

姓名：席崇援

学习任务：

1. typedef;
2. 位域;
3. 文件读写;
4. 预处理。

完成情况：

已完成1-4

算法题

你的飞碟在这儿 Your Ride Is Here

题目描述

众所周知，在每一个彗星后都有一只 UFO。这些 UFO 时常来收集地球上的忠诚支持者。不幸的是，他们的飞碟每次出行都只能带上一组支持者。因此，他们要用一种聪明的方案让这些小组提前知道谁会被彗星带走。他们为每个彗星起了一个名字，通过这些名字来决定这个小组是不是被带走的那个特定的小组（你认为谁给这些彗星取的名字呢？）。关于如何搭配的细节会在下面告诉你；你的任务是写一个程序，通过小组名和彗星名来决定这个小组是否能被那颗彗星后面的 UFO 带走。

小组名和彗星名都以下列方式转换成一个数字：最终的数字就是名字中所有字母的积，其中 A 是 1，Z 是 26。例如，USACO 小组就是 $21 * 19 * 1 * 3 * 15 = 17955$ 。如果小组的数字 mod 47 等于彗星的数字 mod 47，你就得告诉这个小组需要准备好被带走！（记住“a mod b”是 a 除以 b 的余数，例如 $34 \bmod 10$ 等于 4）

写出一个程序，读入彗星名和小组名并算出用上面的方案能否将两个名字搭配起来，如果能搭配，就输出 **GO**，否则输出 **STAY**。小组名和彗星名均是没有空格或标点的一串大写字母（不超过 6 个字母）。

输入格式

第1行：一个长度为 1 到 6 的大写字母串，表示彗星的名字。

第2行：一个长度为 1 到 6 的大写字母串，表示队伍的名字。

输出格式

样例 #1

样例输入 #1

```
COMETQ  
HVNGAT
```

样例输出 #1

```
GO
```

样例 #2

样例输入 #2

```
ABSTAR  
USACO
```

样例输出 #2

```
STAY
```

解答

```
#include<stdio.h>

int main() {
    char a[26] = {0}; // 初始化为0，稍后将被覆盖
    char b1[12] = {0}; // 第一次输入的字符串
    char b2[12] = {0}; // 第二次输入的字符串
    int c1 = 1; // 第一次输入的字符串的计算结果
    int c2 = 1; // 第二次输入的字符串的计算结果

    // 读取两次输入，分别存储在数组b1和b2中
    scanf("%s", b1);
    scanf("%s", b2);

    // 初始化数组a为大写字母A到Z
    for (int i = 0; i < 26; i++) {
        a[i] = 'A' + i; // 'A' 的 ASCII 值是 65，所以 'A' + i 会得到 A 到 Z 的每个字母
    }

    // 计算b1和b2的值
    for (int k = 0; k < 2; k++) {
        char b[12] = {0}; // 当前处理的字符串
        int c = 1; // 当前处理的字符串的计算结果
        if (k == 0) {
            strcpy(b, b1); // 复制第一次输入的字符串
        } else {
            strcpy(b, b2); // 复制第二次输入的字符串
        }

        // 计算当前字符串的值
        for (int i = 0; i < 6 && i < strlen(b); i++) { // 假设只处理前6个字符
            for (int j = 0; j < 26; j++) {
                if (b[i] == a[j]) {
                    c *= (j + 1);
                    break; // 找到匹配后退出内层循环
                }
            }
        }
        if (k == 0) {
            c1 = c; // 保存第一次输入的计算结果
        } else {
            c2 = c; // 保存第二次输入的计算结果
        }
    }
}
```

```
// 比较c1和c2对47取模的结果
if (c1 % 47 == c2 % 47) {
    printf("GO");
} else {
    printf("STAY");
}

return 0;
}
```

笔记:

typedef

1.什么是typedef

typedef 是C语言中的一个关键字，它的作用是给一个已经存在的类型起一个别名。

typedef 可以为基本数据类型、自定义数据类型（结构体、共用体、枚举类型）、数组和指针定义简洁的类型名称。一旦用户在程序中定义了自己的数据类型名称，就可以在该程序中用自己的数据类型名称来定义变量的类型、数组的类型、指针变量的类型与函数的类型等。

typedef 在编译时处理，有类型检查功能，不同于 #define 的简单文本替换。

typedef 可以增强程序的可读性和可移植性。

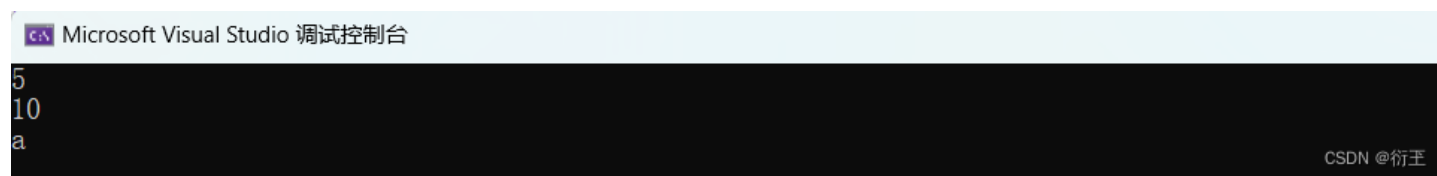
2.typedef 用法

1) .基本数据类型的重定义:

