Function

Index

Function

Definition

A function is a code block for executing a specific task

Suppose, you should write a program to generate a canvas and brush it. Hence, you can define two functions to solve the issue

- ➢ canvas function
- ➢ brush function

Dividing a complex issue into smaller chunks makes a program easy to reuse and understand

Function

User-defined Functions

The function is a small program in essential, with its own declarations and statements

Advantages

- ➢ Make a program easier to read and alter
- ➢ Avoid duplicating code by reusing
- ➢ Be called by any program

must be declared

```
(return type) function_name (parameters)
{
    declarations
    statements
}
```

Function

User-defined Functions

Some programmers move the *return* type above the function name

```
(return type)
function_name (parameters)
{
    declarations
    statements
}
```

When? The return type is lengthy as the following

unsigned long int

Function

User-defined Functions

An *average* function

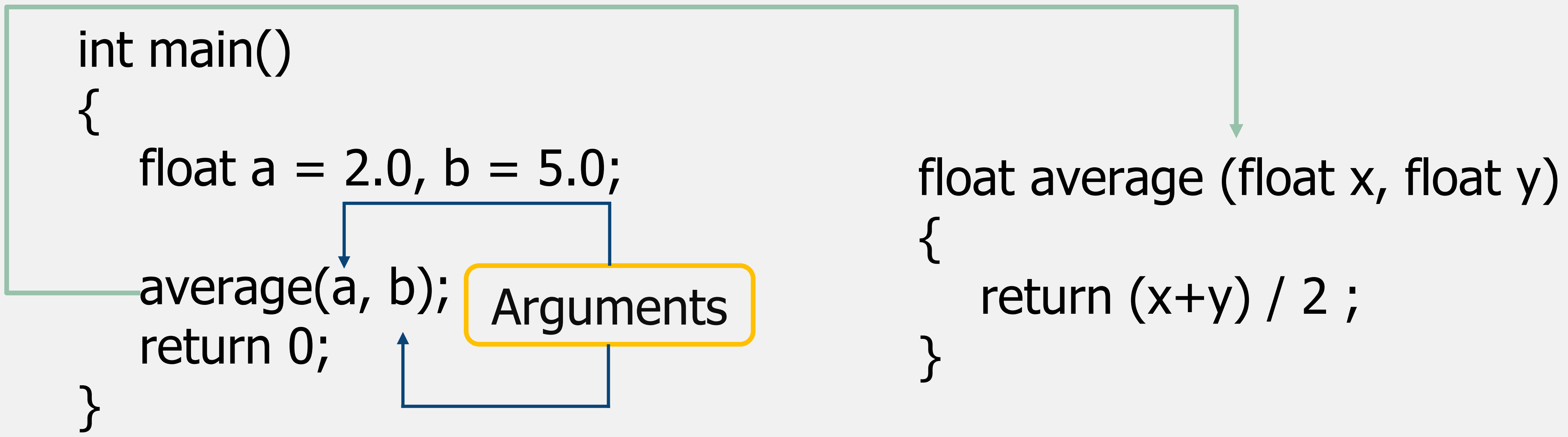
```
float average (float x, float y)
{
    return (x+y) / 2 ;
}
```

- The float is the return type of the function average
- The identifiers (or the function's parameters), x and y, represent the numbers which will be provided if the average function is called

Function

User-defined Functions

A function is called with the combination of function name and a list of arguments



```
int main()
{
    float a = 2.0, b = 5.0;

    average(a, b);
    return 0;
}
```

```
float average (float x, float y)
{
    return (x+y) / 2 ;
}
```

Arguments are used to provide information to a function

- If calling average(a, b), the values of *a* and *b* will be *copied* into the parameters x and y

Function
User-defined Functions

A function is called in the place where needing to use the return value

```
int main()
{
    float a = 2.0, b = 5.0;
    printf("Average is: %f\n", average(a, b));
    return 0;
}
```

