

CS102/IT102

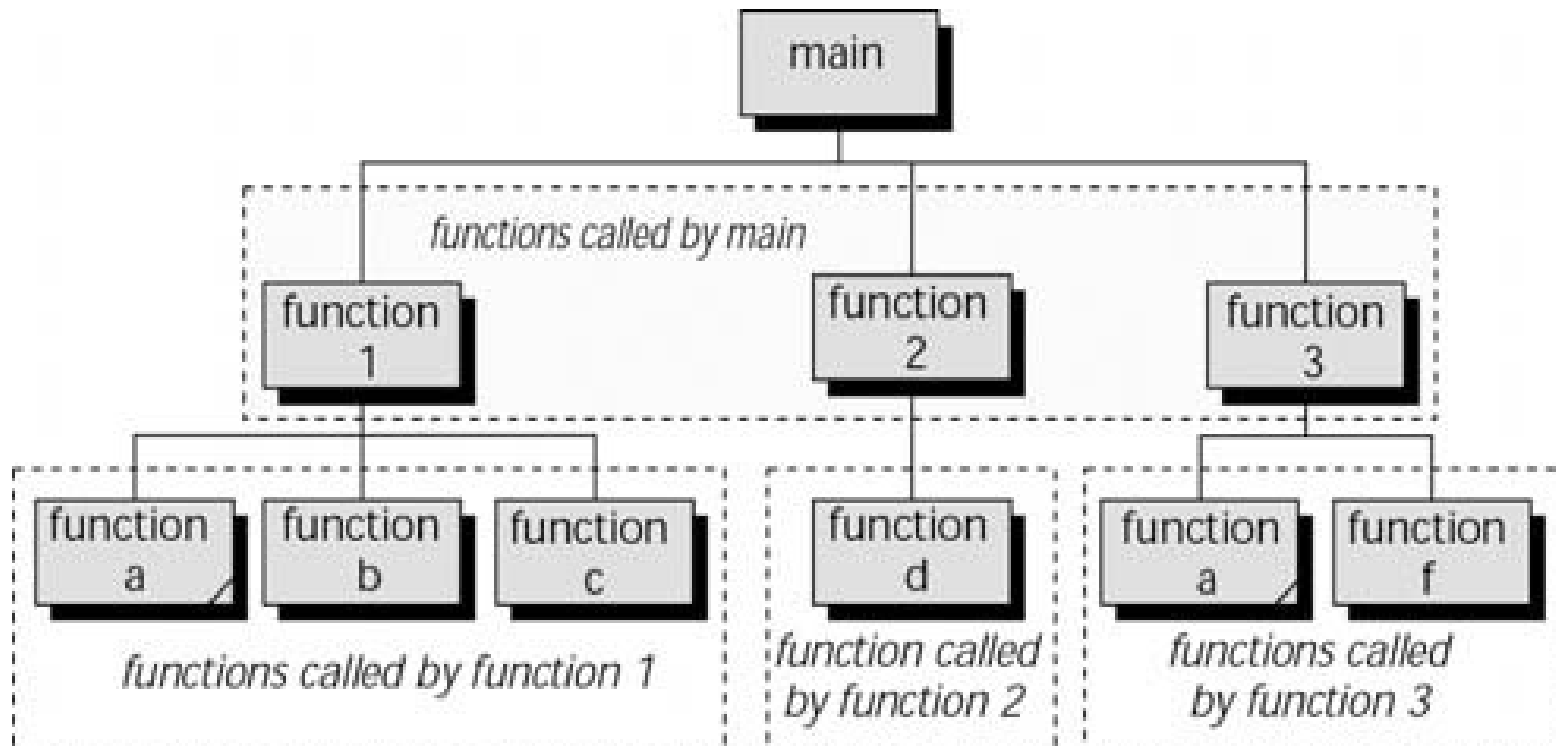
Computer Programming I

Lecture 12: Functions

Bicol University College of Science
CSIT Department
1st Semester, 2023-2024

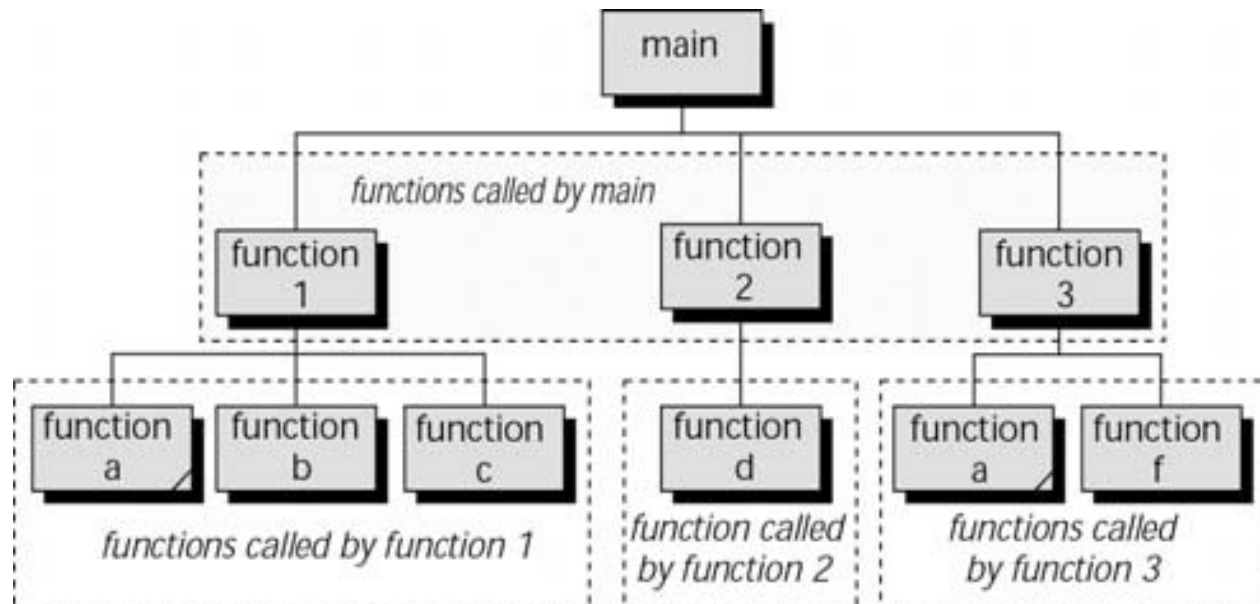
Functions in C

- In C, the idea of top-down design is done using **functions**.
- A C program is made of one or more functions, one and only one of which must be named **main**.
- The execution of the program always starts with **main**, but it can call other functions to do some part of the job.



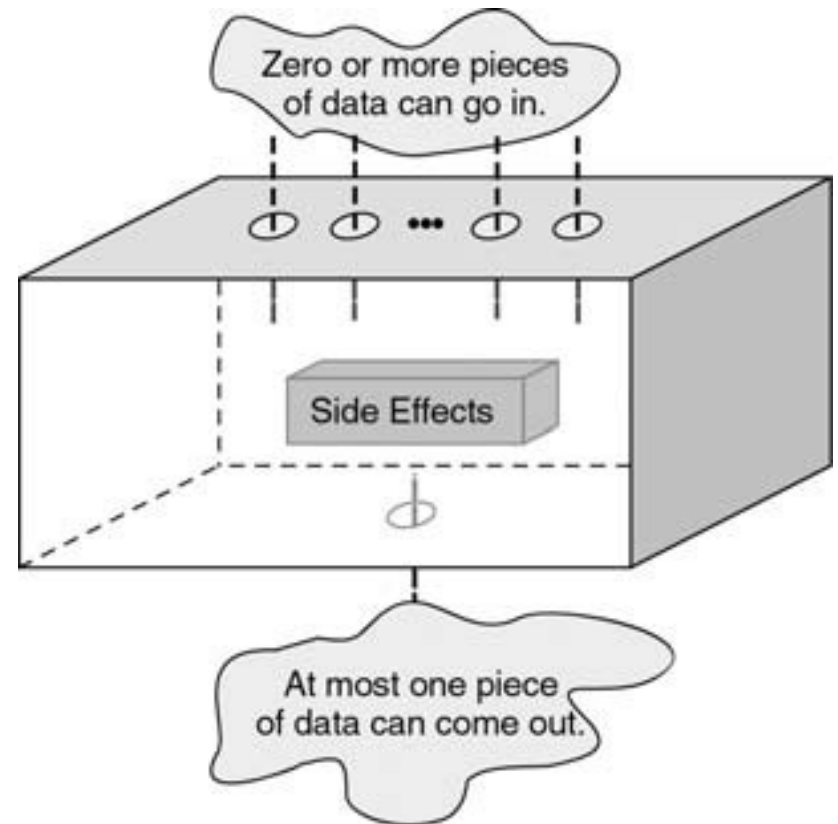
Functions in C

- A function in C (including **main**) is an independent module that will be called to do a specific task.
- A **called function** receives control from a **calling function**.
 - When the called function completes its task, it returns to the calling function
 - It may or may not return a value to the caller
- The **main** function is called by the operating system.
- When main is complete, control returns to the operating system.



Functions in C

- In general, the purpose of a function is to receive zero or more pieces of data, operate on them, and return at most one piece of data.
- A function in C can have a value, a side effect, or both.
 - The **side effect** occurs before the value is returned.
 - The **function's value** is the value of the expression in the return statement.
 - A function can be called for its value, its side effect, or both.

