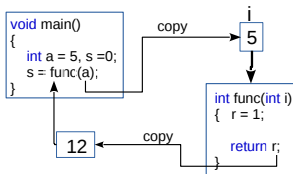


C Programming

Lecture 5: Functions and MACROs



Lecturer: *Dr. Wan-Lei Zhao*
Spring Semester 2022

- 1 Functions: declaration, definition and calling
- 2 Recursive Functions
- 3 Visibility and Life-cycle of Variables
- 4 Precompilation Instructions and Macros

- Functions we know

```
1 int main(.);  
2 int printf(..);  
3 int scanf(.);  
4 float sqrt(.);  
5 float floor(.);  
6 float fabs(.);
```

- Functions in math

$$f(x) = \sin(x)$$

$$g(x) = x^2$$

- They are actually comparable
- Function in C is more general
- We are going to learn to organize our codes into functions (blocks)

Advantages of function (1)

- We are already familiar with functions

```
1 int main(.); //entrance of the program
2 int printf(..); //print things onto screen
3 int scanf(.); //read input from keyboard
4 float sqrt(.); //take square root
5 float floor(.); //take maximum number smaller than input
6 float fabs(.); //take absolute value of a float number
```

- Advantages
 - No need to repeat others work (reinvent the wheel)
 - No need to write things again and again
 - Your codes become cleaner

Introduction of function (1)

- Let's start with a simple example

```
1 #include <stdio.h>
2 void hi(int i) //<—declaration of function "hi"
3 {
4     printf(" Hello %d\n", i);
5 }
6
7 int main()
8 {
9     int i = 0;
10    for(i = 0; i < 5; i++)
11        hi(i); //<— call function hi(int i)
12    return 0; //return value to the one who calls it
13 }
```

- We call **hi()** inside main
- "main()"** cannot be called by any other function

Declaration of function (1)

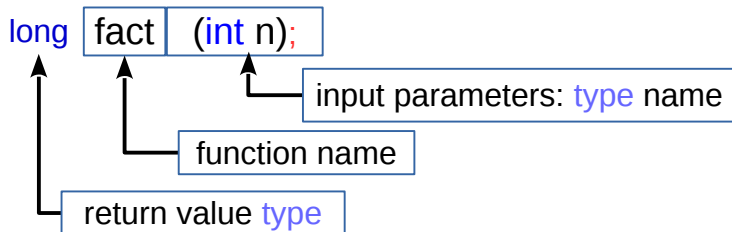
- Declare a function for $n!$

```
1 long fact(int i); //← this is the declaration
2
3 int main()
4 {
5     int i = 5, f = 0;
6     f = fact(i);
7     return 0; //return value to the one who calls it
8 }
```

- The name should be **unique**
- There is should be input parameter(s) along with the types
- There is should be output value type

Declaration of function (2)

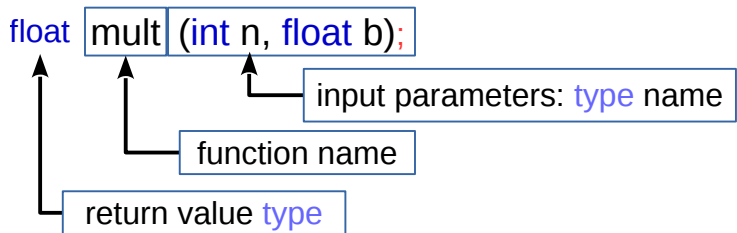
- Declare a function for $n!$



- The name should be **unique**
- There should be input parameter(s) along with the types
- There should be output value type
- If there is nothing, the returning type is `int`

Declaration of function (3)

- Declare a function for $n!$



- The name should be **unique**
- There should be input parameter(s) along with the types
- There should be output value type
- If there is nothing, the returning type is `int` by default

Define a function (1)

- Declare a function for $n!$

```
1 long fact(int i); //<— this is the declaration
2
3 int main()
4 {
5     int i = 5, f = 0;
6     f = fact(i);
7     return 0; //return value to the one who calls it
8 }
```

error: undefined reference to 'fact'

