DASF004
Basic and Practice in Programming

Lecture 4

Function

Digital Tattoo

- Our brain treats our body differently
  - Information on the body (or near body) will be processed differently
- Our brain treats tools as our body if it is being extensively used
- What about digital tools?
-
- Using the body as the interface...

Q&A

- Discussion for iCampus
  - For asking questions
  - Not required (you don’t have to participate if you don’t have question)
- Can I submit multiple version?
  - Yes, but only the last version before the deadline will be graded

Agenda

Function
Scope of Variables

Nested Loops (revisit...)

Consider the following code segment:

```
for (int i = 1; i <= 6; i++) {
   for (int j = 1; j <= i; j++) {
      put("*");
   }
   put("\n");
}
```

- What is the output of the code segment???

i = 1; j = 1 → inside loop breaks; outside loop increase by 1
i = 2; j = 1, 2 → inside loop breaks; outside loop increase by 1
i = 3; j = 1, 2, 3 → inside loop breaks; outside loop increase by 1
i = 4; j = 1, 2, 3, 4 → inside loop breaks; outside loop increase by 1
i = 5; j = 1, 2, 3, 4, 5 → inside loop breaks; outside loop increase by 1
i = 6; j = 1, 2, 3, 4, 5, 6 → inside loop breaks; outside loop increase by 1
I = 7; → outside loop breaks

Operator Precedence Rules

<table>
<tr><td>()</td><td>Parenthesis</td></tr>
<tr><td>++ --</td><td>Increment, Decrement</td></tr>
<tr><td>* / %</td><td>Multiplication, Division, Remainder</td></tr>
<tr><td>+ -</td><td>Addition, Subtraction</td></tr>
<tr><td>< <= > >=</td><td>Relational Comparison</td></tr>
<tr><td>!= ==</td><td>Equality</td></tr>
<tr><td>!</td><td>Boolean not</td></tr>
<tr><td>&&</td><td>Boolean and</td></tr>
<tr><td>||</td><td>Boolean or</td></tr>
<tr><td></td><td>Left to Right</td></tr>
<tr><td>=</td><td>Assignment</td></tr>
</table>

Exercise

What is the Boolean Value of the following expression?

```
7 % 2 == 1 and 7 / 3 != 1 or 1 + 1 * 1 == 1
```

Good Programming Style 3.1

Too many levels of nesting can make a program difficult to understand. Try to avoid.

```
for(i=0; i>10; i++)
{ for(j=0; j<20; j++)
  { for(k=10; k<2; k--)
    { for(l=20; l<-20; l--)
      { ......
      }
    }
  }
}
```

Good Programming Style 3.2

When performing arithmetic division to a variable (e.g. 7 / x), explicitly test for this case and handle the potential case of division by zero.

```
if(x != 0)
  y = 7 / x;
else
  printf("Warning: division by zero\n");
```

Good Programming Style 3.3

For novice programmer, you can start writing a loop as a simple loop, then build your code based on the simple loop.

```
for(i=0; i<9; i++)
{ printf("i: %d\n", i);
}
```

Good Programming Style 3.4

When you got lost in coding the loop, print out the counter variable and other intermediate variables so that you know how many times the loop body has executed and the value of each variables.

```
while (counter <= 10)
{ scanf("Input: %f", Input);
  Sum = Sum + Input;
  printf("While loop %d times, Sum=%f\n", counter, Sum);
  counter++;
}
```

Good Programming Style 3.5

Use redundant parentheses in complex calculations to make the expression clearer.

Bad Example:

```
float result = a + b * 2 + c / 3;
if(grade >=60 AND grade <= 70)
```

Good Example:

```
float result = a + (b * 2) + (c / 3);
if((grade >= 60) AND (grade <= 70))
```

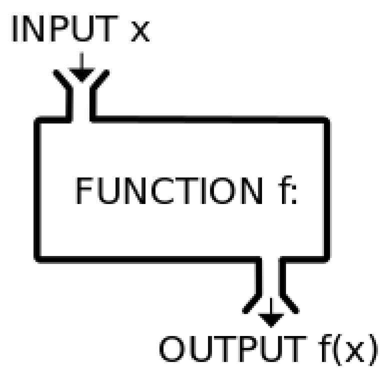
Good Programming Style 3.6

Development your program incrementally.
- Make a functional program first.
- Compile and test it.
- Add a few lines.
- Compile, test, and make sure you get the correct outcome.
- Add a few lines and then test … … … …

Don’t try to write many many lines of code without testing.

