

Lecture 4

Functions

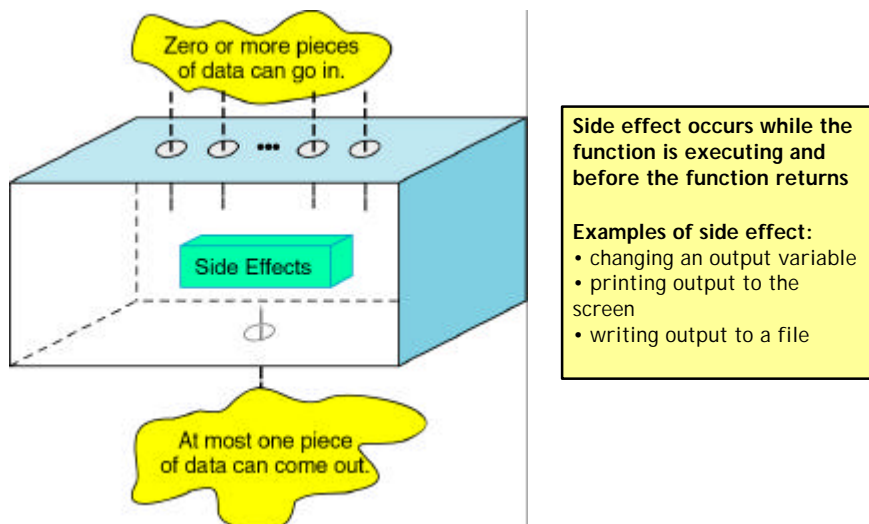
Today's Topics

- Function
- Scope
- Parameter Passing
- Modular Programming

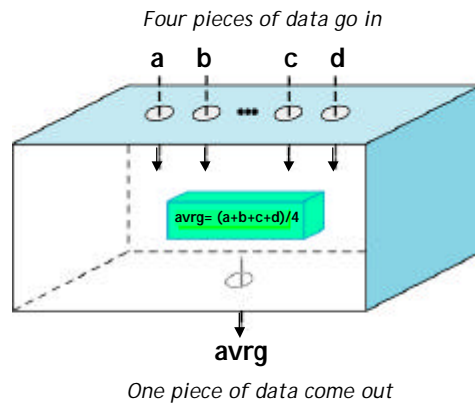
Functions

- Function is a "black box", a **mini program** with
 - a set of **inputs**
 - an **output**
 - a **body** of statements
- It performs **one well-defined task**
 - (e.g. printf only do the printing process)

Figure 4-3: Function concepts



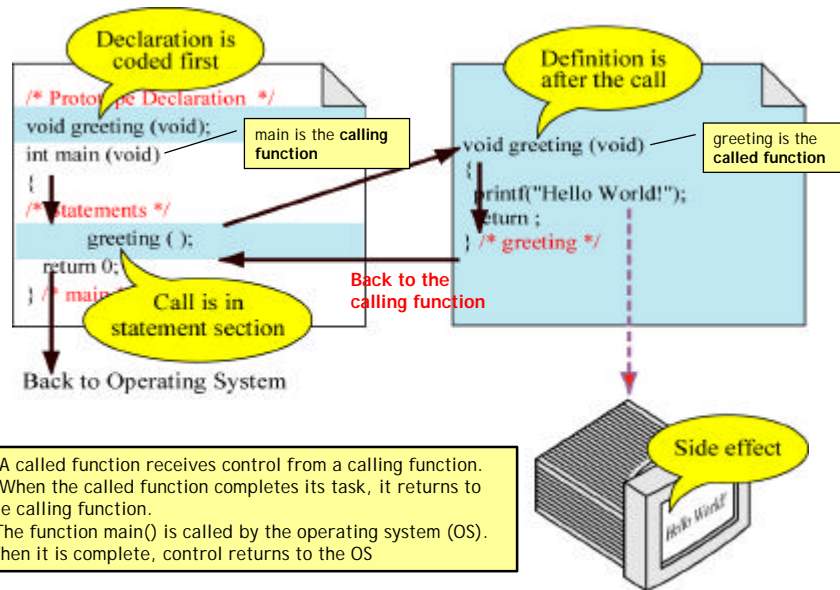
Example: Calculate the average of four numbers (a,b,c and d)