- “fact” has been declared, however not defined (implemented)
- There is no function body
- When you compile it, above error comes out

Define a function (2)

- Declare a function for $n!$

```
1 long fact(int i); //← this is the declaration
2
3 int main()
4 {
5     int i = 5;
6     long f = 0;
7     f = fact(i);
8     return 0; //return value to the one who calls it
9 }
```

- Now, let's think about how to implement fact()

Define a function (3)

```
long func1(int n, int i)
{
    .....
    return r;
}
```

- You need put function implementation inside the brackets “{”

Define a function (4)

- Now, let's think about how to implement fact()
 - 1 For i from n to 1 do
 - 2 $r = r * i$
 - 3 $i --$
 - 4 End-for
 - 5 return r

Define a function (4): separate declaration from definition

```
1 long fact(int i);
2 int main()
3 {
4     int i = 5;
5     long f = 0;
6     f = fact(i);
7     return 0;
8 }
9 long fact(int i)
10 {
11     long n = 1;
12     if(i < 0)
13         return 0;
14     else if(i == 0)
15         return 1;
```

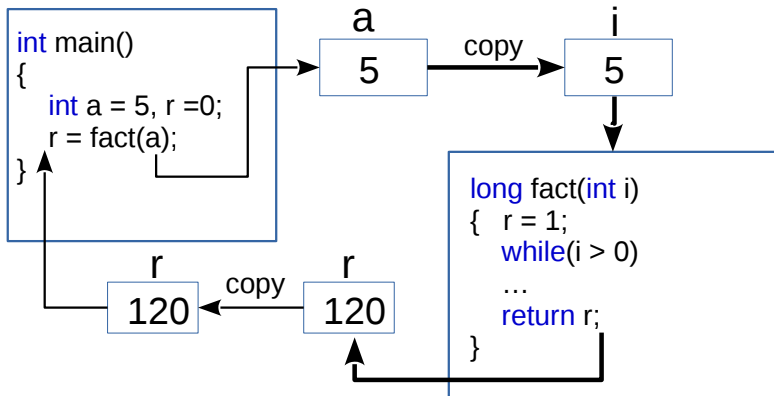
```
16     else
17     {
18         while(i > 0)
19         {
20             n = n*i;
21             i--;
22         }
23     }
24     return n;
25 }
```

Define a function (5): combine declaration with definition

```
1 long fact(int i)
2 {
3     long n = 1;
4     if(i < 0)
5         return 0;
6     else if(i == 0)
7         return 1;
8     else
9     {
10         while(i>0)
11         {
12             n = n*i;
13             i--;
14         }
15     }
16     return n;
17 }
```

```
18 int main()
19 {
20     int i = 5;
21     long f = 0;
22     f = fact(i);
23     return 0;
24 }
```

Function Calling

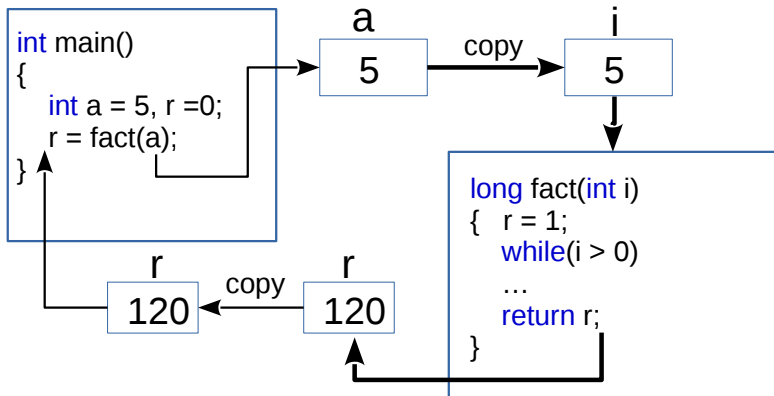


- Parameters are transferred in by value not **by address**

- Principles in function definition

- ① Remember return type, if there is no need, put `void`
- ② Give a unique and self-telling name to your function
- ③ Define function first, then you can call it (just as variable in C)
- ④ Parameters along with the type appear in pair