Program modules in C

- The functions `printf` and `scanf` that we've used in previous chapters are standard library functions.
- You can <u>write your own functions</u> to define tasks that may be used at many points in a program.
- Functions are like mini-program.
  - Input: what it takes
  - Processing: what it does
  - Output: what it returns



- These are sometimes referred to as <u>programmer-defined functions</u>.
- Functions are invoked by a function call, which specifies the function name and provides information (as arguments) that the called function needs to perform its designated task.
- **Code reuse**
  - Code written can be reused many many times without rewriting the program
  - Code written previously can be reused later on without rewriting the program
  - Code written by others can be used by others

Program modules in C

- A function may call other functions
  - You may not aware of this when you call the function
- We’ll soon see how this “hiding” of implementation details promotes good software engineering.
- Figure 5.1 shows a boss function communicating with several worker functions in a hierarchical manner.

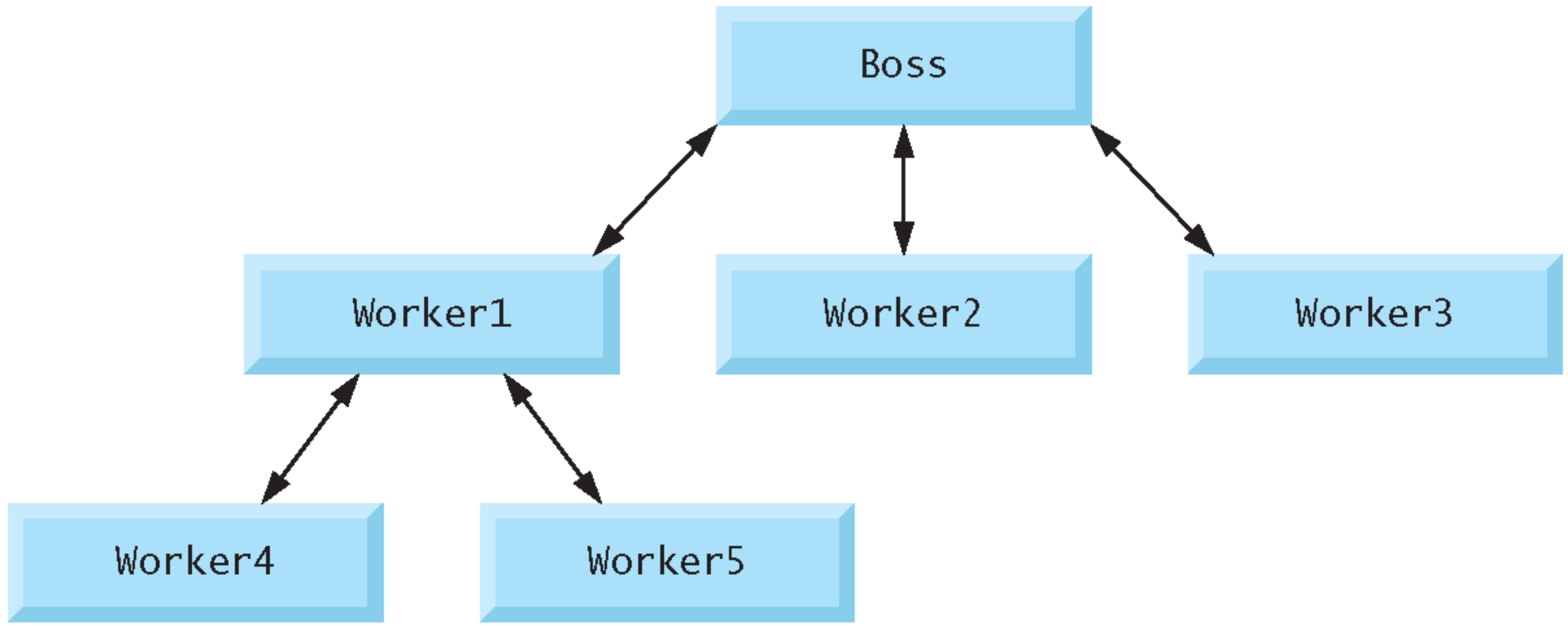


Fig. 5.1 | Hierarchical boss-function/worker-function relationship.