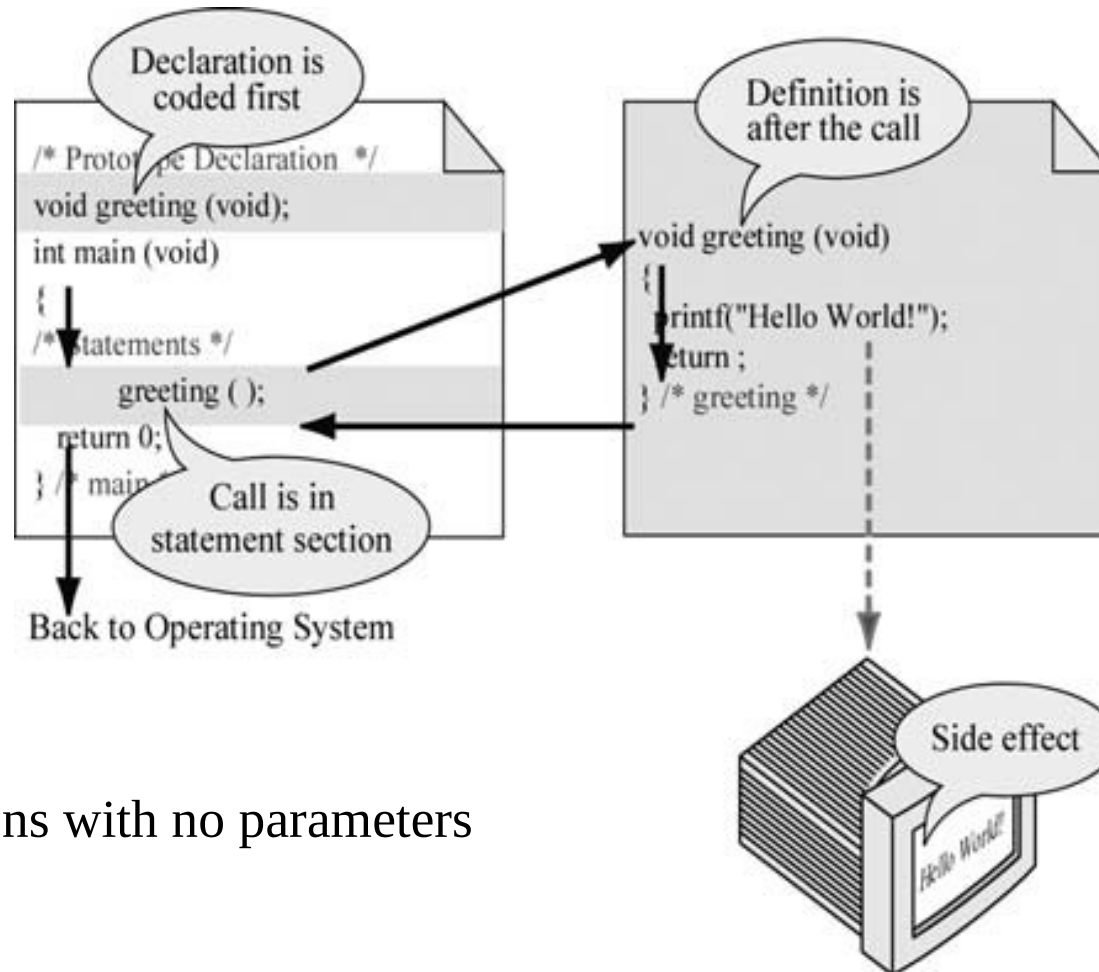


User-Defined Functions

- Functions must be both declared and defined:
 - **function declaration**, done first with a prototype statement;
 - **function definition**, coded after the function that makes the call, contains the code needed to complete the task.
- The name of a function is used in three ways:
 - for **declaration**, in a **call**, and for **definition**.
- Three general functions design
 - **void** functions with no parameters
 - **void** functions with parameters
 - Functions that return values

User-Defined Functions

- Declaring, calling and defining functions



void functions with no parameters

User-Defined Functions

- Create your own functions, similar to `printf()` or `sqrt()`
- Recall a *procedure* in an algorithm - a named collection of instructions
 - InviteToParty
 - RingUp
 - MakeToParty
- A function implements the procedure or function parts of an algorithm.

Writing User-defined Functions

- Need to specify:
 - the **name** of the function
 - its **parameters**
 - what it **returns**
 - **block** of statements to be carried out when the function is called
- The block of statements is called the “**function body**”

Function Definition

- The function definition contains the code for a function.

function
header

```
return_type function_name (formal parameter list)
```

```
{  
/* Local Definitions */  
...  
/* Statements */  
...  
} /* function_name */
```

function
body

Example: hello1.c

Prints a simple greeting.

```
procedure sayHello  
{  
  output "Hello World!"  
}
```

```
Main Program  
{  
  do procedure sayHello  
  
}
```

Example: hello1.c

Prints a simple greeting.

```
procedure sayHello
{
    output "Hello World!"
}
```

Main Program

```
{
    do procedure sayHello
}
```

```
#include <stdio.h>
void sayHello ( void );
```

```
/*
 * Call a function which
 * prints a simple greeting.
 */
```

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

```
/*
 * Print a simple greeting.
 */
```

```
void sayHello ( void )
{
    printf("Hello World!\n");
}
```

Example: hello1.c

Function declaration

```
#include <stdio.h>
void sayHello ( void );
```

Function call

```
/*
 * Call a function which
 * prints a simple greeting.
 */
```

```
int main(void)
```

```
{
    sayHello();
    return 0;
}
```

```
/*
 * Print a simple greeting.
 */
```

Function definition

```
void sayHello ( void )
{
    printf("Hello World!\n");
}
```

Example: hello1.c

Prints a simple greeting.

```
procedure sayHello
{
    output "Hello World!"
}
```

Main Program

```
{
    do procedure sayHello
}
```

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

```
/*
 * Print a simple greeting.
 */
```

```
void sayHello ( void )
{
    printf("Hello World!\n");
}
```

```
/*
 * Call a function which
 * prints a simple greeting.
 */
```