Three steps in using functions

- 1. Declare** the function:
 - Known as function declaration or function prototyping.
 - Write a **function prototype** that specifies:
 - the **name** of the function
 - the **type** of its return value
 - its list of **arguments** and their types
- 2. Define** the function:
 - Known as function definition or function implementation.
 - Write the block of statements (**body**) of the function to define processes should be done by the function.
- 3. Call** the function:
 - Known as function call or function invocation.
 - Call the name of the function in order to execute it.

Figure 4-4: Declaring, calling and defining functions



Declaring a function

```
return_type function_name ( formal_parameter_list );
```

- The syntax of a function declaration (formally called *prototype*) contains:
 - The *type of the return value* of the function
 - if the function does not return anything, the type is **void**
 - if *return_type* is not written the Compiler will assume it as **int**
 - The *name* of the function
 - same rules as for variable naming
 - A list of *formal parameter* made up of its names and its types. They are enclosed in parentheses
 - The prototype must be terminated by a *semicolon*
- Function prototypes are usually written between the preprocessor directives and `main()`.

Calling a function

- The name of a function is called in order to execute the function.
- A **called function** receives control from a **calling function**.
- When the called function completes its task, it returns to the calling function.
- The called function may or may not returns a value to the calling function

Functions that return a value can be used in an expression or as a statement.

Example:

if given function definition as below:

```
float avrg(int a, int b, int c)
{
    return (a+b+c)/3.0;
}
```

All function calls below are valid

```
result = avrg(1,2,3) + avrg(4,5,6); // function calls are
// used in an expression
avrg(1,2,3); // function call is used as a statement
printf("The average is %.2f", avrg(1,2,3) );
```

void function cannot be used in an expression because it does not return any value. It can only be used as a statement.

Example:
if given function definition as below:

```
void greeting(void)
{
    printf("Hello");
    return;
}
```

Function call below would be an error

```
result = greeting(); // Error! greeting() is a void function
```

- **Formal parameters** are variables that are declared in the header of the function definition
- **Actual parameters** are the expressions in the calling statement
- When making a function call, the formal and actual parameters must match exactly in **type**, **order** and **number**.
- The parentheses is compulsory, even when no parameters present. This is the way, how the compiler knows an identifier either it is a function or a variable.
 - Example:

```
greeting; // Error. greeting is a function. So, it must have the ()
// eventhough no parameter present
```

Figure 4-5: void function with parameters

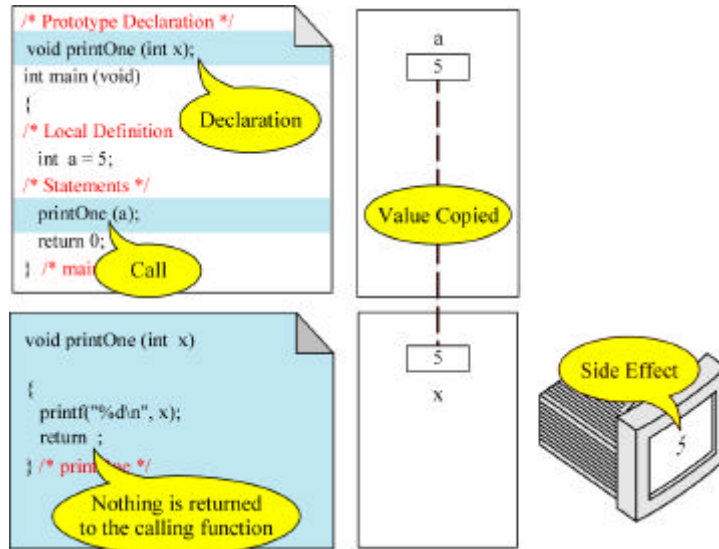
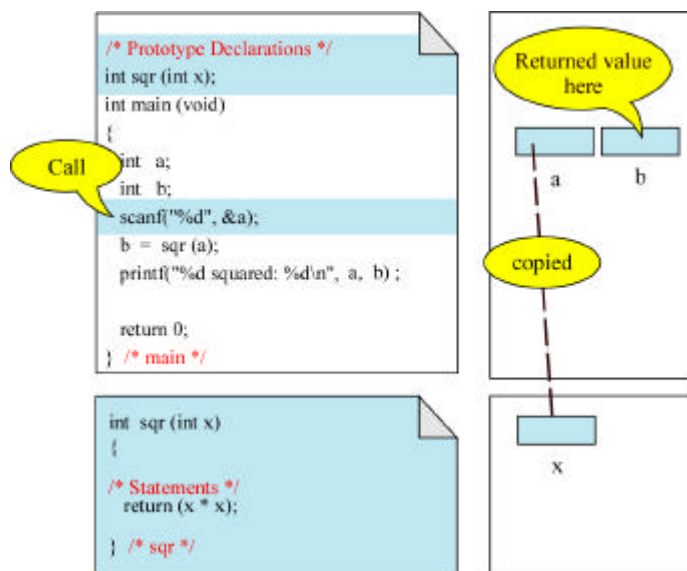