Parameter Transfer (1)



- Parameters are transferred in **by value** not **by address**

Parameter Transfer (2)

- Let's consider a simple coding problem
- Given integers a and b
- You are required to swap their values
- For example, $a = 5$, $b = 8$
- After swapping, it becomes $a = 8$, $b = 5$

Parameter Transfer (3)

- You are required to swap their values
- For example, $a = 5$, $b = 8$
- After swapping, it becomes $a = 8$, $b = 5$

```
1 int main()  
2 {  
3     int a = 5, b = 8;  
4     int tmp;  
5     printf("a=%d, b=%d\n", a, b);  
6     tmp = a; a = b;  
7     b = tmp;  
8     printf("a=%d, b=%d\n", a, b);  
9     return 0;  
10 }
```

Parameter Transfer (4)

- Now, let's do it by a function

```
1 #include <stdio.h>
2 void swap(int a, int b)
3 {
4     int tmp = a;
5     a = b; b = tmp;
6     return ;
7 }
8 int main()
9 {
10     int a = 5, b = 8;
11     printf("a=%d, b=%d\n", a, b);
12     swap(a, b);
13     printf("a=%d, b=%d\n", a, b);
14 }
```

Parameter Transfer (5)

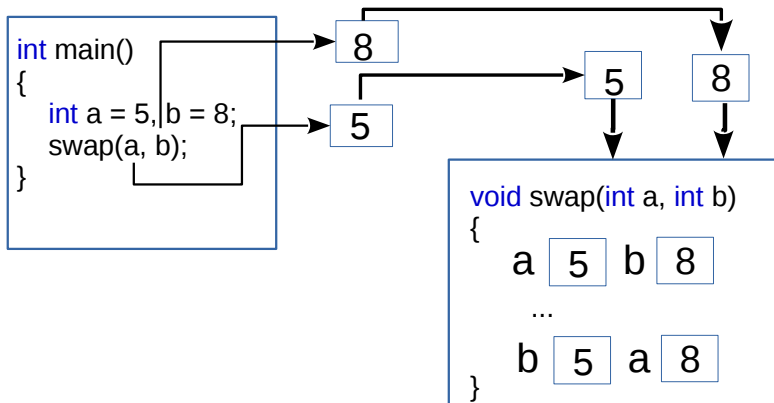
- The result is against our will, why???

```
1 #include <stdio.h>
2 void swap(int a, int b)
3 {
4     int tmp = a;
5     a = b; b = tmp;
6     return ;
7 }
8 int main()
9 {
10     int a = 5, b = 8;
11     printf("a=%d, b=%d\n", a, b);
12     swap(a, b);
13     printf("a=%d, b=%d\n", a, b);
14     return 0;
15 }
```

[Output:]

```
1 a = 5, b = 8
2 a = 5, b = 8
```

Parameter Transfer (6)



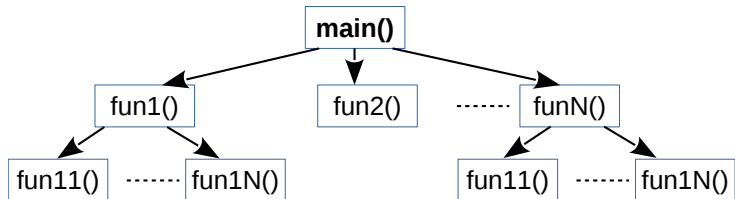
- Parameters are transferred in **by value** not **by address**

Parameter Transfer (7)

- Parameters are transferred in **by value** not **by address**
- Arguments and Parameters should be matched

```
1 float calc(int n, float a, short int c)
2 {
3     return (a*a*n+c);
4 }
5 int main()
6 {
7     int n = 2;
8     short int w = 4;
9     float x = 4.12, r = 0;
10    r = calc(n, x, w);
11    return 0;
12 }
```