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

Example: hello1.c

***Function
definition***

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

```
/*  
 * Print a simple greeting.  
 */
```

```
void sayHello ( void )  
{  
    printf("Hello World!\n");  
}
```

```
/*  
 * Call a function which  
 * prints a simple greeting.  
 */
```

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

Function call

Example: hello1.c

Function name



Function body



```
#include <stdio.h>

/*
 * Print a simple greeting.
 */
void sayHello ( void )
{
    printf("Hello World!\n");
}

/*
 * Call a function which
 * prints a simple greeting.
 */

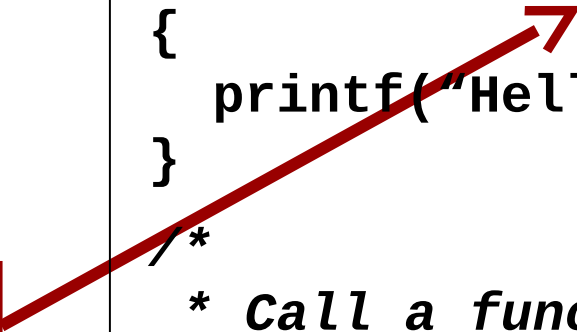
int main(void)
{
    sayHello();
    return 0;
}
```

Example: hello1.c

Return type



***Formal
Parameter List***



```
#include <stdio.h>

/*
 * Print a simple greeting.
 */
void sayHello ( void )
{
    printf("Hello World!\n");
}

/*
 * Call a function which
 * prints a simple greeting.
 */

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


Parameters

- Information passed to a function
- “Formal” parameters are local variables declared in the function declaration.
- “Actual” parameters are values passed to the function when it is called.

Example: badsort.c

```
/* Print two numbers in order. */
```

```
void badSort ( int a, int b )
```

```
{
```

```
    int temp;
```

```
    if ( a > b )
```

```
    {
```

```
        printf("%d %d\n", b, a);
```

```
    }
```

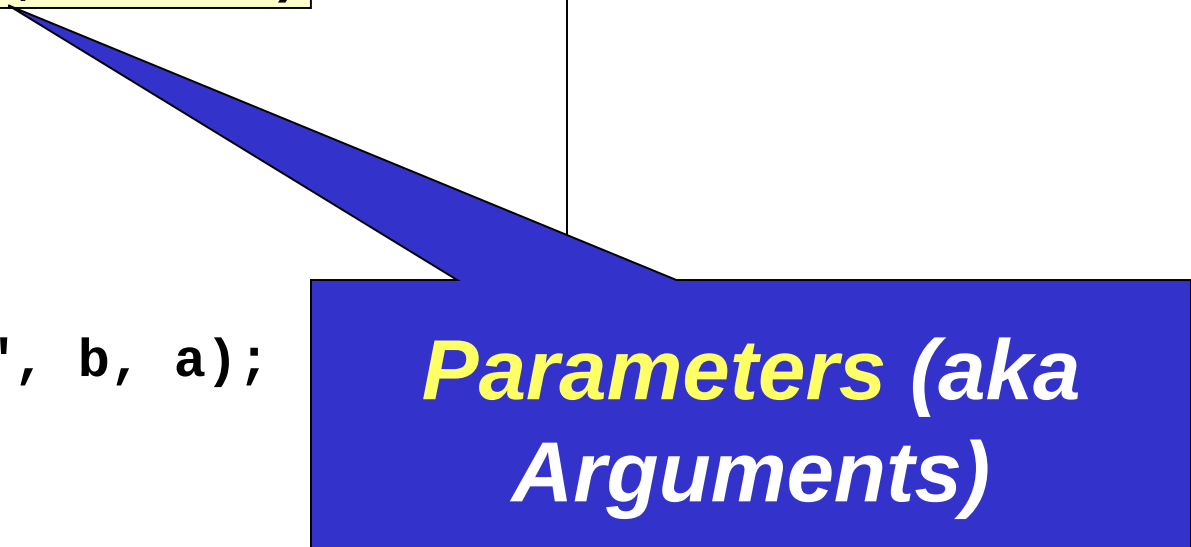
```
    else
```

```
    {
```

```
        printf("%d %d\n", a, b);
```

```
    }
```

```
}
```



Parameters (aka Arguments)

Example: badsort.c

```
/* Print two numbers in order. */
```

```
void badSort ( int a, int b )  
{  
    int temp;  
  
    if ( a > b )  
    {  
        printf("%d %d\n", b, a);  
    }  
    else  
    {  
        printf("%d %d\n", a, b);  
    }  
}
```

Example: badsort.c

**Formal
parameters**

```
/* Print two numbers in order.  
*/
```

```
void badSort ( int a, int b )  
{  
    int temp;  
  
    if ( a > b )  
    {  
        printf("%d %d\n", b, a);  
    }  
    else  
    {  
        printf("%d %d\n", a, b);  
    }  
}
```

**Actual
parameters**