Figure 4-6: Function that returns a value



Function that **calls itself** is known as **recursive function**

Example:

```
int factorial(int n)
{
    if (n>1) return n * factorial(n-1);
    return 1;
}
```

This function calculates the factorial of n,
 $n! = n \times (n-1) \times (n-2) \times \dots \times 2 \times 1$

At the first statement of the function definition, the function calls itself.

return statement

- A function returns a computed value back to the calling function via a `return` statement.
- A function with a non-void return type must always have a `return` statement.
- Code after a `return` statement is never executed.

The following function always returns 10.

```
int square (int n)
{
    return 10;
    n = n * n;
    return n;
}
```

This line causes the control back to the calling function and ignores the rest of lines.

These two lines are ignored and never executed

Local & global variables

- Local variable is a variable declared inside a function.
 - This variable can only be used in the function.
- Global variable is a variable declared outside of any functions.
 - This variable can be used anywhere in the program.

Example: Local vs. Global

```
#include<stdio.h>

void print_number(void) {
    int p;
}

void main (void)
{
    int q = 5;
    printf("q=%d", q);
    p=10;
    print_number();
}

void print_number(void)
{
    printf("%d", p);
    q = q + 5;
}
```

p is declared outside of all functions. So, it is a **global variable**.

q is declared inside the function main. So, it is a **local variable** to the function.

p can be used anywhere

Error! **q** can only be used in the function main, because it is a local variable.

Example: Local vs. Global

```
#include<stdio.h>

int p=10;

void main (void)
{
    int p=5;

    printf("p=%d", p);
}
```

Global and local variables use the same name, **p**. They are valid statements.

Which **p** will be used? The global or local? When this happen, the **local variable is more dominant**. So, the value of local variable p=5, will be used.

Output: p=5

Example: Local vs. Global

```
#include<stdio.h>

double compute_average (int num1, int num2);

void main (void)
{
    double average;
    int age1 = 18, age2 = 23;
    average = compute_average(age1, age2);
    return 0;
}

double average (int num1, int num2)
{
    double average;
    average = (num1 + num2) / 2.0;
    return average;
}
```

Same variable names?!?
--it's OK; they're local to
their functions. Compiler treat
them as different variables.

Scope

- **Scope** determines the **area** of the program in which an identifier is visible (*ie. the identifier can only be used in that area*)
- Remember, identifier can be a variable, constant, function, etc.
- Examples:
 - Scope of a local variable : only in the function body where it was declared.
 - Scope of a global variable : everywhere in the program.

Scope

- Scope that enclosed in { } is called a **block**.
- Inner block can use identifiers that were declared outside of it.
 - eg. Any function can use any global variables.
- But outer block cannot use identifiers that were declared in inner block.
- Any block cannot use any identifier that was declared in other block.
 - eg. You cannot use any local variable from a function in another function.