The return value is temporary and will be discarded

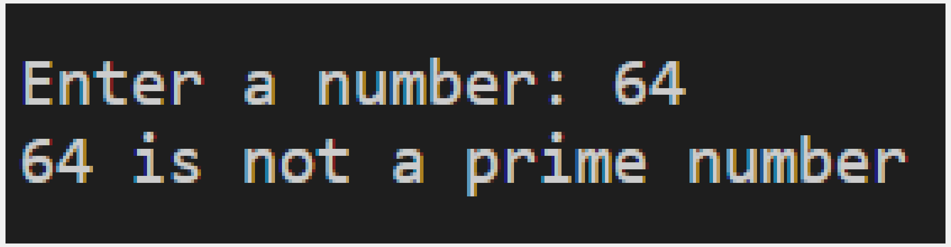
For this issue, the return value can be captured in a variable

```
avg = average(a, b);
printf("Average is: %f\n", avg);
```

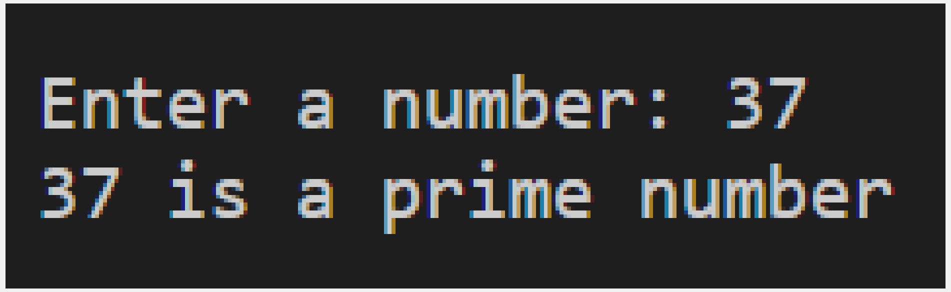
Function

User-defined Functions

Write a program to determine a number as prime or non-prime



```
Enter a number: 64
64 is not a prime number
```



```
Enter a number: 37
37 is a prime number
```

Function

Function Type

A function name is followed by a list of parameters, and each parameter is preceded by a specification of its type

Parameters are separated by commas

If there is no parameter in the function, the void should appear between the parentheses

```
(return type) function_name (pa_1, pa_2, …)
{
   declarations
   statements
}
```

```
(return type) function_name (void)
{
   declarations
   statements
}
```

Function

Function Type

1. No arguments passed and no return value

```
void get_num (void)
{
    int x = 3;
    …
}
```

2. No arguments passes but returns a value

```
int get_num (void)
{
    int x = 3;
    …
    return 2 * x;
}
```

Function

Function Type

3. Arguments passed but no return value

```
void get_num (int n)
{
    int x = n;
    …
}
```

4. Arguments passed and returns a value

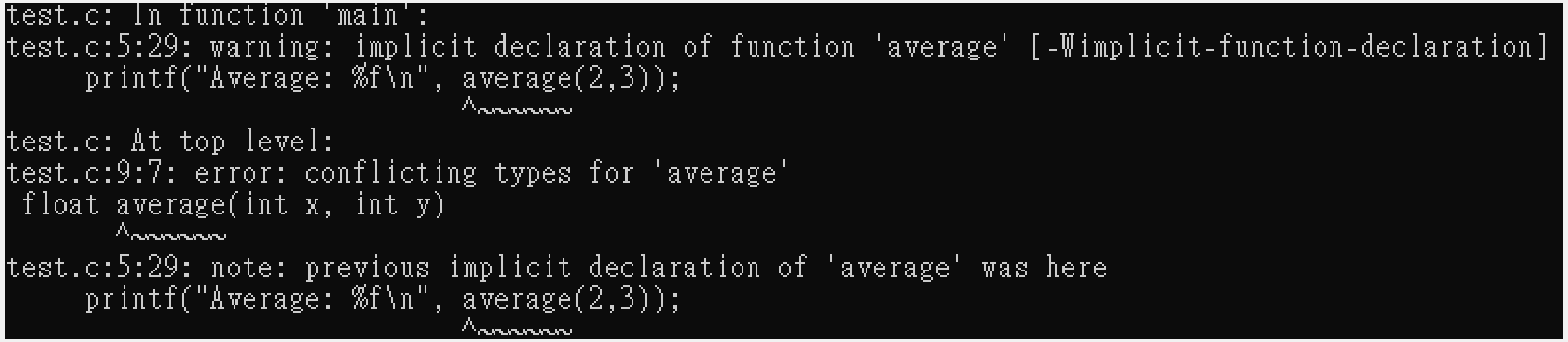
```
int get_num (int n)
{
    int x = n;
    …
    return 2 * x;
}
```

