

# Foundations of C Programming (Structured Programming)

- Functions(函数)

# Outline

- Structured programming
- Function
- Function declaration
- Library functions and user-defined functions
- Function call

# A Function

```
int main() {  
    int i = 1;  
    int number = 100;  
    int sum = 0;  
    while (i <= number) {  
        sum = sum + i;  
        i++;  
    }  
    printf("the sum of integers from 1 to  
        100 is %d", sum);  
    return 0;  
}
```

This is a definition of a function with **main** as its name.

# Function

How about the handling of a **complex** problem?

Can we write a function that include a **thousand or more lines**?

# Functions

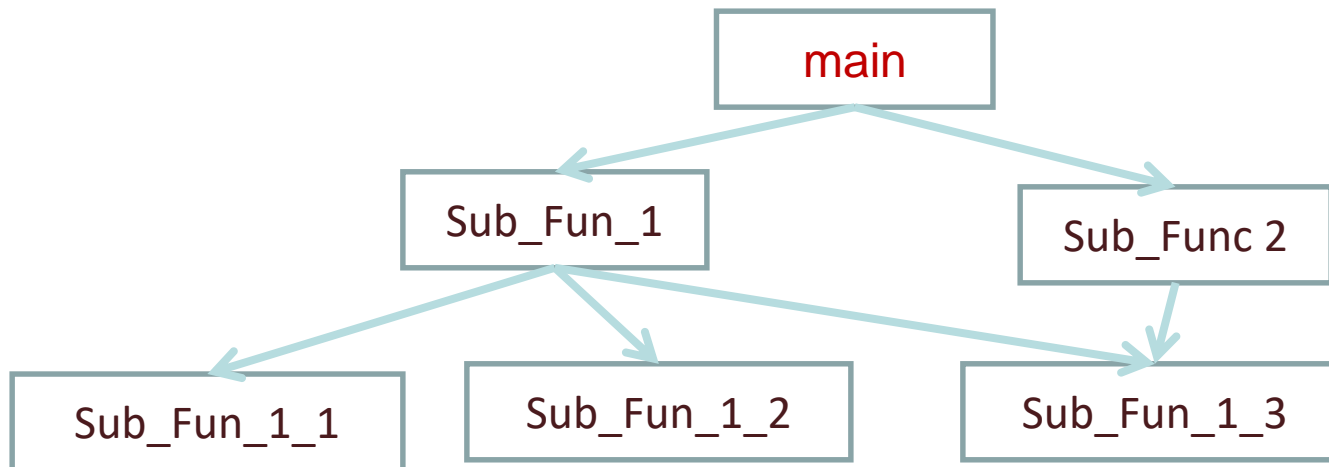
- Divide-and-conquer (分而治之) principle (can be applied in many areas in our daily life)
  - Decompose (分解) a complicated problem into simple ones
  - A complex problem is often easier to solve by dividing it into several smaller sub-problems (tasks)
- This rule can be applied to C programming
  - These smaller sub-problems are sometimes made into **code chunks (代码块)** called **functions** in C.

# Structured Programming

- C language is a structured programming
  - Top-down design (自上而下)
  - Stepwise refinement (逐步精化)
- A C program is a collection of functions
  - In each function
    - 3 basic program structures can be used
      - Sequence
      - Decision
      - Loop
    - Each structure can be embedded in another structure