```
int main(void)  
{  
    int x = 3, y = 5;  
  
    badSort ( 10, 9 );  
    badSort ( y, x+4 );  
    return 0;  
}
```

Parameters (cont.)

- Parameters are passed by **copying** the value of the actual parameters to the formal parameters.
- Changes to formal parameters do not affect the value of the actual parameters.

Example: badswap.c

```
/* Swap the values of two  
variables. */
```

```
void badSwap ( int a, int b )  
{  
    int temp;  
  
    temp = a;  
    a = b;  
    b = temp;  
  
    printf("%d %d\n", a, b);  
}
```

```
int main(void)  
{  
    int a = 3, b = 5;  
  
    printf("%d %d\n", a, b);  
    badSwap ( a, b );  
    printf("%d %d\n", a, b);  
  
    return 0;  
}
```

Example: badswap.c

```
/* Swap the values of two  
   variables. */  
void badSwap ( int a, int b )  
{  
    int temp;  
  
    temp = a;  
    a = b;  
    b = temp;  
  
    printf("%d %d\n", a, b);  
}
```

```
int main(void)  
{  
    int a = 3, b = 5;  
  
    printf("%d %d\n", a, b);  
    badSwap ( a, b );  
    printf("%d %d\n", a, b);  
  
    return 0;  
}
```

Output:

3 5

Example: badswap.c

```
/* Swap the values of two
   variables. */
void badSwap ( int a, int b )
{
    int temp;

    temp = a;
    a = b;
    b = temp;

    printf("%d %d\n", a, b);
}
```

```
int main(void)
{
    int a = 3, b = 5;

    printf("%d %d\n", a, b);
    badSwap ( a, b );
    printf("%d %d\n", a, b);

    return 0;
}
```

Output:

```
3  5
5  3
```


Example: badswap.c

```
/* Swap the values of two
   variables. */
void badSwap ( int a, int b )
{
    int temp;

    temp = a;
    a = b;
    b = temp;

    printf("%d %d\n", a, b);
}
```

```
int main(void)
{
    int a = 3, b = 5;

    printf("%d %d\n", a, b);
    badSwap ( a, b );
    printf("%d %d\n", a, b);

    return 0;
}
```

Output:

3 5

5 3

3 5

Example: badswap.c

```
/* Swap the values of two
   variables. */
void badSwap ( int a, int b )
{
    int temp;

    temp = a;
    a = b;
    b = temp;

    printf("%d %d\n", a, b);
}
```

```
int main(void)
{
    int a = 3, b = 5;

    printf("%d %d\n", a, b);
    badSwap ( a, b );
    printf("%d %d\n", a, b);

    return 0;
}
```

Called function's
environment:

a: 5

b: 3

Calling function's
environment:

a: 3

b: 5

Parameters (cont.)

- If a function does not take parameters, declare its formal argument list **void**.

Declaration:

```
void sayHello ( void )  
{  
    printf("Hello World!\n");  
}
```

Function call:

```
sayHello();
```

Return Values

- Values are returned by copying a value specified after the **return** keyword

Example: max.c

Return type

```
/* Returns the larger of two  
   numbers. */  
int max (int a, int b)  
{  
    int result;  
  
    if (a > b)  
    {  
        result = a;  
    }  
    else  
    {  
        result = b;  
    }  
  
    return result;  
}
```

Example: max.c

```
/* Returns the larger of two  
   numbers. */  
int max (int a, int b)  
{  
    int result;  
  
    if (a > b)  
    {  
        result = a;  
    }  
    else  
    {  
        result = b;  
    }  
  
    return result;  
}
```

For example:

The value of the
expression

max(7, 5)

is the integer 7.

Example: max.c

This style okay.

```
/* Returns the larger of two  
   numbers. */  
int  
max (int a, int b)  
{  
    int result;  
  
    if (a > b)  
    {  
        result = a;  
    }  
    else  
    {  
        result = b;  
    }  
  
    return result;  
}
```

Return Values (cont.)

- If a function does not return a value, declare its return type **void**.

Declaration:

```
void sayHello ( void )  
{  
    printf("Hello World!\n");  
}
```

Function call:

```
sayHello();
```


Example: hello1.c

```
#include <stdio.h>

void info(void);

void main(void)
{   double gross, tax;
    int dependents;

    printf("Dependents
(including employee)? ");
    scanf("%d", &dependents);
    info( );