Function
Function Type

If calling a function before defining it, error messages will be shown by the compiler

- ➢ The compiler will assume that average returns an int value

```
int main()
{
    float a = 2.0, b = 5.0;
    printf("Average is: %f\n", average(a, b));
    return 0;
}
float average (float x, float y)
{
    return (x+y) / 2 ;
}
```



```
test.c: In function 'main':
test.c:5:29: warning: implicit declaration of function 'average' [-Wimplicit-function-declaration]
     printf("Average: %f\n", average(2,3));
                             ^~~~~~~
test.c: At top level:
test.c:9:7: error: conflicting types for 'average'
 float average(int x, int y)
       ^~~~~~~
test.c:5:29: note: previous implicit declaration of 'average' was here
     printf("Average: %f\n", average(2,3));
                             ^~~~~~~
```

Function

Function Type

How to avoid?

- Declare a function before calling it to provide the compiler with a brief glimpse
- The declaration must be consistent with the function's definition
- General form of function declaration

(*return type*) *function_name* (*parameters*);

```
float average (float x, float y);
int main()
{
    float a = 2.0, b = 5.0;
    printf("Average is: %f\n", average(a, b));
    return 0;
}
```

Function

Function Type

Function prototypes

- A function declaration which only specifies the parameter type

```
float average (float x, float y);
```

⇕

```
float average (float, float);
```

Function

Function Type

In C, arguments are passed by value: when calling a function, each value of argument is assigned to the corresponding parameter

```
int main()
{
    float a = 2.0, b = 5.0;
    printf("Average is: %f\n", average(a, b));
    return 0;
}

float average (float x, float y)
{
    return (x+y) / 2 ;
}
```

x = a, y = b;

Function

Function Type

Arguments

- When the *pointer* is not used, the parameter only contains a copy of argument's value and any changes in parameter doesn't affect the argument

Advantage

- Parameter and argument do not affect each other

Disadvantage

- The advantage is the disadvantage

Function

Function Type

C allows calls in which the types of the arguments don’t match the types of the parameters

The rules governing how the arguments are converted depend on whether or not the compiler has seen a prototype for the function (or the function’s full definition) prior to the call

- The compiler has encountered a prototype prior to the call
  - The type of argument is converted to the type of the corresponding parameter
- The compiler has not encountered a prototype prior to the call
  - The compiler performs the default argument promotions

float converts to ***double***
char or ***short*** convert to ***int***

Function

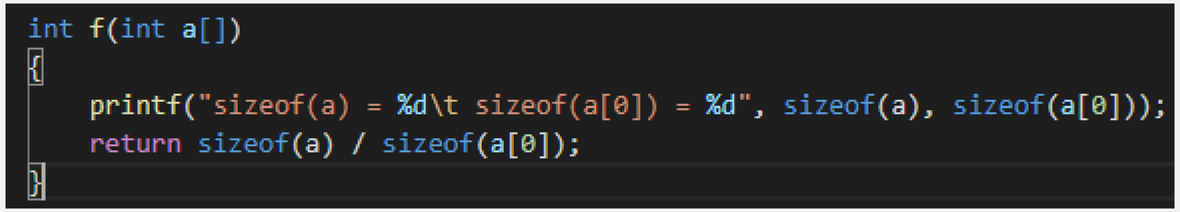
Function Type

Array arguments

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

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

- However, there is no any easy way in C for a function to determine the length of an array passed to it



```
int f(int a[])
{
    printf("sizeof(a) = %d\t sizeof(a[0]) = %d", sizeof(a), sizeof(a[0]));
    return sizeof(a) / sizeof(a[0]);
}
```

sizeof(a) = 4   sizeof(a[0]) = 4

Function

Function Type

The prototype for a function with the array arguments

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

⇕

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

Function

Function Type

When a function is called, the array argument will be the name of the passed array

```
#define LEN 100

int main(void)
{
    int b[LEN], total;
    ...
    total = f(b, LEN);
    ...
}

total = f (b[], LEN);          // Error
```