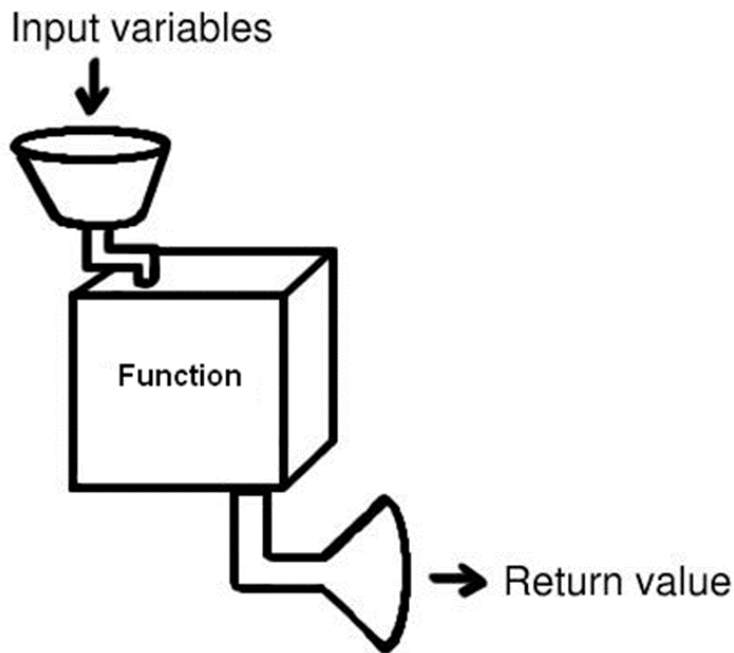
# Structured Programming

- A C program is made up of one or more functions.
  - One and only one **main** function (must)
  - Sub – functions (optional)



# Function

- To write a function, we normally need to specify
  - function **declaration**(声明) (prototype) (原型)
  - function **definition** (定义, code)





# Function Declaration

- A function is a block of program code which deals with a particular task.
- The prototype (also called function declaration) is usually put above main function.
- The purpose is to tell other functions, they can use this function. The code of this function is written somewhere else.
- Function prototype syntax:

`<type> <function_name> (<type_list>) ;`

Type of return value  
(can be "void" if no value is returned)  
e.g, int

List of types of input variables  
(can be empty if there is no input)  
e.g. (int a, char b)

# An Example of Function Prototype

- `float averageGrade(int, int);`

(or written as `float averageGrade (int a, int b);`)

- Function name? `averageGrade`
- How many input variables? two input variables
- What are the types of input variables? `int`
- What is the type of the return value (output value)? `float`

# Class Exercise

- `void printASCIICode(char c);`
  - Function name?
  - How many input variables?
  - What are the types of input variables?
  - What is the type of the return value (output value)

# Function Name

- A function name must be meaningful
  - If we want to write a function to calculate the average of grades, which of the following function names is better?
    - averageGrade
    - Abc
    - aG
    - avaragegrade
    - ag

# Function Definition

- The function definition (code) can be placed anywhere in the program.
- Syntax

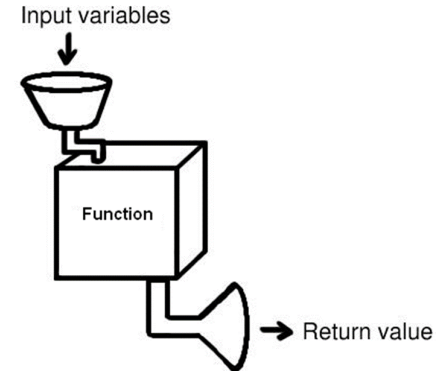
形式参数

```
<type> <function name> (<Formal parameter list>) {  
    <local declarations>  
    <sequence of statements>  
}
```

This part can be the same as  
function prototype

# Examples of Function Definition

```
int sum(int operand1, int operand2){  
    int s;  
    s = operand1 + operand2;  
    return s;  
}
```



```
int absolute(int x){  
    if (x >= 0)  
        return x;  
    else  
        return -x;  
}
```

```
void printASCIIcode(char code){  
    printf("%c 's ASCII code is %d", code, code);  
    return;  
}
```

# Function Call

- After a function is declared and defined, it can be used (called 调用, 使用).
- A function can be called to implement (实现) a task
- Function call can be an independent sentence, or in an expression
- Syntax

实际参数

<**function name**> (<actual parameter list>)



The name of the function  
that has been declared

The list of the input values  
are in the same sequence as in  
the prototype

# An Example of Function Call

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

int main()
{
    printASCIIcode('A');
    printASCIIcode('C');
    printASCIIcode('F');
}