示例:

```
#include<stdio.h>
typedef int i;
typedef char ch;
typedef long long int li;
int main()
{
    i a = 5;
    li b = 10;
    ch c = 'a';
    printf("%d\n", a);
    printf("%lld\n", b);
    printf("%c\n", c);
    return 0;
}
```

运行结果如下图所示：



在这里，我们将 int、char、long long int 分别重定义为 i、ch、li，并通过新的类型名声明变量 a,b,c 由此我们可以知道，当使用 typedef 定义类型名后，我们可以使用新的类型名声明与原类型名声明的变量相同的变量。

注：使用 typedef 定义类型名后，原类型名作用不变。

2) .自定义数据类型重定义：

a.结构体：

示例：

```
#include<stdio.h>
struct student
{
    char name[20];
    int age;
    char sex[7];
};
typedef struct student stu;
int main()
{
    stu a = { "wang", 10, "male" };
    printf("%d", a.age);
    return 0;
}
```

运行结果如下图所示：



在这里我们对结构体 struct student 重定义为stu，并通过 stu 声明变量 a 并进行初始化。

我们也可以在结构体定义时进行重定义，如下：

```
#include<stdio.h>
typedef struct student
{
    char name[20];
    int age;
    char sex[7];
}stu;
int main()
{
    stu a = { "wang", 10, "male" };
    printf("%d", a.age);
    return 0;
}
```

此时，结构体变量 第一个名称可以省略，如下：

```
#include<stdio.h>
typedef struct
{
    char name[20];
    int age;
    char sex[7];
}stu;
int main()
{
    stu a = { "wang", 10, "male" };
    printf("%d", a.age);
    return 0;
}
```

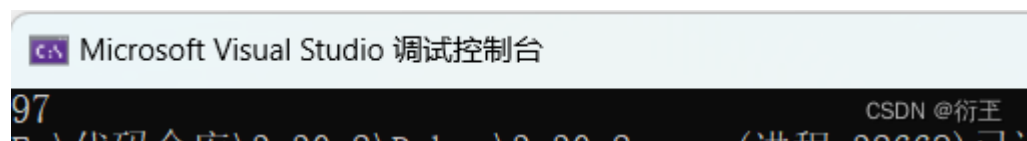
b.共用体：

共用体的重定义与结构体相似

示例：

```
#include<stdio.h>
union un
{
    char a;
    int b;
};
typedef union un u;
int main()
{
    u b = {0};
    b.a = 'a';
    printf("%d", b.b);
    return 0;
}
```

运行结果如下图：



在这里我们对共用体 union un 重定义为 u，并通过 u 声明变量共用体变量 b 并进行初始化。

我们也可以像结构体一样在定义共用体时进行重定义，如下：

```
#include<stdio.h>
typedef union un
{
    char a;
    int b;
}u;
int main()
{
    u b = { 0 };
    b.a = 'a';
    printf("%d", b.b);
    return 0;
}
```

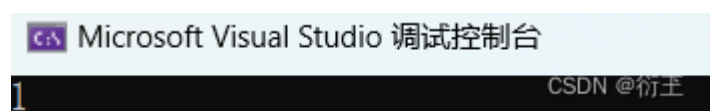
c.枚举变量：

示例：

```
#include<stdio.h>
enum week
{
    one = 1,
    two,
    three,
    four,
    five,
    six,
    seven
};
typedef enum week w;
int main()
{
    w a = one;
    printf("%d", a);
    return 0;
}
```

在这里我们定义枚举类型 week 后对 week 进行重定义为 w，并通过 w 声明枚举变量 a 并进行初始化。

运行结果如下图：



同时，也可以像共用体和结构体一样，在定义枚举类型时进行重定义，如下：

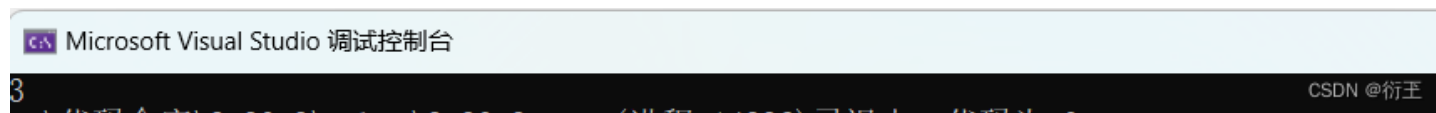
```
#include<stdio.h>
typedef enum week
{
    one = 1,
    two,
    three,
    four,
    five,
    six,
    seven
}w;
int main()
{
    w a = one;
    printf("%d", a);
    return 0;
}
```

3) .数组的重定义：

示例：

```
#include<stdio.h>
typedef int (i)[5];
int main()
{
    i a = { 1,2,3,4,5 };
    printf("%d", *(a+2));
    return 0;
}
```