Function Calling again (1)



- Function can be called in a cascaded manner
- 'main' cannot be called
- Functions are not necessarily called by 'main' directly

Function Calling again (2)

- Parameters are transferred in **by value** not **by address**
- Arguments and Parameters should be matched

```
1 float calc(int n, float a, short int c)
2 {
3     return (a*a*n+c);
4 }
5 int main()
6 {
7     int n = 2;
8     short int w = 4;
9     float x = 4.12, r = 0;
10    r = 3*calc(n, x, w);
11    return 0;
12 }
```

Example 1: perfect number (1)

- 1. Define a function to judge whether an integer is a **perfect number**
- Perfect number: number equals to the sum of all its factors
- $6 = 1 + 2 + 3$
- 2. Call it to output all the perfect numbers in range [2, 300]

Think about this problem in 5 minutes...

Example 1: perfect number (2)

- 1. Define a function to judge whether an integer is a **perfect number**
 - Perfect number: number equals to the sum of all its factors
 - $6 = 1 + 2 + 3$
 - 2. Call it to output all the perfect numbers in range [2, 300]
- ① Given a number
 - ② We should work out all its factors
 - ③ Sum all the factors up
 - ④ See whether it is equal to the number
 - ⑤ We should use % operator a lot

Example 1: perfect number (3)

- 1. Define a function to judge whether an integer is a **perfect number**
- **Perfect number**: number equals to the sum of all its factors
- $6 = 1 + 2 + 3$
- 2. Call it to output all the perfect numbers in range [2, 300]
- Steps:
 - ① Give n
 - ② For i from 2 to n do
 - ③ check whether n is dividable by i
 - ④ if yes, sum up
 - ⑤ Check whether sum equals to n
 - ⑥ Return 1 or 0
- Let's do it now!!

Example 1: perfect number (4)

- 1 Give n
- 2 For i from 2 to n do
- 3 check whether n is dividable by i
- 4 if yes, sum up
- 5 Check wether sum equals to n
- 6 Return 1 or 0

```
1 int isPerfect(int n)
2 {
3     int i = 0, sum = 1;
4     int up = ceil(n/2.0);
5     for(i = 2; i < up; i++)
6     {
7         if(n%i == 0)
8         {
9             sum += i;
10        }
11    }
12    if(sum == n)
13        return 1;
14    else
15        return 0;
16 }
```

Example 1: perfect number (5)

```
1 #include <stdio.h>
2 #include <math.h>
3 int isPerfect(int n)
4 {
5     int i = 0, sum = 1;
6     int up = ceil(sqrt(n));
7     for(i = 2; i < up; i++)
8     {
9         if(n%i == 0)
10        {
11            sum += i;
12        }
13    }
14    if(sum == n)
15        return 1;
16    else
17        return 0;
18 }
```

```
19 int main()
20 {
21     int i = 0;
22     for(i = 2; i <= 300; i++)
23     {
24         if(isPerfect(i))
25         {
26             printf("%d\n", i);
27         }
28     }
29     return 0;
30 }
```

- 1 Functions: declaration, definition and calling
- 2 Recursive Functions**
- 3 Visibility and Life-cycle of Variables
- 4 Precompilation Instructions and Macros

Recursive Function (1)

- We already know that function is allowed to call any other function
- Function is allowed to call itself, this is called **recursive**
- It looks like following

```
1 int func2(int n);  
2 int func1(int n)  
3 {  
4     int a = 2*func1(n-2);  
5     ...  
6     int b = func2(n-3);  
7     return (a+b);  
8 }
```

- Noticed that “func1” has been called inside “func1”
- The scale of the problem decreases in each calling

Recursive Function: how it works (1)

