

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.

Local Variables

- Since C doesn't require variable declarations to come at the beginning of a function, it's possible for a local variable to have a very small scope:

```
void f(void)
{
  ...
  int i;
  ...
}
```

scope of i

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.

IMSLab since 2010

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.***

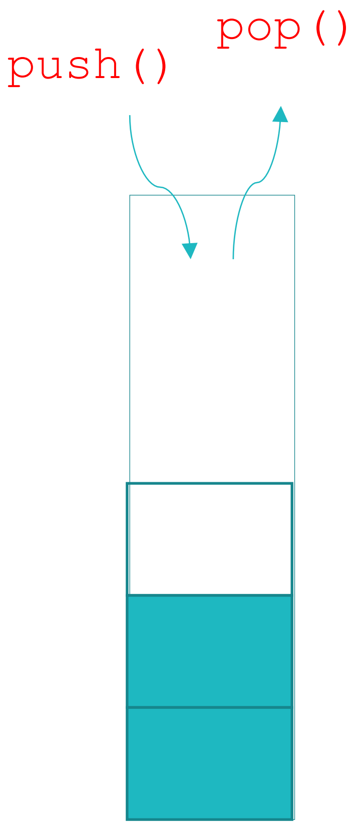
External Variables

- 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.

IMSLab since 2010

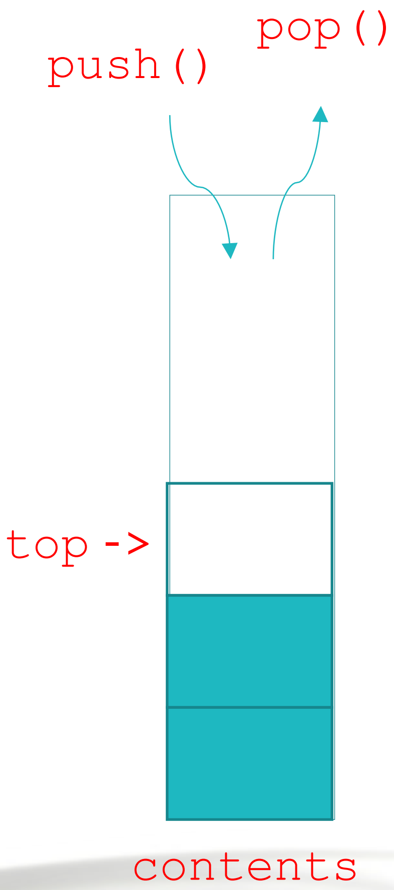
Example: Using External Variables to Implement a Stack

- To illustrate how external variables might be used, let's look at a data structure known as a ***stack.***
- A stack, like an array, can store multiple data items of the same type.
- The operations on a stack are limited:
  - ***Push*** an item (add it to one end—the "stack top")
  - ***Pop*** an item (remove it from the same end)
- Examining or modifying an item that's not at the top of the stack is forbidden.



Example: Using External Variables to Implement a Stack (cont.)

- One way to implement a stack in C is to store its items in an array, which we'll call `contents`.
- A separate integer variable named `top` marks the position of the stack top.
  - When the stack is empty, `top` has the value 0.
- To *push* an item: Store it in `contents` at the position indicated by `top`, then increment `top`.
- To *pop* an item: Decrement `top`, then use it as an index into `contents` to fetch the item that's being popped.



Example: Using External Variables to Implement a Stack (cont.)