Figure 4-23: Scope for global and block areas

```
/* This is a sample to demonstrate scope. The techniques
   used in this program should never be used in practice.
*/
#include <stdio.h>
int fun (int a, int b);

int main ( void )
{
    int a;
    int b;
    float y;
    ...
    { /* Beginning of nested block */
        float a = y / 2;
        float y;
        float z;
        ...
        x = a * b;
        ...
    } /* End of nested block */
    ...
} /* End of Main */

int fun (int i,
        int j)
{
    int a;
    int y;
    ...
} /* fun */
```

Example: Scope of inner and outer block and function

```

#include <stdio.h>

void fun(void);
int a=1 , b=2;

void main(void)
{ // outer block
  int b=3, c=4;

  b = b + 22; // b=3 + 22 =25
  fun();
  printf("a=%d", a); // output: a=1

  { // inner block
    int d=5;

    printf("c=%d", c); // output: c=4
    printf("d=%d", d); // output: d=5
    d = b + 10; // d=3 + 10 =13
  }

  printf("d=%d", d);
}

```

```

void fun(void)
{
  int a = b + 5; // a=2 + 5 = 7
  int b=1;

  print ("a=%d", a); // output: a=7
  print ("b=%d", b); // output: b=1

  print ("c=%d", c);
}

```

Error! Variable c was declared in the function main. It cannot be used outside of the function.

Error! Variable d was declared in the inner block. It cannot be used in the outer block.

Chapter 4: Functions | SCP1103 Teknik Pengaturcaraan C | Jumail, FSKM, UTM, 2005 | Last Updated: September 2005 | Slide 27

Parameter Passing

- To call a function, we write its name and give it some information which are called **parameters**.
- Giving information to a *function call* is called **parameter passing**.
- You have learnt these:
 - **formal parameters** – parameters that are used in *function definition*
 - **actual parameters** – parameters that are used in *function call*

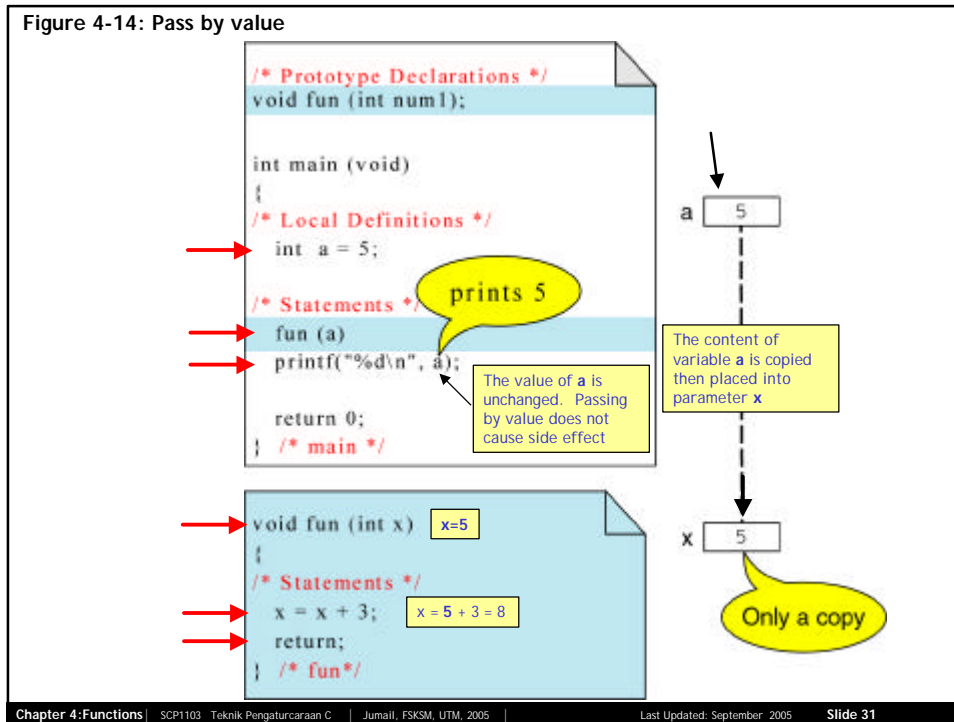
Parameter Passing

- In order to pass parameters, the actual and formal parameters must match exactly in **type**, **order** and **number**.
 - Eg. If you have defined a function with its formal parameter as an “**output parameter**”, you must use the ampersand (&) for its actual parameter when you call the function. Otherwise you will get a *syntax error* “*Type mismatch*”
- Two types of passing:
 - Pass by value
 - Pass by reference

Pass by Value

- When a data is passed by value, a copy of the data is created and placed in a local variable in the called function.
- Pass by value does not cause side effect.
 - After the function call completed, the original data remain unchanged.