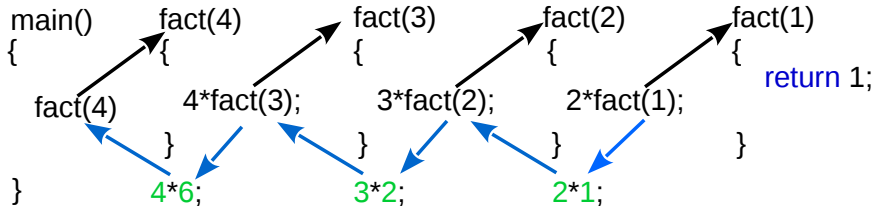
```
1 long fact(int n)
2 {
3     long a = 0;
4     if (n < 0)
5         a = 0;
6     else if (n == 1 || n == 0)
7         a = 1;
8     else
9         a = n * fact(n-1);
10
11     return a;
12 }
```

```
1 long fact(int n)
2 {
3     long a = 1;
4     int i = 0;
5     if (n < 0)
6         return 0;
7     for (i = n; i > 0; i--)
8     {
9         a = a * i;
10    }
11    return a;
12 }
```

```
1 int main()
2 {
3     int n = 4, b = 0;
4     b = fact(n);
5     printf("fact(%d) = %d\n", n, b);
6     return 0;
7 }
```

Recursive Function: how it works (2)

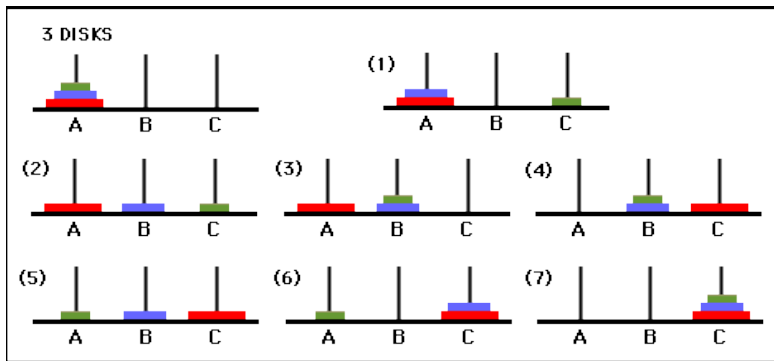
- “fact” calls itself until the **bottom** is reached
- Noticed that the scale of the problem decreases gradually
- Advantage: simple
- Darkside: requires a lot of memory



- Suggestion: try to avoid to use recursive function

Recursive Function: Hanoi Tower Problem

- One is allowed to move one disc from one beam to another a day
- Move all 64 discs from beam A to C



- It would not be fulfilled even till the end of this world!!

Source code for Hanoi Tower (1)

```
1 #include <stdio.h>
2 void hanoi(int n, char b1, char b2, char b3)
3 {
4     if(n == 1)
5     {
6         printf("%c————>%c\n", b1, b3);
7     } else if(n == 2)
8     {
9         printf("%c————>%c\n", b1, b2);
10        printf("%c————>%c\n", b1, b3);
11        printf("%c————>%c\n", b2, b3);
12    } else {
13        hanoi(n-1, b1, b3, b2);
14        printf("%c————>%c\n", b1, b3);
15        hanoi(n-1, b2, b1, b3);
16    }
17 }
```

Source code for Hanoi Tower (2)

```
19 int main()
20 {
21     int n = 20;
22     printf("Input n: ");
23     scanf("%d", &n);
24     hanoi(n, 'A', 'B', 'C');
25 }
```

- 1 Move top **n-1** plates from **A** to **B** via **C**
- 2 Move the **bottom one** to **C**
- 3 Move **n-1** plates from **B** to **C** via **A**

- 1 Functions: declaration, definition and calling
- 2 Recursive Functions
- 3 Visibility and Life-cycle of Variables**
- 4 Precompilation Instructions and Macros

Visibility and Life-cycle of Variables (1)

- We take something for granted before
- Now we study them in detail
 - 1 Could we use the same variable name in different functions?
 - 2 Could we use the same variable name in the same functions?
 - 3 Could different functions share the same variable?
 - 4 When a variable is born, when it dies??