结果如下图所示：



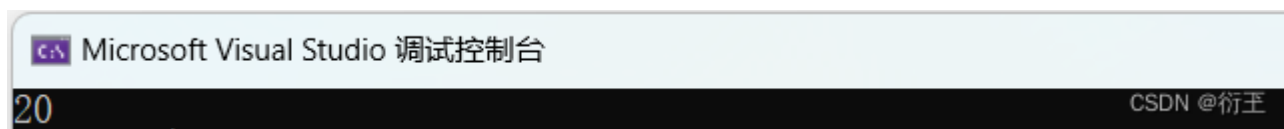
在这里我们将包含五个整型的数组重定义为i，并通过 i 声明枚举变量 a 并进行初始化。

4) .函数的重定义：

示例：

```
#include<stdio.h>
int fun(void)
{
    return 20;
}
typedef int(f)(void) ;
int main()
{
    f* a = fun;
    printf("%d", (a()));
    return 0;
}
```

结果如下图所示：



这里我们定义一个返回整型，并且无参数的函数类型为 f，通过 f 声明一个函数变量并赋值为 fun。

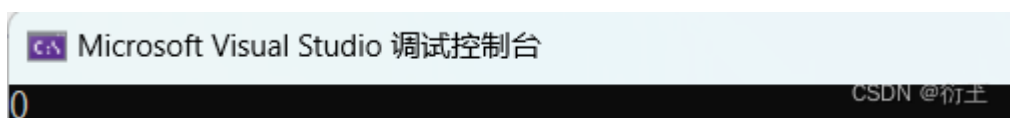
5) .指针的重定义：

a.指针：

示例：

```
#include<stdio.h>
typedef int* i;
int main()
{
    int a = 0;
    i b = &a;
    printf("%d", *b);
    return 0;
}
```

运行结果如下图：



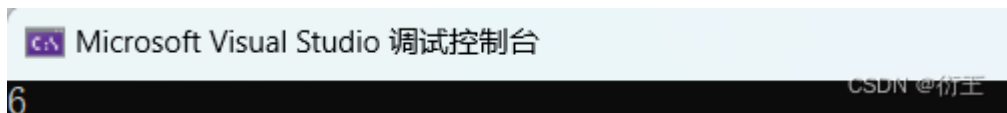
这里我们将 int* 类型重定义为 i，并通过 i 声明整型指针变量 b，并将整型变量 a 的地址赋给 b。

b.数组指针重定义：

示例：

```
#include<stdio.h>
typedef int(*i)[5];
int main()
{
    int a[5] = { 1,2,5,6,9 };
    i b = &a;
    printf("%d", *((*b)+3));
    return 0;
}
```

运行结果如下图：



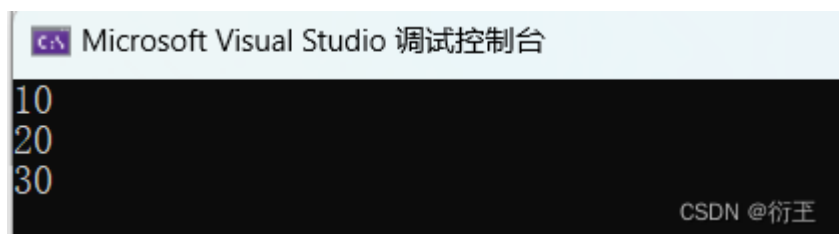
这里我们将指向包含五个整型的数组的指针重定义为 i，通过 i 声明 数组指针变量 b，并赋值为数组 a 的地址。

c.指针数组的重定义：

指针数组，顾名思义就是包含多个指针的数组。

```
#include<stdio.h>
typedef int* i[5];
int main()
{
    int a = 10, b = 20, c = 30;
    i arr = { &a,&b,&c };
    printf("%d\n", **arr);
    printf("%d\n", **(arr+1));
    printf("%d\n", **(arr+2));
    return 0;
}
```

运行结果如下图所示：



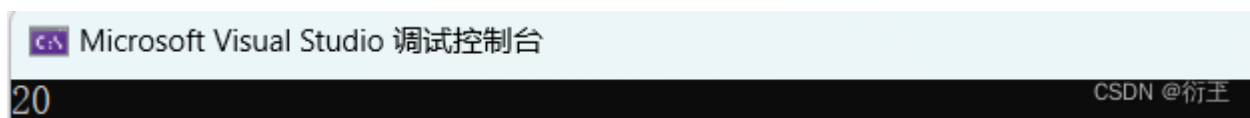
这里我们对指针数组重定义为 `i`，通过 `i` 声明变量 `arr`，并初始化为 `&a, &b, &c`。

d.函数指针：

示例：

```
#include<stdio.h>
int fun(void)
{
    return 20;
}
typedef int( *f)(void) ;
int main()
{
    f a = fun;
    printf("%d", (a()));
    return 0;
}
```

返回值如下图：



这里将返回值为 `int`（整型），并且没有参数的函数类型重定义为 `(*f)`，（即 `f` 为指向函数的指针类型），并将函数 `fun` 赋给函数变量 `a`。

3.注意事项：

1) .小心typedef的陷阱：

我们先看一个简单的代码：

```
#include<stdio.h>
typedef int* i;
int main()
{
    int a = 10;
    const i b = &a;
    return 0;
}
```