void printASCIIcode(char code)
{
    printf("%c 's ASCII code is %d", code, code);
    return;
}
```

Function prototype  
(declaration)

Actual parameter  
实际参数

Function calls (调用)

Formal parameter  
形式参数

Sub-function  
Definition (code)

Attention: system only executes the main function. A sub-function is executed only when it is called.



# Three Steps in Defining and Using a Function

- First, declare a function using a function prototype (函数声明, 原型)
  - `<type> name(<type_1>,<type_2>,..., <type_n>);`
  - Or `<type> name(<type1 v1>,<type2 v2>,... <type_n vn>);`
- Second, define (code) the function (函数定义, code)
  - `<type> name(<type1 v1>,<type2 v2>,... <type_n vn>);`
- Third, call the function (函数调用)
  - `name(value1, value2,..., valuen);`

# Function prototype, declaration and call


- If the function definition is placed in the same file **before** it is called
  - No function prototype is needed
- If the function definition is placed in the same file **after** it is called or the function is defined in another file
  - The function prototype is required

Part 1: pre-processor

Part 2: function prototypes

Part 3: main() function

Part 4: sub-functions




Structure in .c file

Part 1: pre-processor

Part 2: sub-functions

Part 3: main() function



Structure in .c file

# Suggested: Function Call before Function Definition

```
#include <stdio.h>
int absolute(int); // This prototype is required.

int main() {
    int value, answer;
    scanf("%d", &value);
    answer = absolute(value);
    printf("The absolute value is %d.\n", answer);
    return 0;
}

int absolute(int x) {
    if (x >= 0)
        return x;
    else
        return -x;
}
```

**Function declaration**

**Function call**

**Function definition**

# Function Call after Function Definition

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

```
int absolute(int x){  
    if (x >= 0)  
        return x;  
    else  
        return -x;  
}
```

Function definition

```
int main(){  
    int value, answer;  
    scanf("%d", &value);  
    answer = absolute(value);  
    printf("The absolute value is %d.\n", answer);  
    return 0;  
}
```

Function call

No **prototype** is required.

# Class Exercise

```
#include <stdio.h>

int main() {
    int value;
    scanf("%d", &value);
    printf("The power value is %d.\n", power2(value));
    /*This sentence can also be written as
        int p;
        p = power2(value);
        printf("The power value is %d.\n", p);
    */
    return 0;
}

int power2(int x) {
    return x * x;
}
```

What is the problem with this program? How to solve?

# Variable Scope (变量作用范围)

- Each variable needs to be declared before it is used.
- Each variable has a scope which is the block of code where the variable is valid for use.
- A **global** variable (全局变量) declaration is made outside the bodies of all functions and outside the main program.
  - It is normally placed at the beginning of the program file.
- A **local** variable (局部变量) declaration is one that is made inside the body of a function (or code block).
  - Locally declared variables cannot be accessed outside of the functions (or code block) where they were declared.
  - Formal parameter variables are also local
- It is possible to declare the same variable name in different parts of a program.

# Scope

```
int y = 38; // global variable
```

```
int f(int, int);
```

```
int main( )
```

```
{
```

```
    int z = 47;
```

```
    while(z < 400){
```

```
        int a = 90;
```

```
        z += a++;
```

```
        z++;
```

```
    }
```

```
    y = 2 * z;
```

```
    f(1, 2);
```

```
    return 0;
```

```
}
```

```
int f(int s, int t)
```

```
{
```

```
    int r;
```

```
    r = s + t;
```

```
    int i = 27;
```

```
    r += i;
```

```
    return r;
```

```
}
```

scope of y

scope of f

scope of z

scope of a

scope of s & t

scope of r

scope of i

# An Example

```
#include <stdio.h>
int number; //number: global variable
void doubleNum(int num){ // num: local variable
    int n; // n: local variable
    n = num * 2;
    printf("%d\n", n);
    number = number + 1;
}
int main(){
    int n = 1; // n: local variable
    number = 2;
    doubleNum(n);
    printf("%d\n", number);
    return 0;
}
```

What is the output?



# Disadvantage of Global Variables

- Use of global variables
  - The advantage is that different functions can share the same variable values.
  - **Undisciplined** use of global variables may lead to **confusion** and debugging difficulties.
  - **It is not encouraged to use global variables.**
  - To share the values among the functions, we can pass address to formal parameters or use references.
    - How to pass address will be introduced in the lectures later.
    - Reference will not be introduced in this course.

# Another Example

```
#include <stdio.h>
void foo(int j);
int main(){
    int k = 10;
    foo(k);
    printf("%d", k);
    return 0;
}
void foo(int j){
    j = 0;
}
```

k: actual  
parameter

j: formal  
parameter