Visibility and Life-cycle of Variables (2)

❶ Could we use the same variable name in different functions?

```
1 int func1(int n)
2 {
3     int r = 3, a = 1;
4     return (r*n+a);
5 }
6 float func2(int n, float a)
7 {
8     float r = 1;
9     int i = 0;
10    for(i = 0; i < n; i++)
11    {
12        r = r*a;
13    }
14    return r;
15 }
```

- The answer is **Yes**
- The visibility is inside function only
- It is born when the function is called
- It dies when calling is done

Visibility and Life-cycle of Variables (3)

② Could we use the same variable name in the same function?

```
1 float func2(int n, float a)
2 {
3     float r = 1;
4     int r = 0;
5     int i = 0;
6     float i = 0;
7     for(i = 0; i < n; i++, r++)
8     {
9         r = r*a;
10    }
11    return r;
12 }
```

- The answer is **No**
- Codes on the left cannot pass the compilation
- Basically, it is ambiguous
- Imagine there are two **Li Mins** in your class

Visibility and Life-cycle of Variables (4-1)

③ Could different functions share the same variable?

```
1 int x, y;  
2 void swap()  
3 {  
4     int t;  
5     t = x; x = y; y = t;  
6     return ;  
7 }  
8 int main()  
9 {  
10    x = 3, y = 5;  
11    swap();  
12    printf("x=%d\n", x);  
13    printf("y=%d\n", y);  
14    return 0;  
15 }
```

- The answer is **Yes**
- They are called global variables
- They are visible to all functions in this **file**
- They are defined outside of functions
- They are born when “main” is called
- They die when calling of “main” complete

Visibility and Life-cycle of Variables (4-2)

③ Could different functions share the same variable?

```
1 #include <stdio.h>
2 int x, y;
3 void swap()
4 {
5     int t;
6     t = x; x = y; y = t;
7     return ;
8 }
9 int main()
10 {
11     x = 3, y = 5;
12     swap();
13     printf("x=%d\n", x);
14     printf("y=%d\n", y);
15     return 0;
16 }
```

```
1 #include <stdio.h>
2 void swap(int a, int b)
3 {
4     int tmp = a;
5     a = b; b = tmp;
6     return ;
7 }
8 int main()
9 {
10     int a = 5, b = 8;
11     swap(a, b);
12     printf("a=%d, b=%d", a, b);
13     return 0;
14 }
```

Visibility and Life-cycle of Variables (5)

5 When a variable is born, when it dies??

```
1 int incr(int a)
2 {
3     static int x = 3;
4     x = x + a;
5     //printf("x = %d\n", x);
6     return x;
7 }
8 int main()
9 {
10     int i = 0, a = 0;
11     for(i = 0; i < 4; i++)
12     {
13         a = incr(i);
14         printf("a = %d\n", a);
15     }
16     return 0;
17 }
```

- When you put “**static**” before a local variable
- Its life-cycle becomes as long as global variable
- It is born when “main” is called
- It dies when calling of “main” complete
- However, it is only visible within the function

Visibility and Life-cycle of Variables (6)

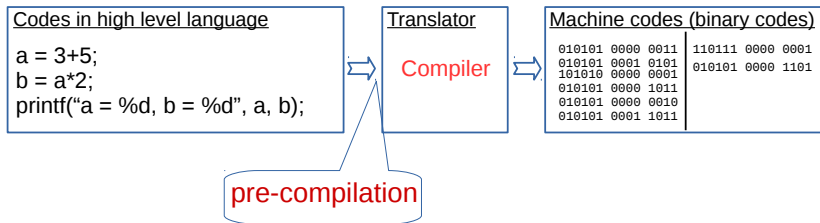
Variable types	inside a function		outside a function	
	visibility	life cycle	visibility	life cycle
auto and register	√	√	X	X
static (inside)	√	√	X	√
Static (outside)	√	√	√ (within the file)	√
extern	√	√	√	√

- It is NOT recommended to use global variables
- Advantage: you can transfer value easily
- Darkside
 - You do NOT know where they have been changed
 - Hard to debug your code
 - Your code will be very messy!!!