对于这个代码，我们会下意识的认为 `const` 的用法“左定值，有定向”，所以这个代码相当于“`const int * b;`”，**实则不然，typedef 不同于宏，不是简单的字符替换**，这个代码相当于“`int * const b;`”。

2).对于多个储存类关键字:

在语法上, typedef 不没有储存类功能, 但在语法上属于储存类关键字, 因此, 下图这种情况是错误的:

```
typedef static int a;
```

在这里, typedef 占据了储存类关键字的位置, 不再允许其他储存类关键字的出现。

位域

位域的概念

有些数据在存储时并不需要占用一个完整的字节, 只需要占用一个或几个二进制位即可。例如开关只有通电和断电两种状态, 用0和1表示足以, 也就是用一个二进位。正是基于这种考虑, C语言又提供了一种数据结构, 叫做“**位域**”或“**位段**”。

位域是操控位的一种方法 (操控位的另一种方法是使用按**位运算符**, 按位运算符将在之后的笔记中做介绍) 。

位域通过一个结构声明来建立: 该结构声明为每个字段提供标签, 并确定该字段的宽度。例如, 下面的声明建立了个4个1位的字段:

```
struct
{
    unsigned int autfd:1;
    unsigned int bldfc:1;
    unsigned int undin:1;
    unsigned int itals:1;
}prnt;
```

根据该声明, prnt包含4个1位的字段。现在, 可以通过普通的结构成员运算符(.)单独给这些字段赋值:

```
prnt.itals = 0;
prnt.undin = 1;
```

由于每个字段恰好为1位, 所以只能为其赋值1或0。变量prnt被储存在int大小的内存单元中, 但是在本例中只使用了其中的4位。

:后面的数字用来限定成员变量占用的位数。位域的宽度不能超过它所依附的数据类型的长度。通俗地讲，成员变量都是有类型的，这个类型限制了成员变量的最大长度，:后面的数字不能超过这个长度。

如上述结构中autfd、bldfc、undin、itals后面的数字不能超过unsigned int的位数，即在32bit环境中就是不能超过32。

位域的取值范围非常有限，数据稍微大些就会发生溢出，请看下面的例子：

```
#include <stdio.h>

struct pack
{
    unsigned a:2; // 取值范围为：0~3
    unsigned b:4; // 取值范围为：0~15
    unsigned c:6; // 取值范围为：0~63
};

int main(void)
{
    struct pack pk1;
    struct pack pk2;

    // 给pk1各成员赋值并打印输出
    pk1.a = 1;
    pk1.b = 10;
    pk1.c = 50;
    printf("%d, %d, %d\n", pk1.a, pk1.b, pk1.c);

    // 给pk2各成员赋值并打印输出
    pk2.a = 5;
    pk2.b = 20;
    pk2.c = 66;
    printf("%d, %d, %d\n", pk2.a, pk2.b, pk2.c);

    return 0;
}
```

程序输出结果为：

```
pk1.a = 1, pk1.b = 10, pk1.c = 5
pk2.a = 1, pk2.b = 4, pk2.c = 2
```

显然，结构体变量pk1的各成员都没有超出限定的位数，能够正常输出。而结构体变量pk2的各成员超出了限定的位数，并发生了上溢（溢出中的一种），关于溢出的概念可查看往期笔记：[【C语言笔记】](#)

整数溢出

C语言标准规定，只有有限的几种数据类型可以用于位域。在ANSI C 中，这几种数据类型是signed int 和unsigned int；到了C99、C11新增了_Bool 的位字段。关于C语言的几套标准可查看往期笔记：[【C语言笔记】什么是ANSI C标准？](#)

位域的存储

位域的存储同样遵循结构体内存对齐的规则，关于结构体内存对齐的问题可查看往期笔记：[【C语言笔记】结构体内存对齐问题](#)

看一个例子：

```
#include <stdio.h>
struct pack
{
    unsigned a:2;
    unsigned b:4;
    unsigned c:6;
};
int main(void)
{
    printf("sizeof(struct pack) = %d", sizeof(struct pack));
    return 0;
}
123456789101112
```

程序输出结果为：

```
sizeof(struct pack) = 4
```

这是因为，a、b、c成员所占的位长之和在一个存储单元（此处为unsigned类型所占的字节数）内，即4个字节内，所以struct pack类型的变量所占的字节长度为4个字节（实际a、b、c一共占用12bit,还有20bit空间为保留的空白）。

可能有人有疑问，此处a、b、c加起来一共才12bit，两个字节都不到，那么只需要，2个字节不就好了吗。这就是因为内存对齐搞的鬼，此处要将内存对齐到4个字节（unsigned类型所占的字节数），以便提高存取效率。