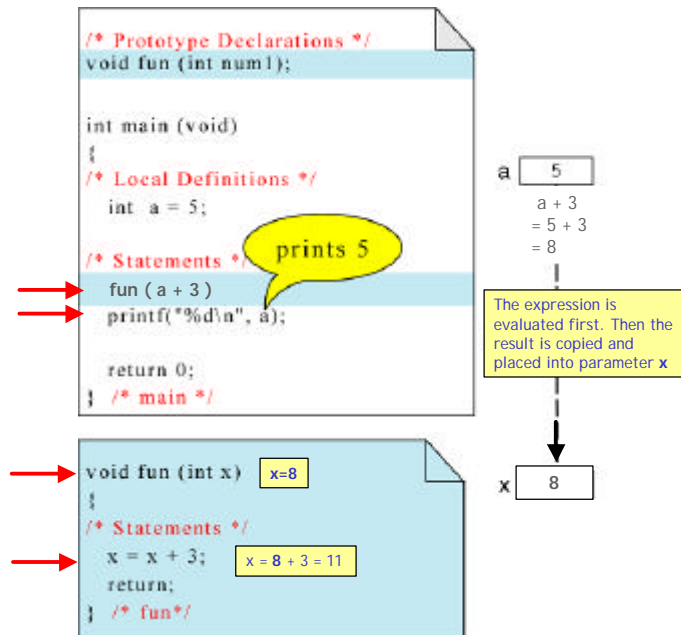
Figure 4-14: Pass by value



Pass by Value

- You have been introduced with the term “input parameter” in Tutorial 4. This type of parameter is passed using “Pass by Value”.
- When passing an expression, the expression is evaluated first, then the result is passed to the called function.

Passing expression by value



Pass by Reference

- Passing by reference is a passing technique that passes the address of a variable instead of its value.
 - That's why it is also called [Pass by Address](#)
- Pass by reference causes side effect to the actual parameters.
 - When the called function changes a value, it actually changes the original variable in the calling function.
- Only variables can be passed using this technique.
- You have been introduced with the term "[output parameter](#)" in Tutorial 4. This type of parameter is passed using "Pass by Reference".

Pass by Reference

- The formal parameter must be a **pointer**.

Example:

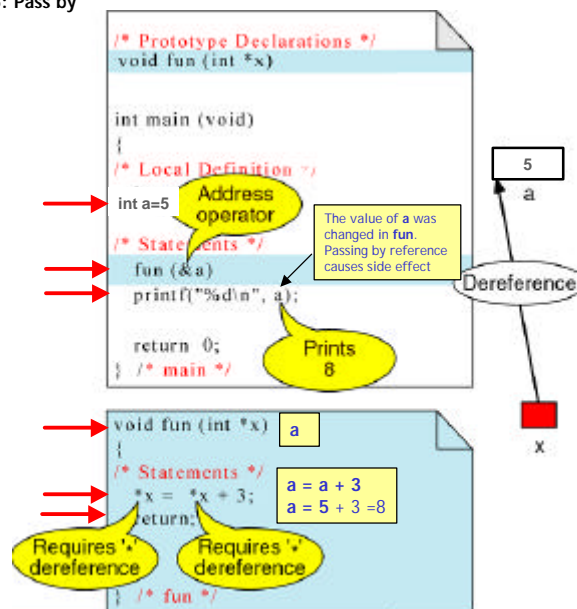
```
void fun(int *x) // x is a pointer variable
{
    // function body
}
```

- The actual parameter must be an address of a variable.