    printf("Write the check!\n");
}
```

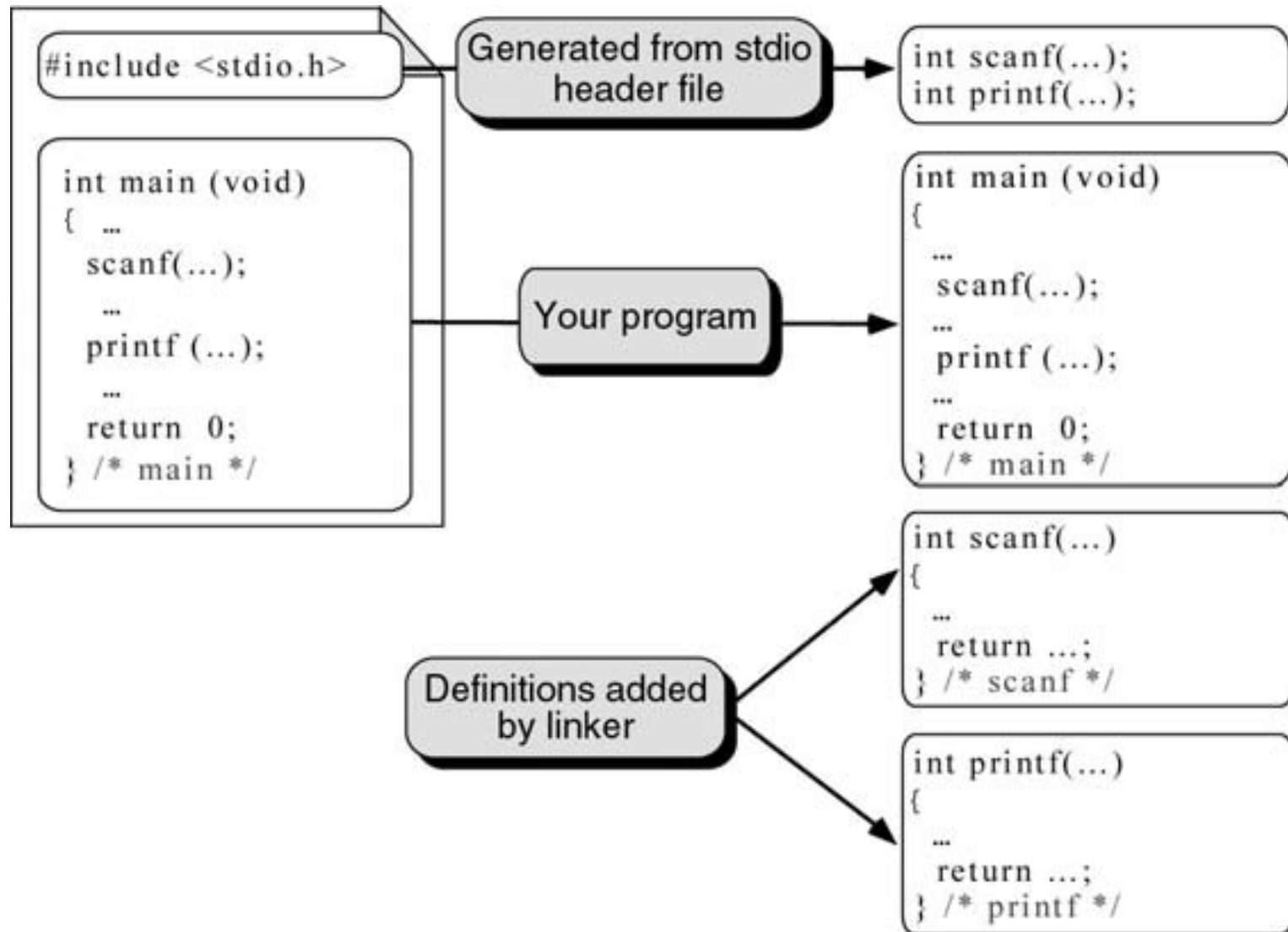
```
void info(void)
{   int emp_id, emp_num, dept;