假如把该结构声明改为：

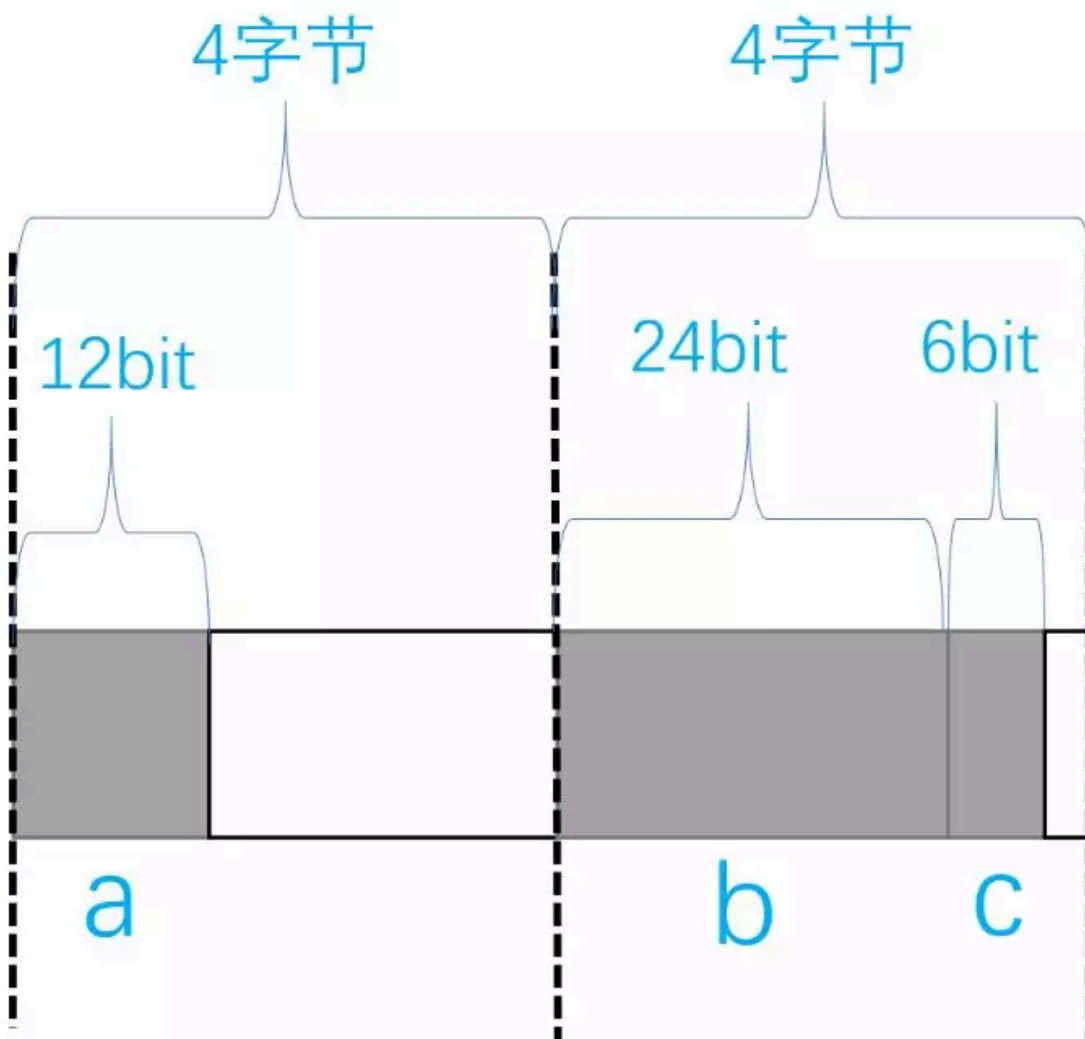
```
struct pack
{
    unsigned a:12;
    unsigned b:24;
    unsigned c:6;
};
```

那么，输出结构应该为什么呢？

输出结果为：

```
sizeof(struct pack) = 8
```

因为此时a成员单独占一个内存单元（4字节），b、c成员紧挨着占下一个内存单元（4字节），所以结果为8字节，这也是因为内存对齐。a、b、c占用内存的示意图如：



其中，空白部分为保留的空白填充内存。这里的空白内存是系统自动留出的，同时，我们也可以自己留出填充内存。如**无名位域**就可以用来作填充：


```
struct pack
{
    unsigned a:12;
    unsigned :20; //该位域成员不能使用，用于填充
    unsigned c:6;
};
```

无名位域一般用来作填充或者调整成员位置。因为没有名称，无名位域不能使用。

上面的例子中，如果没有位宽为 20 的无名成员，a、c 将会挨着存储，sizeof(struct pack) 的结果为 4；有了这 20 位作为填充，a、c 将分开存储，sizeof(struct pack) 的结果为 8。

文件读写

一、文件的打开和关闭

1. fopen()

function

fopen

<stdio>

FILE * fopen (const char * filename, const char * mode);

CSDN @XYLoveBarbecue

第二个参数为文件的打开方式：mode:传入文件的打开方式（打开方式有： "r","w","a","rb","wb","ab","r+","w+","a+","rb+","wb+","ab+"）：