Math library functions

- Math library functions allow you to perform certain common mathematical calculations.
- `#include <math.h>`
- Functions are normally used in a program by writing the name of the function followed by a left parenthesis followed by the argument (or a comma-separated list of arguments) of the function followed by a right parenthesis.
- For example, a programmer desiring to calculate and print the square root of `900.0` you might write

```
printf( "%.2f", sqrt( 900.0 ) );
```

- When this statement executes, the math library function `sqrt` is called to calculate the square root of the number contained in the parentheses (`900.0`).

Math library functions

Function	Description	Example
`sqrt( x )`	square root of x	`sqrt( 900.0 )` is `30.0` `sqrt( 9.0 )` is `3.0`
`cbrt( x )`	cube root of x (C99 and C11 only)	`cbrt( 27.0 )` is `3.0` `cbrt( -8.0 )` is `-2.0`
`exp( x )`	exponential function e^x	`exp( 1.0 )` is `2.718282` `exp( 2.0 )` is `7.389056`
`log( x )`	natural logarithm of x (base e)	`log( 2.718282 )` is `1.0` `log( 7.389056 )` is `2.0`

`log10( x )`	logarithm of x (base 10)	`log10( 1.0 )` is `0.0`
		`log10( 10.0 )` is `1.0`
		`log10( 100.0 )` is `2.0`

`fabs( x )`	absolute value of x as a floating-point number	`fabs( 13.5 )` is `13.5`
		`fabs( 0.0 )` is `0.0`
		`fabs( -13.5 )` is `13.5`

`ceil( x )`	rounds x to the smallest integer not less than x	`ceil( 9.2 )` is `10.0` `ceil( -9.8 )` is `-9.0`

Fig. 5.2 | Commonly used math library functions. (Part 1 of 2.)

Math library functions

Function	Description	Example
`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0` `floor( -9.8 )` is `-10.0`
`pow( x, y )`	*x* raised to power *y* (x^y)	`pow( 2, 7 )` is `128.0` `pow( 9, .5 )` is `3.0`
`fmod( x, y )`	remainder of *x*/*y* as a floating-point number	`fmod( 13.657, 2.333 )` is `1.992`
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0.0`
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0.0`

Fig. 5.2 | Commonly used math library functions. (Part 2 of 2.)

Function definitions

- Each program we've presented has consisted of a function called `main` that called standard library functions to accomplish its tasks.
- We now consider how to write custom functions.
- Consider a program that uses a function `square` to calculate and print the squares of the integers from 1 to 10 (Fig. 5.3).

```c
// Fig. 5.3: fig05_03.c
// Creating and using a programmer-defined function.
#include <stdio.h>

int square( int y ); // function prototype

// function main begins program execution
int main( void )
{
   int x; // counter

   // loop 10 times and calculate and output square of x each time
   for ( x = 1; x <= 10; ++x ) {
      printf( "%d  ", square( x ) ); // function call
   } // end for

   puts( "" );
} // end main

// square function definition returns the square of its parameter
int square( int y ) // y is a copy of the argument to the function
{
   return y * y; // returns the square of y as an int
} // end function square
```

```
1  4  9  16  25  36  49  64  81  100
```

Fig. 5.3 | Creating and using a programmer-defined function. (Part 2 of 2.)

Function definitions

- Function `square` is invoked or called in `main` within the `printf` statement (line 14)

```
printf( "%d  ", square( x ) ); // function call
```

- Function `square` receives a *copy* of the value of `x` in the parameter `y` (line 21).
- Then `square` calculates `y * y`.
- The result is passed back returned to function `printf` in `main` where `square` was invoked (line 14), and `printf` displays the result.
- This process is repeated 10 times using the `for` statement.

Function definitions

- The definition of function `square` shows that `square` expects an integer parameter `y`.
- The keyword `int` preceding the function name (line 21) indicates that `square` *returns* an integer result.
- The `return` statement in `square` passes the value of the expression `y * y` (that is, the result of the calculation) back to the calling function.
- Line 5

```
int square( int y ); // function prototype
```

is a function prototype.

- The `int` in parentheses informs the compiler that `square` expects to *receive* an integer value from the caller.

Function definitions

- Function `square` is invoked or called in `main` within the `printf` statement (line 14)

```
printf( "%d  ", square( x ) ); // function call
```

- Function `square` receives a *copy* of the value of `x` in the parameter `y` (line 21).
- Then `square` calculates `y * y`.
- The result is passed back returned to function `printf` in `main` where `square` was invoked (line 14), and `printf` displays the result.
- This process is repeated 10 times using the `for` statement.