    printf("Employee ID? ");
    scanf("%i", &emp_id);
    emp_num = emp_id / 10;
    printf("Employee #%i in the ", emp_num);
    dept = emp_id % 10;
    switch (dept)
    {   case 1:
            printf("Information Systems");
            break;
        case 2:
            printf("Accounting");
            break;
        default:
            printf("Manufacturing or other");
    }
    printf(" department.\n");
}
```

Standard Library Functions

- C/C++ provides a rich collection of standard functions whose definitions have been written and are ready to be used in our programs.
- To use these functions, you must include their prototype declarations. The prototypes for these functions are grouped together and collected in several header files. Instead of adding the individual prototype of each function, therefore, we simply include the headers at the top of our programs.)

Standard Library Functions



Standard Library Functions

Standard functions for mathematical manipulation

- **abs**, **fabs**, and **labs** return the absolute value of a number. The **abs** and **labs** functions are found in **stdlib.h**. The **fabs** function is found in **math.h**.

Prototypes:

- `int abs (int number);`
- `long labs (long number);`
- `double fabs (double number);`

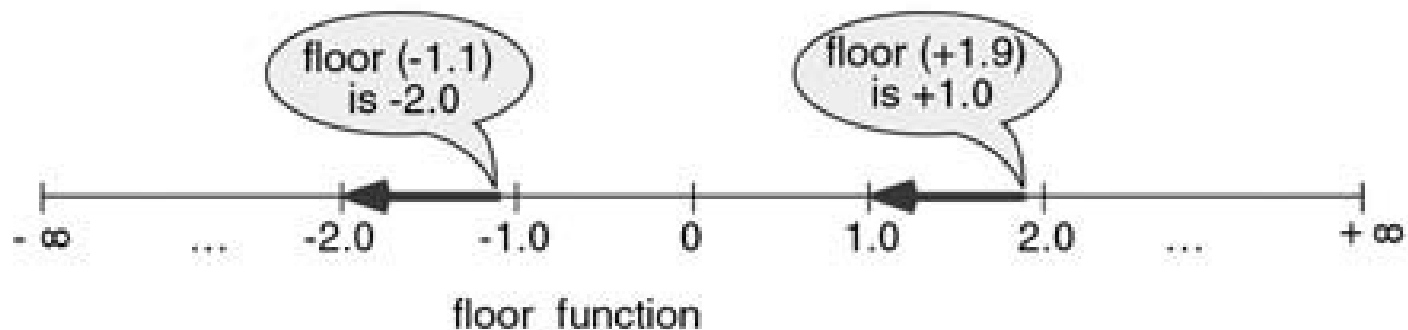
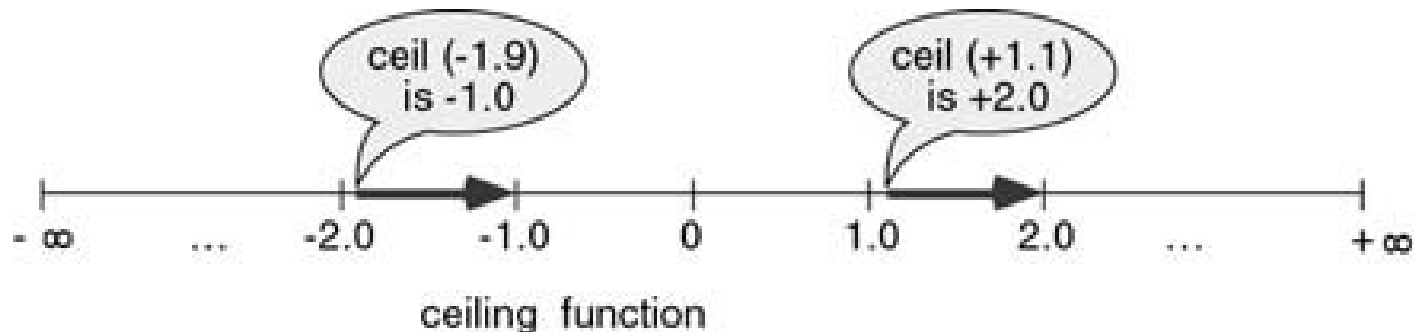
Examples:

- `abs (3) /* returns 3 */`
- `fabs (-3.5) /* returns 3.5 */`

Standard Library Functions

Standard functions for mathematical manipulation

- A ceiling, or **ceil**, is the smallest integral value greater than or equal to a number. For example, the ceiling of 2.00001 is 3.
- A **floor** is the largest integral value that is equal or less than a number. For example, the floor of 2.99999 is 2.



Standard Library Functions

Standard functions for mathematical manipulation

- The **pow** function returns the value of the **x** raised to the power of **y** – that is, x^y . An error occurs if the base (x) is negative and the exponent (y) is not an integer, or is the base is zero and the exponent is not positive.

Prototype: `double pow (double x, double y);`

Examples:

- `pow (3.0, 4.0) /* returns 81.0 */`
- `pow (3.4, 2.3) /* returns 16.687893 */`