文件使用方式	含义	如果指定文件不存在
"r"(只读)	为了输入数据，打开一个已经存在的文本文件	出错
"w"(只写)	为了输出数据，打开一个文本文件	建立一个新的文件
"a"(追加)	向文本文件尾添加数据	建立一个新的文件
"rb"(只读)	为了输入数据，打开一个二进制文件	出错
"wb"(只写)	为了输出数据，打开一个二进制文件	建立一个新的文件
"ab"(追加)	向一个二进制文件尾添加数据	出错
"r+"(读写)	为了读和写，打开一个文本文件	出错
"w+"(读写)	为了读和写，建一个新的文件	建立一个新的文件
"a+"(读写)	打开一个文件，在文件尾进行读写	建立一个新的文件
"rb+"(读写)	为了读和写打开一个二进制文件	出错
"wb+"(读写)	为了读和写，新建一个新的二进制文件	建立一个新的文件
"ab+"(读写)	打开一个二进制文件，在文件尾进行读和写	建立一个新的文件

注：当使用“w”，“wb”，“w+”，“wb+”打开文件时会清除文件原本存储的数据。

该函数的返回值是一个FILE类型的指针，在文件成功打开后返回此文件信息区的起始地址，打开失败则返回一个NULL（空指针），所以使用fopen时需要一个FILE类型的指针来接收其返回值。

```
//打开
FILE* pf = fopen("text.txt","w");
//判断是否打开成功
if (pf == NULL)
{
    perror("fopen");
    return 1;
}
```

2.fclose()

fclose

```
int fclose ( FILE * stream );
```

stream:传入将要关闭的文件的文件指针。

该函数的返回值是int类型的：如果关闭成功就返回0值，否则返回EOF（-1）值。

```
int main()
{
    //打开
    FILE* pf = fopen("text.txt","w");
    //判断是否打开成功
    if (pf == NULL)
    {
        perror("fopen");
        return 1;
    }
    //关闭
    if (fclose(pf) == EOF)
    {
        //关闭失败
        perror("fclose");
        return 1;
    }
    pf = NULL;
    return 0;
}
```

在关闭文件成功之后我们应该将原本的文件指针置为空（NULL），避免野指针的产生。

二、文件的读写

1.顺序读写

功能	函数名	适用于
字符输入函数	fgetc	所有输入流
字符输出函数	fputc	所有输出流
文本行输入函数	fgets	所有输入流
文本行输出函数	fputs	所有输出流
格式化输入函数	fscanf	所有输入流
格式化输出函数	fprintf	所有输出流
二进制输入	fread	文件
二进制输出	fwrite	文件

CSDN @XYLoveBarbecue

2.1.1 fputc()

该函数的作用是用来向文件输入单个字符数据的

fputc

```
int fputc ( int character, FILE * stream );
```

CSDN @XYLoveBarbecue

该函数有int类型的character和FILE*类型的stream两个参数:

character:接收所要存入的字符的ASCII值

stream:接收指向所要存入文件的指针

```
int main()
{
    //打开
    FILE* pf = fopen("text.txt", "w");
    //判断是否打开成功
    if (pf == NULL)
    {
        perror("fopen");
        return 1;
    }
    //写入数据
    int i;
    for (i = 0; i < 26; i++)
    {
        fputc('a' + i, pf);
    }
    //关闭
    if (fclose(pf) == EOF)
    {
        //关闭失败
        perror("fclose");
        return 1;
    }
    pf = NULL;
    return 0;
}
```

2.1.2 fgetc()

该函数一般被用来读取文件中单个字符数据

fgetc

```
int fgetc ( FILE * stream );
```

CSDN @XYLoveBarbecue

该函数有一个FILE*类型的参数stream:

stream:接收指向所要读取文件的指针

该函数成功读入数据时会返回读取字符的ASCII值，反之则会返回EOF（-1）值。

```
int main()
{
    //打开
    FILE* pf = fopen("text.txt", "r");//此时读取文件要用"r"的方式打开
    //判断是否打开成功
    if (pf == NULL)
    {
        perror("fopen");
        return 1;
    }
    //读入数据
    int i;
    while ((i = fgetc(pf)) != EOF)
    {
        printf("%c ", i);
    }
    //关闭
    if (fclose(pf) == EOF)
    {
        //关闭失败
        perror("fclose");
        return 1;
    }
    pf = NULL;
    return 0;
}
```

2.1.3 fputs()

读写文件一个一个字符操作太麻烦了，我们可以使用fputs函数直接向文件写入一个字符串

fputs

```
int fputs ( const char * str, FILE * stream );
```

CSDN @XYLoveBarbecue

str:接收指向将要存入字符串的指针

stream:接收指向所要存入文件的指针

该函数成功运行返回一个非负值，否则返回EOF（-1）值。

```
int main()
{
    //打开
    FILE* pf = fopen("text.txt", "w");
    //判断是否打开成功
    if (pf == NULL)
    {
        perror("fopen");
        return 1;
    }
    //存入数据
    fputs("hello", pf);
    fputs("world", pf);
    //关闭
    if (fclose(pf) == EOF)
    {
        //关闭失败
        perror("fclose");
        return 1;
    }
    pf = NULL;
    return 0;
}
```

2.1.4fgets()