Function definitions

- The `int` to the *left* of the function name `square` informs the compiler that `square` returns an integer result to the caller.
- The compiler refers to the function prototype to check that any calls to `square` (line 14) contain the *correct return type*, the *correct number of arguments*, the *correct argument types*, and that the *arguments are in the correct order*.
- The format of a function definition is

  return-value-type function-name`(` *parameter-list* `)`
  `{`
  definitions
  statements
  `}`

- The *return-value-type* is the data type of the result returned to the caller.
- The *return-value-type* `void` indicates that a function does not return a value.
- The *parameter-list* is a comma-separated list that specifies the parameters received by the function when it's called.
- If a function does not receive any values, *parameter-list* is `void`.

```c
// Fig. 5.4: fig05_04.c
// Finding the maximum of three integers.
#include <stdio.h>

int maximum( int x, int y, int z ); // function prototype

// function main begins program execution
int main( void )
{
   int number1; // first integer entered by the user
   int number2; // second integer entered by the user
   int number3; // third integer entered by the user

   printf( "%s", "Enter three integers: " );
   scanf( "%d%d%d", &number1, &number2, &number3 );

   // number1, number2 and number3 are arguments
   // to the maximum function call
   printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );
} // end main

```

Fig. 5.4 | Finding the maximum of three integers. (Part 1 of 3.)

```
// Function maximum definition
// x, y and z are parameters
int maximum( int x, int y, int z )
{
   int max = x; // assume x is largest

   if ( y > max ) { // if y is larger than max,
      max = y; // assign y to max
   } // end if

   if ( z > max ) { // if z is larger than max,
      max = z; // assign z to max
   } // end if

   return max; // max is largest value
} // end function maximum
```

```
Enter three integers: 22 85 17
Maximum is: 85
```

```
Enter three integers: 47 32 14
Maximum is: 47
```

```
Enter three integers: 35 8 79
Maximum is: 79
```

Fig. 5.4 | Finding the maximum of three integers. (Part 3 of 3.)

Function Prototype

```
#include <stdio.h>

int MyFunction(int input);

int main(void)
{ …
  int x = 89;
  int result = MyFunction(x);
  …
}

int MyFunction(int input)
{ int output = input - 10;
  return output;
}
```

```
#include <stdio.h>

int MyFunction(int input)
{ int output = input - 10;
  return output;
}

int main(void)
{ …
  int x = 89;
  int result = MyFunction(x);
  …
}
```

Scope of Variable

- The scope of a variable is the portion of the program in which the variable can be referenced.
- When you declare a variable, that name and value is only “alive” for some parts of the program
- What is a variable’s scope?
  - Starts at the declaration statement
  - Ends at the end of the block it was declared in
- For example, when we define a local variable in a block, it can be referenced only following its definition in that block or in blocks nested within that block.

Function Scope

- If the variable is declared within a block (compound statement, { }) it only stays alive until the end of the block
- Variables declared in one function can only be referenced within that function

```
#include <stdio.h>

int MyFunction(int input);

int main(void)
{ …
  int x = 89;
  int result = MyFunction(x);
  …
}

int MyFunction(int input)
{ int output = x - 10;
  return output;
}
```

Scope of variable x

Error! Variable x is not defined
in this scope.

Function Scope

- It is OK to define another variable with the same name in different scope (but it could be confusing).

```
#include <stdio.h>

int MyFunction(int input);

int main(void)
{ …
  int x = 89;
  int result = MyFunction(x);
  printf(“X: %d\n”,x);  // X: 89
  …
}
```

Scope of variable x

```
int MyFunction(int input)
{ int output = input - 10;
  int x = 20;
  Printf(“X: %d\n”,x);  // X: 20
  return output;
}
```

Scope of the second variable x

OK to declare x here again,
since it is in a different scope.

Block Scope

- Variables declared in one block can only be referenced within that block

```
#include <stdio.h>

int MyFunction(int input);

int main(void)
{ …
  for(int x; x<=9; x++)
  { printf(“X: %d\n”,x);
  }

  printf(“X: %d\n”,x);  // ERROR! Variable x is not defined in this scope!
  …
}
```

Scope of variable x

Block Scope

- Variables declared in one block can only be referenced within that block
- Block scope ends at the terminating right brace (}) of the block.

```
#include <stdio.h>

int MyFunction(int input);

int main(void)
{ …
  for(int x; x<=9; x++)
  { printf("X: %d\n",x);
  }

  for(int x; x<=19; x++)
  { printf("X: %d\n",x);
  }

}
```