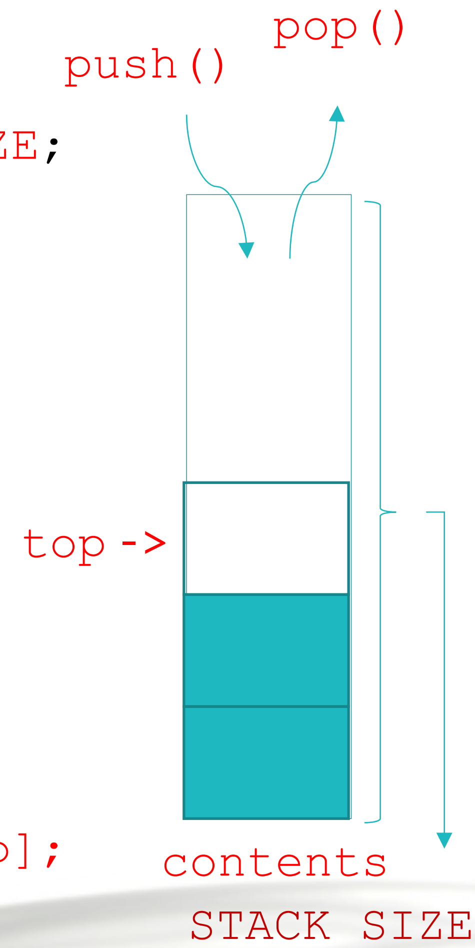
- The following program fragment declares the `contents` and `top` variables for a stack.
- It also provides a set of functions that represent stack operations.
- All five functions need access to the `top` variable, and two functions need access to `contents`, so `contents` and `top` will be external.

Example: Using External Variables to Implement a Stack (cont.)

stack.c

```c
#include <stdbool.h>

#define STACK_SIZE 100

/* external variables */
int contents[STACK_SIZE];
int top = 0;

void make_empty(void)
{
  top = 0;
}

bool is_empty(void)
{
  return top == 0;
}
```

```c
bool is_full(void)
{
  return top == STACK_SIZE;
}

void push(int i)
{
  if (is_full())
    stack_overflow();
  else
    contents[top++] = i;
}

int pop(void)
{
  if (is_empty())
    stack_underflow();
  else
    return contents[--top];
}
```



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.

IMSLab Since 2010

Pros and Cons of External Variables (cont.)

- 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 (cont.)

- 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 (cont.)

- 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.

Program: Guessing a Number

- The `guess.c` program generates a random number between 1 and 100, which the user attempts to guess in as few tries as possible:

```
Guess the secret number between 1 and 100.

A new number has been chosen.
Enter guess: 55
Too low; try again.
Enter guess: 65
Too high; try again.
Enter guess: 60
Too high; try again.
Enter guess: 58
You won in 4 guesses!
```

IMSLab Since 2010

Program: Guessing a Number (cont.)

```
Play again? (Y/N) y

A new number has been chosen.
Enter guess: 78
Too high; try again.
Enter guess: 34
You won in 2 guesses!

Play again? (Y/N) n
```

- Tasks to be carried out by the program:
  - Initialize the random number generator
  - Choose a secret number
  - Interact with the user until the correct number is picked
- Each task can be handled by a separate function.

Program: Guessing a Number (cont.)

guess.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_NUMBER 100

/* external variable */
int secret_number;

/* prototypes */
void initialize_number_generator(void);
void choose_new_secret_number(void);
void read_guesses(void);
```

```
void initialize_number_generator(void)
{
  srand((unsigned) time(NULL));
}
void choose_new_secret_number(void)
{
  secret_number = rand() % MAX_NUMBER + 1;
}
```

Program: Guessing a Number (cont.)

```
int main(void)
{
  char command;
  printf("Guess the secret number between 1 and %d.\n\n",
         MAX_NUMBER);
  initialize_number_generator();
  do {
    choose_new_secret_number();
    printf("A new number has been chosen.\n");
    read_guesses();
    printf("Play again? (Y/N) ");
    scanf(" %c", &command);
    printf("\n");
  } while (command == 'y' || command == 'Y');

  return 0;
}
```

Program: Guessing a Number (cont.)