对于输出文件文本行（字符串）数据一般使用fgets函数来进行操作

fgets

```
char * fgets ( char * str, int num, FILE * stream );
```

CSDN @XYLoveBarbecue

str:接收指向储存读出数据的字符串指针

num:接收要复制到 str 中的最大字符数（包括终止空字符）

stream:接收指向所要读取文件的指针

成功后，该函数返回 str的头指针

如果在尝试读取字符时遇到文件结尾，则会设置 EOF 指示符。如果在读取任何字符之前发生这种情况，则返回的指针为空指针（并且 str 的内容保持不变）。

如果发生读取错误，则设置错误指示器（ferror）并返回空指针（但 str 所指向的内容可能已更改）。

```
#include <stdio.h>

int main()
{
    FILE *fp;
    char str[60];

    /* 打开用于读取的文件 */
    fp = fopen("file.txt" , "r");
    if(fp == NULL) {
        perror("打开文件时发生错误");
        return(-1);
    }
    if( fgets (str, 60, fp)!=NULL ) {
        /* 向标准输出 stdout 写入内容 */
        puts(str);
    }
    fclose(fp);

    return(0);
}
```

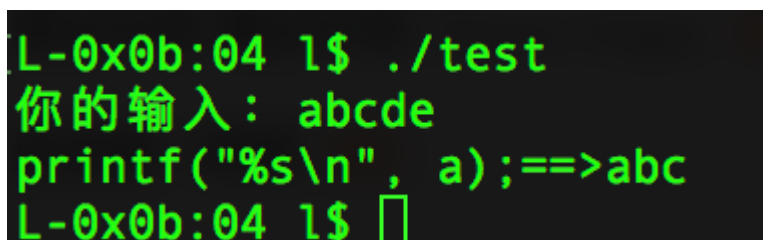
fgets()函数的注意事项1

fgets() 函数的最大读取大小是其“第二个参数减1”，这是由于字符串是以 ‘\0’ 为结束符的，fgets() 为了保证输入内容的字符串格式，当输入的数据大小超过了第二个参数指定的大小的时候，fgets() 会仅仅读取前面的“第二个参数减1”个字符，而预留 1个字符的空间来存储字符串结束符 ‘\0’。

测试代码：

```
#include <stdio.h>
int main(void)
{
    char a[10] = {0};
    printf("你的输入: ");
    fgets(a, 4, stdin);
    //printf("%s\n", a); //下面这句的输出和这句是一样的
    printf("printf(\"%s\\n\", a)%c==>%s\\n", ';', a);
    return 0;
}
```

运行效果：



```
L-0x0b:04 1$ ./test
你的输入: abcde
printf("%s\\n", a);==>abc
L-0x0b:04 1$
```

在这个例子中，`fgets()` 的第二个参数是 4，所以它最多只能存储输入的 $(4-1 = 3)$ 个字符进第一个参数指向的地址空间里面。输入“abcde”，数组 `a[]` 中只有“abc”。

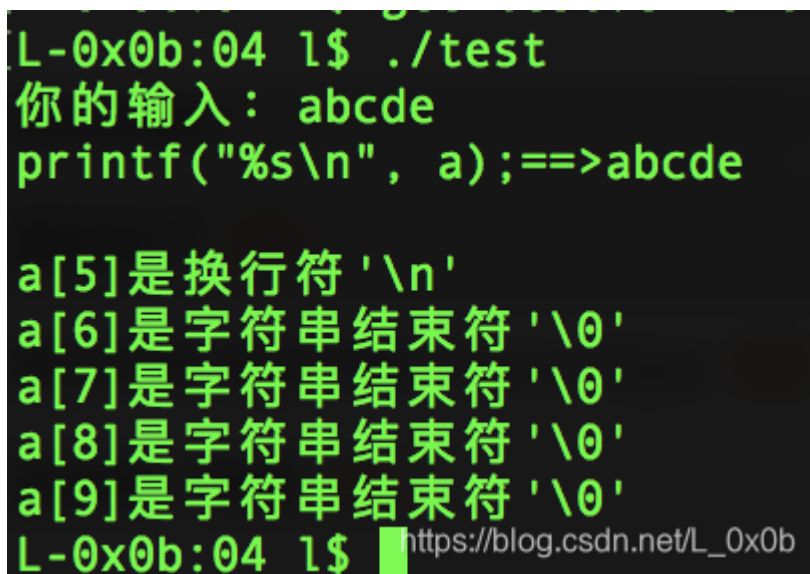
fgets()函数的注意事项2

在 `fgets()` 函数的眼里，换行符 `'\n'` 也是它要读取的一个普通字符而已。在读取键盘输入的时候会把最后输入的回车符也存进数组里面，即会把 `'\n'` 也存进数组里面，而又由于字符串本身会是以 `'\0'` 结尾的。所以在输入字符个数没有超过第二个参数指定大小之前，你输入 `n` 个字符按下回车输入，`fgets()` 存储进第一个参数指定内存地址的是 `n+2` 个字节。最后面会多出一个 `'\n'` 和一个 `'\0'`，而且 `'\n'` 是在 `'\0'` 的前面一个 (`\n\0`)。

测试代码：

```
#include <stdio.h>
int main(void)
{
    char a[10] = {0};
    printf("你的输入: ");
    fgets(a, 10, stdin);
    //printf("%s\n", a); //下面这句的输出和这句是一样的
    printf("printf(\"%%s\\n\", a)%c==>%s\n", ';', a);
    for(int i=0; i<10; i++)
    {
        if(a[i] == '\n')
            printf("a[%d]是换行符'\\n'\\n", i);
        if(a[i] == '\0')
            printf("a[%d]是字符串结束符'\\0'\\n", i);
    }
    return 0;
}
```