Function

Function Type

A function is allowed to change the elements of the array parameter, and the change is reflected in the corresponding argument

An array is initialized by the initial_zero function into each element

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

int main()
{
    int b[100];
    initial_zeros(b, 100);
    return 0;
}
```

Function

Function Type

If a parameter is a multidimensional array, only the length of the first dimension may be ignored

```
#define LEN 10
void initial_zeros(int a[][LEN], int n)
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < LEN, j++)
            a[i][j] = 0;
}

int main()
{
    int b[LEN][LEN];
    initial_zeros(b, LEN);
    return 0;
}
```

Function

Function Type

However, it is a nuisance for not being able to pass multidimensional arrays with an arbitrary number of columns

```
#define LEN 10

void initial_zeros(int a[][LEN], int n)
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < LEN, j++)
            a[i][j] = 0;
}

int main()
{
    int b[LEN][LEN];
    initial_zeros(b, LEN);
    return 0;
}
```

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

int main()
{
    int b[LEN][LEN];
    initial_zeros(b, LEN);
    return 0;
}
```

Function

Exit

Return

- ➢ A non-void function must use the return statement to specify what value it will be return

```
return expression;
```

The expression is often a constant, variable, complex expression, or function, such as

```
return 0;
return status;
return n > 0? n : 0;
return average(a, b);
```

Function
Exit

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

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

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

Function

Exit

Executing a return statement in main is one way to terminate a program

Another way is the exit function calling, which belongs to <stdlib.h>

To indicate normal termination

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

||

```
exit(EXIT_SUCCESS);  //normal termination
```

To indicate abnormal termination

```
exit(EXIT_FAILURE);  //abnormal termination
```

Function

Exit

In main function

return expression; = *exit*(*expression*);

The difference between return and exit is that exit causes program termination regardless of which function calls it

The return statement make program termination only when it execute in the main function

Function
Recursion

A function is recursive if it calls itself

The following function computes n! recursively

$n! = n * (n-1)!$

```
int fact(int n)
{
   if (n <= 1)
      return 1;
   else
      return n * fact(n - 1);
}
```

Function

Recursion

How recursion works?

```
x = fact(3);

int fact(int n)
{
   if (n <= 1)
       return 1;
   else
       return n * fact(n - 1);
}
```

fact(3) finds that 3 is not less than or equal to 1, and it calls fact(2)
fact(2) finds that 2 is not less than or equal to 1, and it calls fact(1)
fact(1) finds that 1 is less than or equal to 1, and it returns 1,
causing fact(2) to return 2 × 1 = 2, and
causing fact(3) to return 3 × 2 = 6

Function

Recursion

We can condense the fact function by putting a conditional expression in the return statement

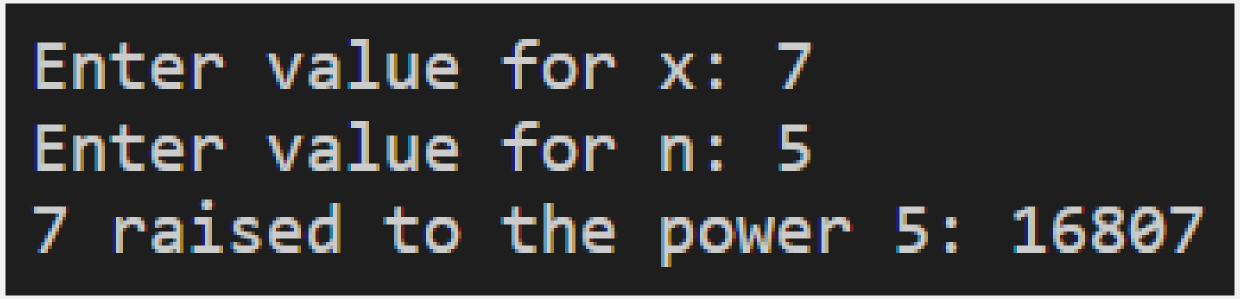
```
int fact(int n)
{
   return n > 1?  n* fact(n - 1) : 1;
}
```

All recursive functions need some kind of termination condition to avoid infinite recursion

Function

Recursion

Write a power computation program using recursion



```
Enter value for x: 7
Enter value for n: 5
7 raised to the power 5: 16807
```