Example:

```
int n;
fun(&n); // &n means "address of variable n"
```

Figure 4-15: Pass by reference



Pointer

- How the passing by reference works? To understand this, let's look first at the concept of pointer.
- What we have used so far is normal variables or also called **data variables**. A data variable contains a **value** (eg. integer number, a real number or a character) .
- Pointer is a variable that contains the **address of another variable**. It also known as **address variable**.

Pointer

- Since pointer also a variable, it is declared like the data variable. The difference is, need to put the asterisk (*) .

Example:

```
int n=5; // n is a normal variable (or data variable)
int *p = &n; // p is pointer that holds the address of variable n
```

Pointer

- There are two special operators for pointers: **address operator** and **indirection operator**.
- Address operator, **&**
 - Get the **address** of a variable
 - Example: **&n**
meaning: "give me the address of variable n"
- Indirection operator, *****
 - Get the **value** of variable where its address is stored in the pointer.
 - Example: ***ptr**
meaning: "give me the value of variable where its address is stored in the variable ptr"
 - Pointer declaration also uses the asterisk (*) . Don't get confused. Pointer declaration and indirection operator are different.

Address and Indirection operators

```
void main(void)
{
  int a = 5;
  int *ptr = &a;
  printf("%d ", ptr);
  printf("%d ", &a);
  printf("%d ", *ptr);
  *ptr = *ptr + 5;
  printf("%d ", *ptr);
  printf("%d ", a);
}
```

Prints 2000

Prints 2000

Prints 2004

Prints 5.
This is how it works. ptr contains 2000.
Go to the address 2000 and get its content
=> 5. Means that, the value of *ptr is 5.

This means, $a = a + 5$, because ptr holds the address of a. The new value of a is 10

Prints 10

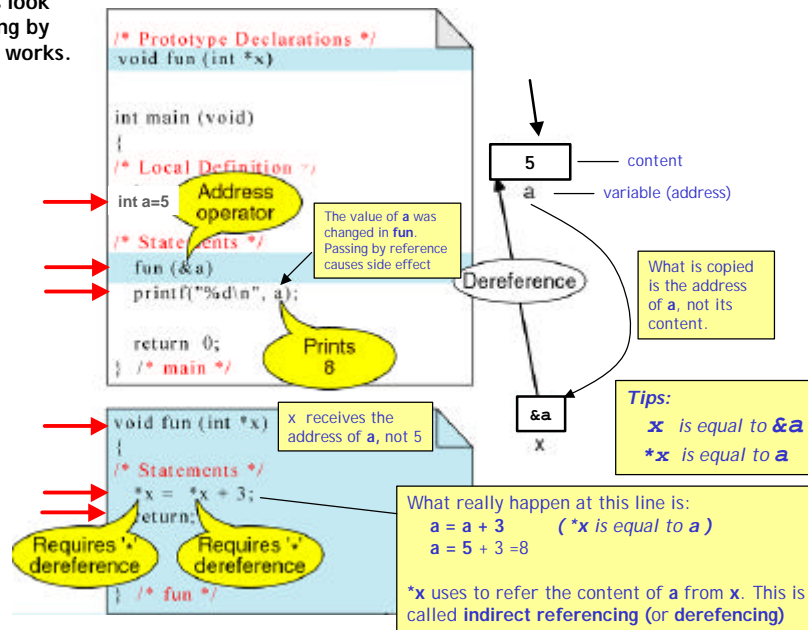
Prints 10

Memory

Address	Content
2000	5 a
2004	2000 ptr

Address of each variable is specified by the Compiler. The addresses that are shown above are not actual addresses. We use them only for examples.

Now, let's look how passing by reference works.



Modular Programming

Why do we use function?