- The **sqrt** function returns the non-negative square root of number. An error occurs if number is negative.

Prototype: `double sqrt (double number);`

Example: `sqrt (81.0) /* returns 9.0 */`

Scope

- **Scope** determines the region of the program in which a defined object is visible – that is, the part of the program in which you can use the object's name.
- Scope pertains to any object that can be defined, such as a variable or a function prototype statement.
- It does not pertain directly to precompiler directives, such as define statements; they have separate rules.
- A **block** is zero or more statements enclosed in a set of braces. Recall that a function body is enclosed in a set of braces; thus, a body is also a block. A block has a declarations section and a statement section.
- A **global area** of a program consists of all statements that are outside functions.

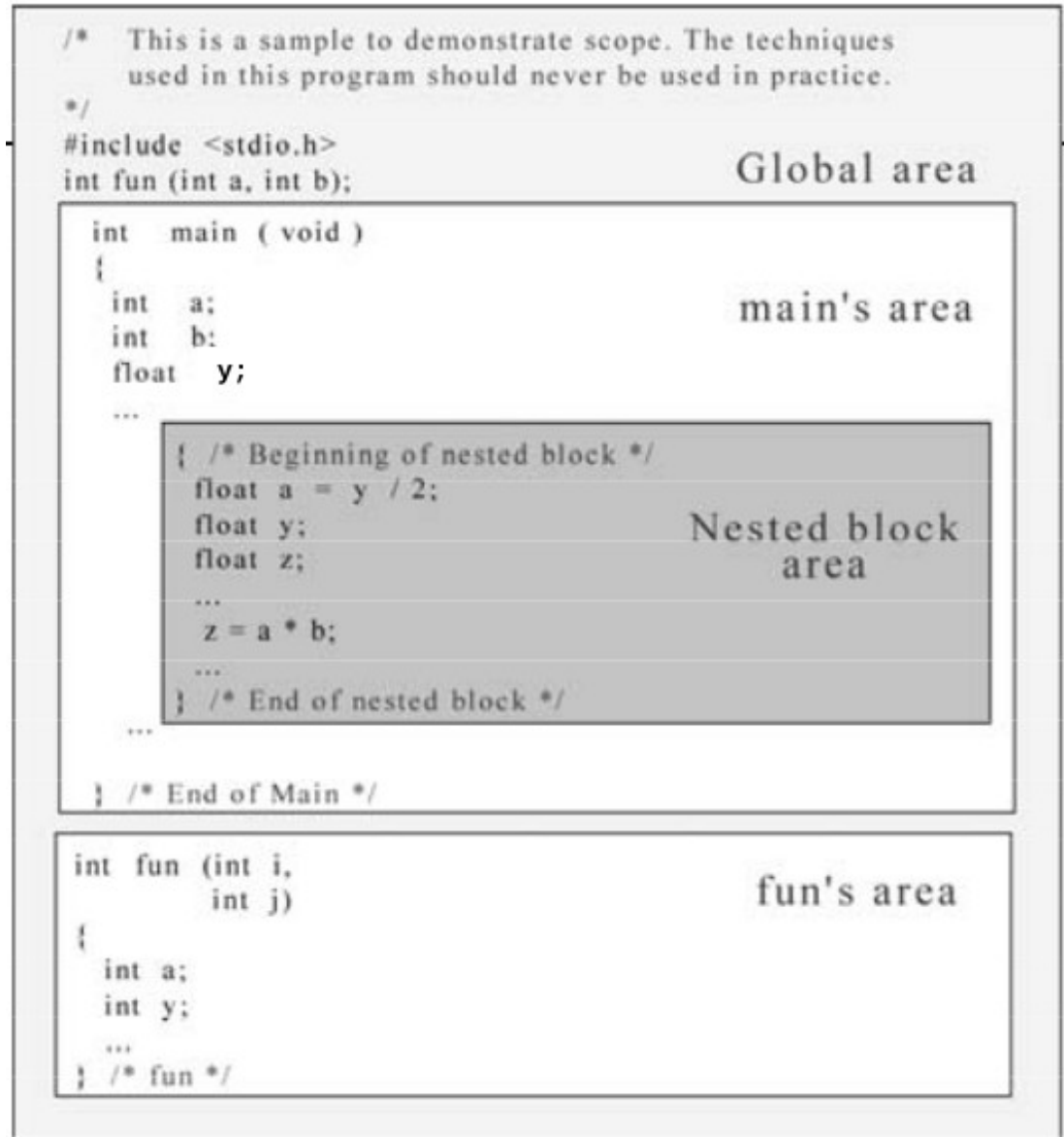
Scope

- **Global scope**

- Any object defined in the global area of a program is visible from its definition until the end of the program.

- **Local scope**

- Variables are in scope from their point of definition until the end of their function or block.
- It is poor programming style to reuse identifiers within the same scope.



Common Programming Errors

- Several possible errors are related to passing parameters.
 - It is a compile error if the types in the prototype declaration and function definition are incompatible.
 - It is a compile error to have a different number of actual parameters in the function call than there are in the prototype statement.
 - It is a logic error if you code the parameters in the wrong order. Their meaning will be inconsistent in the called program.
- Using a void return with a function that expects a return value or using a return value with a function that expects a void return is a compile error.
- Each parameter's type must be individually specified; you cannot use multiple definitions like you can in variables. For example, **the following is a compile error** because y does not have a type:
 - `double fun (float x, y);`

Common Programming Errors

- It is a compile error to define local variables with the same identifiers as formal parameters, as shown below.