- 1 Functions: declaration, definition and calling
- 2 Recursive Functions
- 3 Visibility and Life-cycle of Variables
- 4 Precompilation Instructions and Macros**

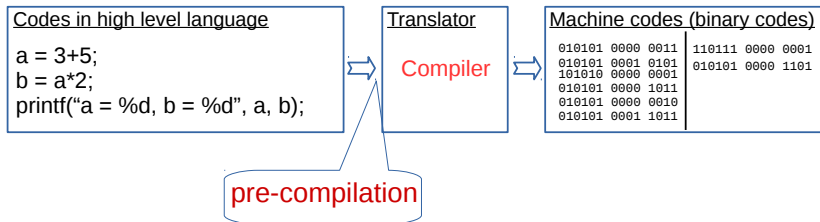
Precompilation: the Concept (1)

- It happens before we compile codes to binary
- Preprocess the codes
- There are instructions we use to communicate with the compiler



- They are executed before compilation is undertaken

Precompilation: the Concept (2)



- There are instructions we use to communicate with the compiler
- They all start with “**#**”, pronounced as “sharp”
 - 1 **#include** header file or full path of file
 - 2 **#define** MACRO
 - 3 **#if...#else** or **#if...#else if** MACRO
 - 4 **#ifndef** MACRO
 - 5 **#endif**

Precompilation instruction: `#include` (1)

- It tells the compiler following thing
 - ① A header file is required to compile the code
 - ② In the header file, the function that is called in the code is declared
 - ③ Where the compiler is able to find the file

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

```
1 #include "myfunc.h"
```

- `<stdio.h>` tells the compiler to search in the system default path
- `"myfunc.h"` tells the compiler to 1. search in the directed path, 2. then go to system default path

Precompilation instruction: #include (2)

[myfunc.h]

```
1 float mypow(float base, int n)
2 {
3     float r = 1;
4     int i = 0;
5
6     if(n == 0)
7         return r;
8
9     for(i = 1; i <= n; i++)
10     {
11         r = base*r;
12     }
13     return r;
14 }
```

[main.c]

```
1 #include "myfunc.h"
2 #include <stdio.h>
3 int main()
4 {
5     float r = mypow(3.14, 3);
6     printf("r = %f\n", r);
7     return 0;
8 }
```

Instruction for Macros: `#define` (1)

- `#define` allows user to define constants or functions
- These constants and functions can be later called in the code
- As a convention, we CAPITALIZE everything
- However, it is possible that PI is defined elsewhere

```
1 #define PI 3.1415926
2 #include <stdio.h>
3 int main()
4 {
5     float a = 0, r = 4.5;
6     a = PI*r*r;
7     printf("a=%f\n", a);
8     return 0;
9 }
```



After pre-compilation

```
1 #define PI 3.1415926
2 #include <stdio.h>
3 int main()
4 {
5     float a = 0, r = 4.5;
6     a = 3.1415926*r*r;
7     printf("a=%f\n", a);
8     return 0;
9 }
```

Instruction for Macros: `#define` (2)

- However, it is possible that PI is defined elsewhere

```
1 #define PI 3.1415926
2 #include <stdio.h>
3 int main()
4 {
5     float a = 0, r = 4.5;
6     a = PI*r*r;
7     printf("a=%f\n", a);
8     return 0;
9 }
```

```
1 #ifndef PI
2 #define PI 3.1415926
3 #endif
4 #include <stdio.h>
5 int main()
6 {
7     float a = 0, r = 4.5;
8     a = PI*r*r;
9     printf("a=%f\n", a);
10    return 0;
11 }
```

Instruction for Macros: `#define` (3)

- Pay attention that the constant has NO type
- We can similarly define Macro function

```
1 #define MULT(x,y) x*y+y
2 #include <stdio.h>
3 int main()
4 {
5     float a = 2, r = 4.5;
6     a = MULT(a, r);
7     printf("a=%f\n", a);
8     return 0;
9 }
```

```
1 #ifndef MULT
2 #define MULT(x,y) x*y+y
3 #endif
4 #include <stdio.h>
5 int main()
6 {
7     float a = 2, r = 4.5;
8     a = MULT(a, r)*4;
9     printf("a=%f\n", a);
10    return 0;
11 }
```

- Please work out the output for each ...