- Big programs get complicated. Small programs are simpler.
- Key idea : Decomposition
 - Break big complicated sets of instructions into smaller, simpler modules

Other reason: **without** functions

```
#include<stdio.h>

void main(void)
{ /*Fahrenheit-to-Celsius */

    double degF,degC;
    printf("Enter degrees F: ");
    scanf("%f", &degF);

    /* F-to-C conversion: */
    ratio = 5.0 / 9.0; /* We will unify this */
    degC = (degF-32)*ratio; /* set of statements to */
    /* make a new function. */
    printf("%f degrees F are %f degrees C\n",degF,degC);
}
```

What if we need this conversion step in many different places in our program?

with functions

```
#include<stdio.h>

double F_to_C (double degreesF); ← Declare the function
                                ('Function prototype')

void main(void)
{ /*Fahrenheit-to-Celsius */

    double degF,degC, ratio;
    printf("Enter degrees F: ");
    scanf("%f", &degF);
    degC = F_to_C (degF); ← Call the function
                          when you need it.
    printf("%f degrees F are %f degrees C\n",degF,degC);
}

double F_to_C (double degreesF) ← Define the function.
{
    const double ratio = 5.0/9.0;
    return (degrees-32)*ratio;
}
```

Modular Programming

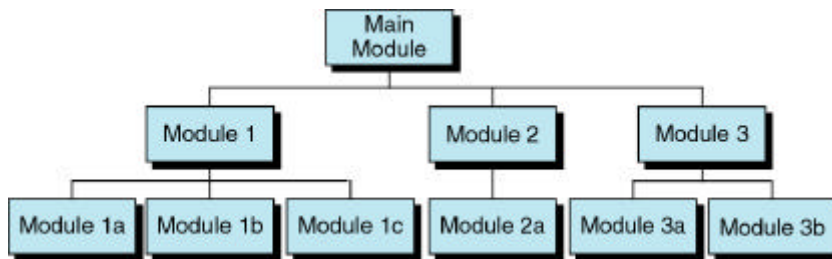


Figure 4-1: Structure Chart

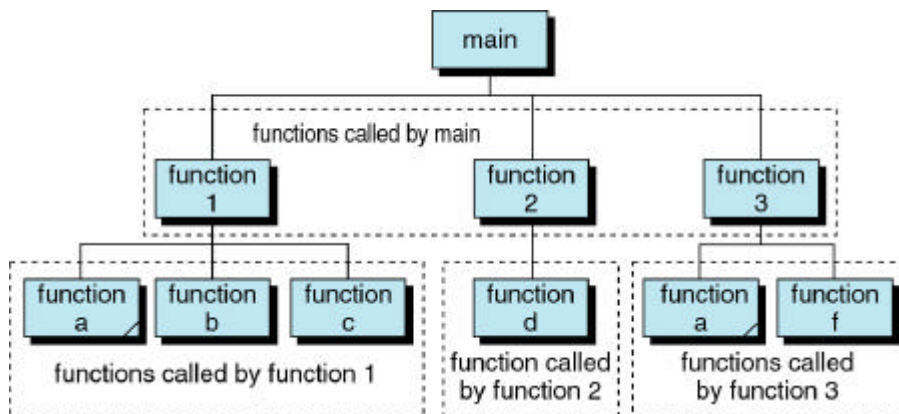


Figure 4-2: Structure chart in C

Summary

- Function is a sub-program that performs one-defined task.
- Three steps to do in using function:
 - declare the function prototype
 - define the function
 - call the function
- Function prototype must be terminated with semicolon, but not for function definition.

Summary

- Formal parameters are parameters that are declared in the header of function definition.
- Actual parameters are parameters that are passed in function call.
- The type, order and number of parameters of formal and actual parameters must match exactly.

Summary

- Function that returns a value can be used directly in an expression.
- void function can only be used as a statement.
- When calling a function, the parentheses is compulsory, even when no parameter present.
- Recursive function is a function that calls itself

Summary

- return statement does two things:
 - returns a value to the calling function
 - returns the control of execution to the calling function
- Local variable is declared inside a function. It can only be used in that function.
- Global variable is declared outside of any functions. It can be used everywhere in the program.
- Scope determines the area in which an identifier can be used.

Summary

- Parameter can be passed to a function in two ways:
 - pass by value
 - pass by reference
- When passing by value, a copy of the data is created and placed in a local variable in the called function.
- When passing by reference, the address of the variable from the calling function is sent to the called function.
- Variable that is sent using passing by reference may have side effect, but not for passing by value.
- Modular programming can be done by breaking one big program into multiple smaller functions.