```
double divide( float dividend, float divisor)
{
    /* Local Definitions */
    float dividend; /* ERROR!!! */
    ...
} /* divide */
```

- Forgetting the semicolon at the end of a function prototype statement is a compile error. Similarly, using a semicolon at the end of the header in a function definition is a compile error.

Common Programming Errors

- It is most likely a logic error to call a function from within itself or one of its called functions.
- It is a compile error to attempt to define a function within the body of another function.
- It is a run-time error to code a function call without the parentheses, even when the function has no parameters.

```
printHello; /* Not a call */
```

```
printHello ( ); /* A valid a call */
```

- It is a compile error if the type of data in the return statement does not match the function return type.

Summary

- Each C program must have one and only one function called *main*.
- A function can return only one value.
- A function can be called for its returned value or for its side effect.
- The function call includes the function name and the values of the actual parameters to provide the called function with the data it needs to perform its job.
- Each actual parameter of the function is an expression. The expression must have a value that can be evaluated at the time the function is called.
- A local variable is known only in a function definition. The local variables do not take part in communication between the calling and the called functions.
- If a function returns no value, the return type must be declared as void.
- If a function has no parameters, the parameter list must be declared void.
- The actual parameters passed to a function must match in number, type, and order with the formal parameters in the function definition.

Summary

- The general format for a function definition is

```
return_type function_name (parameter list) {  
    /* Local Definitions */  
  
    /* Statements */  
} /* function_name */
```

- When a function is called, control is passed to the called function. The calling function "rests" until the called function finishes its job.
- We highly recommend that every function have a return statement. A return statement is required if the return type is anything other than void.
- Control returns to the caller when the return statement is encountered.
- A function prototype requires only the return type of the function, the function name, and the number, types, and order of the formal parameters. Parameter identifiers may be added for documentation but are not required.
- The scope of a parameter is the block following the header.
- A local variable is a variable declared inside a block. The scope of a local variable is the block in which it is declared.