Scope of variable x

Scope of the second variable x

OK to declare x here again,
since it is in different scope

Block Scope in Nested Block

- When blocks are nested, and an identifier in an outer block has the same name as an identifier in an inner block, the identifier in the outer block is "hidden" until the inner block terminates.

```
#include <stdio.h>

int MyFunction(int input);

int main(void)
{ …
  for(int x; x<=9; x++)
  { printf("X: %d\n",x); // Printing out counter variable x
    for(int y; y<=9; y++
    { int x=10;
      printf("X: %d\n",x); // X: 10
    }
  }
}
```

Scope of first variable x

Scope of second variable x

Common Mistakes

Declaring a variable in a for loop then trying to use it after the loop

```
for (int i = 0; i < size; i++)
  { ... //do loop stuff
  }
  System.out.println("Count: " + i);
```

The above is incorrect, since the scope of `i` ends at the end of the for loop.

To correct: declare i before the loop

Correction of above example:

```
int i;
  for (i = 0; i < size; i++)
  { ... //do loop stuff
  }
  System.out.println("Count: " + i);
```

Global Variables

- Global variables can be declared outside any function
- Global variables can be referenced by all scope in the same file

```
#include <stdio.h>

float Pi = 3.1416;

int MyFunction(int input);

int main(void)
{ …
  float radius = 5;
  float cir = 2 * Pi * r;
  …
}

int MyFunction(int input)
{ …
  float radius = 10;
  float area = Pi * r * r;
  …
}
```

Scope of variable Pi

Global Variables

- When the value of a global variables is modified in one function, its value will be updated globally

```
#include <stdio.h>

int x = 3;

int MyFunction(int input);

int main(void)
{ …
  printf("X: %d\n",x);  // X: 3
  MyFunction();
  printf("X: %d\n",x);  // X: 40
  …
}

void MyFunction(void)
{ …
  x = 40;
  printf("X: %d\n",x);  // X: 40
  …
}
```

Scope of variable x

`Static` Variables

- When the variable is declared as static, it exists during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope.

- **<u>At the end of the scope, static variable is not destroyed and its value is retained.</u>**

- Therefore, making local variables static allows them to maintain their values between function calls.

```
#include <stdio.h>

void MyFunction(void);

int main(void)
{ MyFunction();  // X: 4
  MyFunction();  // X: 5
  MyFunction();  // X: 6
  MyFunction();  // X: 7
}

void MyFunction(void)
{ static int x = 3
  x = x + 1;
  printf("X: %d\n",x);
}
```

Scope Rules: Example

```
#include <stdio.h>

void useLocal(void);
void useStaticLocal(void);
void useGlobal(void);

int x = 1; // global variable

int main(void)
{ int x = 5; // local variable to main

  printf(“local x in outer scope of main is %d\n”,x);
  { // start new scope
    int x = 7; // local variable to new scope
    printf(“local x in inner scope of main is %d\n”,x);
  } // end new scope
  printf(“local x in outer scope of main is %d\n”,x);

  void useLocal(void);
  void useStaticLocal(void);
  void useGlobal(void);
  void useLocal(void);
  void useStaticLocal(void);
  void useGlobal(void);

  printf(“local x in main is %d\n”,x);
}
```

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

Scope Rules: Example (cont.)

```
void useLocal(void)
{ int x = 25; // initialized each time useLocal is called
  printf("\nlocal x in useLocal is %d after entering useLocal\n",x);
  ++x;
  printf("\nlocal x in useLocal is %d before exiting useLocal\n",x);
}

void useStaticLocal(void)
{ static int x = 50; // initial once only
  printf("\nlocal static x in useStaticLocal is %d on entering useStaticLocal\n",x);
  ++x;
  printf("\nlocal static x in useStaticLocal is %d on exiting useStaticLocal\n",x);
}

void useGlobal(void)
{ printf("\nglobal x is %d on entering useGlobal\n",x);
  x *= 10;
  printf("\nglobal x is %d on exiting useGlobal\n",x);
}
```

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

Good Programming Style 4.1

Familiar yourself with the collection of functions in C standard functions (so that you don’t have to write it yourself).

Good Programming Style 4.2

Use meaningful function name and function parameters (makes program more readable).

Good Programming Style 4.3

Try to minimize scope.
Only use global variables if you really, really have to !!!
Excessive use of global variables may lead to confusion and debugging difficulties.

Q&A?