```
#include <stdio.h>
void foo(int j);
int main(){
    int k = 10;
    foo(k);
    printf("%d", k);
    return 0;
}
void foo(int k){
    k = 0;
}
```

k: actual  
parameter

k: formal  
parameter

Output?

Attention: Formal parameter's value change in sub-program will NOT affect actual parameter (except array).

# Class Exercise

```
int number; //number: global variable
void increment(int number) // number: local variable
{
    number = number + 1; //use the local number
}
void main()
{
    number = 1; //use the global number
    increment(number); //use the global number
    printf("%d\n", number); //use the global number
}
```

Output?

Attention: This program is not readable. Avoid to use the same variable name in nested scopes.

# Arrays Parameters In Functions

(数组作为函数参数)

```
#include <stdio.h>
// function prototype include variables.
// n is used to accept size of array
void increment(int a[], int n);
int main(){
    int value[4] = {1, 2, 3, 4};
    increment(value, 4); //call with array name and size
    //After above call, values in the array have been changed
    for (int i = 0; i < 4; i++)
        printf("value[%d]= %d\n", i, value[i]);
    return 0;
}
void increment(int a[], int n){
    for (int i = 0; i < n; i++)
        a[i]++;
}
```

Output?

# Pass Arrays to Functions

- In summary
  - any changes in the array in the called function will be reflected in the original array
  - when passing (传递) an array to a function, it is better to set up a parameter to designate the size of array
  - if a formal parameter in a function is an array, the prototype of this function must include parameter names.

# Library Functions

- The standard C library (18 Standard C Headers) contains a large collections of functions that can be called from a C program.
  - [math.h](#)
  - `stdio.h`
  - [stdlib.h](#)
  - [string.h](#)
  - [time.h](#)
  - .....
- See Appendix B of the TEXTBOOK for more information

# Library Functions

- A library is a collection of precompiled object files which can be linked into programs
- Three steps
  - Find the header file which includes the library function to call
  - Use include directive `#include` to encompass the header file
  - Call the function in the program

```
#include<stdio.h>
#include<math.h>
int main() {
    double x = sqrt(2.0); //sqrt: declared in math.
    printf("x = %f", x); //printf: declared in stdio.h
    return 0;
}
```

More functions in math.h:

<https://baike.baidu.com/item/math.h/10991856?fr=aladdin#3>

# Advantages of Functions

- Functions make programs easier to understand.
- Functions can be called several times in the same program, allowing the code to be **reused**.



# Advantages of Functions

```
#include <stdio.h>
int absolute(int);

int main()
{
    int value1, value2;
    int answer1, answer2;
    scanf("%d %d", &value1, &value2);
    answer1 = absolute(value1);
    answer2 = absolute(value2);
    printf("The absolute values are %d, %d.\n", answer1,
          answer2);
    return 0;
}

int absolute(int x) {
    if (x >= 0)
        return x;
    else
        return -x;
}
```

If we do not use function calls,  
How to write the program?

# Placement of Functions (函数的安置)

- For a small program, we can put the functions in **one source file (.c file)**
- For large programs, we can
  - manage functions in multiple **.c** files
  - write **.h** files containing all the prototypes of the functions
  - include the header files in the files that use the functions.

# An Example with Three Files and Four functions

myProgram.c

mymath.h

```
int min(int x, int y);  
int max(int x, int y);
```

mymath.c

```
int min(int x, int y)  
{  
    return x > y? y: x;  
}  
  
int max(int x, int y)  
{  
    return x > y? x: y;  
}
```

```
#include<stdio.h>  
#include "mymath.h"  
int sum(int op1, int op2);  
int main(){  
    int num1, num2;  
    scanf("%d%d", &num1, &num2);  
    printf("the max number is  
           %d\n", max(num1, num2));  
    printf("the min number is  
           %d\n", min(num1, num2));  
    printf("the sum is %d\n",  
           sum(num1, num2));  
    return 0;  
}  
int sum(int op1, int op2){  
    return (op1 + op2);  
}
```

# Perspective of a Program

- A program is comprised of functions (logically).
- A program is comprised of files (physically).
- When one program is placed in several files, we can put logically coherent functions in one source file.

# Summary

- A program is comprised of one or more functions. Functions make the program more modular
- A function has declaration, definition and call
- A function can have parameters. It can also return a value
- Attention should be paid to the relationship between actual parameters and formal parameters
- A variable can be used in its scope
- A program can be managed in several files.