Instruction for Macros: `#define` (4)

- Pay attention that the constant has NO type
- We can similarly define Macro function

```
1 #ifndef MULT
2 #define MULT(x,y) x*y+y
3 #endif
4 #include <stdio.h>
5 int main()
6 {
7     float a = 2, r = 4.5;
8     a = MULT(a, r)*4;
9     printf("a=%f\n", a);
10    return 0;
11 }
```



After pre-compilation

```
1 #include <stdio.h>
2 int main()
3 {
4     float a = 2, r = 4.5;
5     a = a*r+r*4;
6     printf("a=%f\n", a);
7     return 0;
8 }
```

- It is better to put the bracket on the whole

Instruction for Macros: `#define` (5)

- Pay attention that the constant has NO type
- We can similarly define Macro function

```
1 #ifndef MULT
2 #define MULT(x,y) (x*y+y)
3 #endif
4 #include <stdio.h>
5 int main()
6 {
7     float a = 2, r = 4.5;
8     a = MULT(a, r)*4;
9     printf("a=%f\n", a);
10    return 0;
11 }
```

⇒

After pre-compilation

```
1 #include <stdio.h>
2 int main()
3 {
4     float a = 2, r = 4.5;
5     a = (a*r+r)*4;
6     printf("a=%f\n", a);
7     return 0;
8 }
```

- It is better to put the bracket on the whole

Instruction for Macros: `#define` (6)

- It is literally replacement all the time

```
1 #ifndef HI
2 #define HI "hello"
3 #define WD world
4 #endif
5 #include <stdio.h>
6 int main()
7 {
8     printf("HI");
9     printf("\n");
10    printf(WD);
11    printf("\n");
12    printf(HI);
13    return 0;
14 }
```

[Output]

```
1 ??
2 ??
```


Instruction for Macros: `#define` (7)

- It is literally replacement all the time

```
1 #ifndef HI
2 #define HI "hello"
3 #define WD world
4 #endif
5 #include <stdio.h>
6 int main()
7 {
8     printf("HI");
9     printf("\n");
10    // printf(WD); //<--mistake
11    printf("\n");
12    printf(HI);
13    return 0;
14 }
```

[after comment out line
10,
output]

```
1 HI
2 hello
```

Instruction for Macros: `#ifdef` (1)

- We can use Macro to control the compilation

```
1 #define DEBUG
2 #include <stdio.h>
3 int main()
4 {
5     int i = 0, j = 1;
6     for(i = 0; i < 5; i++)
7     {
8         j = i*2+1;
9         #ifdef DEBUG
10             printf("j = %f\n", j);
11         #endif
12     }
13     return 0;
14 }
```

```
1 // #define DEBUG
2 #include <stdio.h>
3 int main()
4 {
5     int i = 0, j = 1;
6     for(i = 0; i < 5; i++)
7     {
8         j = i*2+1;
9         #ifdef DEBUG
10             printf("j = %f\n", j);
11         #endif
12     }
13     return 0;
14 }
```

- The code is compiled inside `#ifdef` only when “**DEBUG**” is defined

Instruction for Macros: `#ifdef` (2)

- Codes after pre-compilation

```
1 #define DEBUG
2 #include <stdio.h>
3 int main()
4 {
5     int i = 0, j = 1;
6     for(i = 0; i < 5; i++)
7     {
8         j = i*2+1;
9         printf("j=%f\n", j);
10    }
11    return 0;
12 }
```

```
1 // #define DEBUG
2 #include <stdio.h>
3 int main()
4 {
5     int i = 0, j = 1;
6     for(i = 0; i < 5; i++)
7     {
8         j = i*2+1;
9     }
10    return 0;
11 }
```

- The code is compiled inside `#ifdef` only when “**DEBUG**” is defined