```
void read_guesses(void)
{
  int guess, num_guesses = 0;

  for (;;) {
    num_guesses++;
    printf("Enter guess: ");
    scanf("%d", &guess);
    if (guess == secret_number) {
      printf("You won in %d guesses!\n\n", num_guesses);
      return;
    } else if (guess < secret_number)
      printf("Too low; try again.\n");
    else
      printf("Too high; try again.\n");
  }
}
```

Program: Guessing a Number (cont.)

- Although `guess.c` works fine, it relies on the external variable `secret_number`.
- By altering `choose_new_secret_number` and `read_guesses` slightly, we can move `secret_number` into the `main` function.
- The new version of `guess.c` follows, with changes in **bold.**

Program: Guessing a Number (cont.)

guess2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_NUMBER 100

/* external variable */
int secret_number;

/* prototypes */
void initialize_number_generator(void);
int new_secret_number(void);
void read_guesses(int secret_number);
```

```
void initialize_number_generator(void)
{
  srand((unsigned) time(NULL));
}
int new_secret_number(void)
{
  return rand() % MAX_NUMBER + 1;
}
```

Program: Guessing a Number (cont.)

```
int main(void)
{
  char command;
  int secret_number;

  printf("Guess the secret number between 1 and %d.\n\n",
         MAX_NUMBER);
  initialize_number_generator();
  do {
    secret_number = new_secret_number();
    printf("A new number has been chosen.\n");
    read_guesses(secret_number);
    printf("Play again? (Y/N) ");
    scanf(" %c", &command);
    printf("\n");
  } while (command == 'y' || command == 'Y');

  return 0;
}
```

Program: Guessing a Number (cont.)

```
void read_guesses(int secret_number)
{
  int guess, num_guesses = 0;

  for (;;) {
    num_guesses++;
    printf("Enter guess: ");
    scanf("%d", &guess);
    if (guess == secret_number) {
      printf("You won in %d guesses!\n\n", num_guesses);
      return;
    } else if (guess < secret_number)
      printf("Too low; try again.\n");
    else
      printf("Too high; try again.\n");
  }
}
```

Blocks

- In Lecture 3, 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.***

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

Blocks (cont.)

- 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 (cont.)

- 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.
- C allows variables to be declared anywhere within a block.

Scope Rules

- In a C program, the same identifier may have several different meanings.
- 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 Rules (cont.)

- 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.

```
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;
}
```

A Quick Review to This Lecture

- General form of a ***function definition:***

  return-type function-name (*parameters*)
  {
  declarations
  statements
  }

- Variables declared in the body of a function can't be examined or modified by other functions.
- Function call: function name followed by arguments in parentheses:

```
average(x, y)
print_count(i)
print_pun()
```

- This statement is legal but has no effect.

  `fun;      // fun()` won't be called

IMSLab Since 2010

A Quick Review to This Lecture (cont.)

- Specifying return type as `void` indicates no return value.

```
void fun(int n){
}
```

- The word `void` is placed in parentheses indicates that a function has no parameters.

```
void fun(void){
}
fun();
```

The parentheses *must* be present.

- Functions may not return arrays.
- General form of a function declaration:

  return-type function-name (*parameters*) ;

- Either a declaration or a definition of a function must be present prior to any call of the function.

A Quick Review to This Lecture (cont.)

- In C, arguments are ***passed by value:*** when a function is called, each argument is evaluated and its value assigned to the corresponding parameter.

- Passing one-dimensional array, length is supplied as second argument:

```
int sum_array(int a[], int n) { … }
```

- Passing two-dimensional array, number of columns must be specified:

```
int sum_two_dimensional_array(int a[][LEN], int n) {…}
```

- The `return` statement has the form

`return` *expression ;*

A Quick Review to This Lecture (cont.)

- The value returned by `main` is a status code that can be tested when the program terminates. (0: normal, non-0: abnormal)
- Program termination
  - `return` statement in `main()`
  - Calling `exit()` in any function

A Quick Review to This Lecture (cont.)

```
int global;
fun ( int parm )
{
   int var;
   static int svar;

   return var;
}
```

- int global; → external variable
- int parm → parameter
- int var; → auto/local variable
- static int svar; → static local variable

Type	Storage duration	Scope
Local variable	Automatic	Block
Parameter	Automaitc	Block
Static local variable	Static	Block
External variable	Static	File