运行效果：



```
L-0x0b:04 l$ ./test
你的输入: abcde
printf("%s\\n", a);==>abcde

a[5]是换行符 '\\n'
a[6]是字符串结束符 '\\0'
a[7]是字符串结束符 '\\0'
a[8]是字符串结束符 '\\0'
a[9]是字符串结束符 '\\0'
L-0x0b:04 l$ https://blog.csdn.net/L_0x0b
```

在这个例子中，由于输入的字符小于 参数2 指定的 最大读取字符数，所以 `fgets()` 函数会把换行符 `'\n'` 也储存进数组 `a[]` 里面，在运行界面的第三行哪里 换了两次行，就是由于这个多出来的换行符 `'\n'` 和我输出代码中的换行符 `'\n'` 共同作用的结果。

fgets()函数的注意事项3


`fgets()` 函数只负责 读取，并不会事先清空 参数1 指向的地址内存。读取到的字节会覆盖原地址储存，但没有覆盖到的内容还是保持原样。

测试代码：

```
#include <stdio.h>
int main(void)
{
    char a[10] = {'1','1','1','1','1','1','1','1','1','1'};
    printf("你的输入: ");
    fgets(a, 10, stdin);
    //printf("%s\n", a); //下面这句的输出和这句是一样的
    printf("printf(\"%%s\\n\", a)%c==>%s\n", ';', a);
    for(int i=0; i<10; i++)
    {
        if(a[i] == '\n' || a[i] == '\0')
            printf("a[%d] = '\\%c'", i, a[i]=='\n'? 'n': '0');
        else
            printf("a[%d] = %c", i, a[i]);
        printf("\n");
    }
    return 0;
}
```

运行结果：

```
[L-0x0b:04 1$ ./test
你的输入： abcd
printf("%s\n", a);==>abcd

a[0] = a
a[1] = b
a[2] = c
a[3] = d
a[4] = '\n'
a[5] = '\0'
a[6] = 1
a[7] = 1
a[8] = 1
a[9] = 1
L-0x0b:04 1$  https://blog.csdn.net/L\_0x0b
```

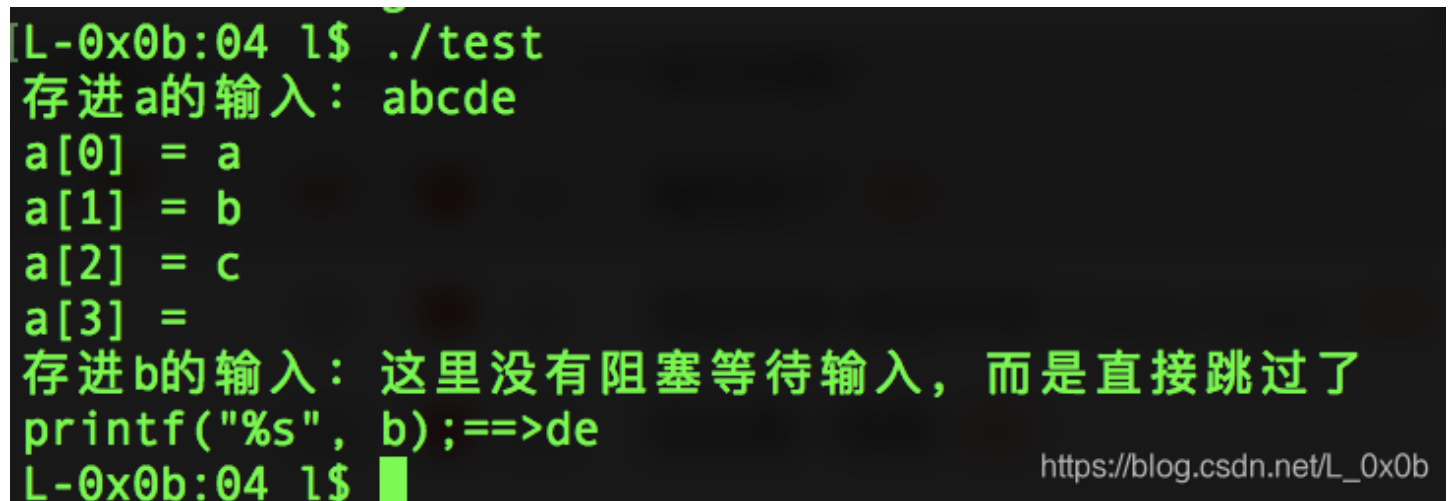
fgets()函数的注意事项4

在用 `fgets()` 函数读取键盘输入的时候，如果输入多于其“第二个参数减1”个字符大小的数据，`fgets()` 只会读取走前“第二个参数减1”个字符，多余的字符残留在输入缓存区里面。如果不清空，可能会影响下次输入。

测试代码：

```
#include <stdio.h>
int main(void)
{
    char a[4] = {0};
    char b[10] = {0};
    printf("存进a的输入: ");
    fgets(a, 4, stdin);
    for(int i=0; i<4; i++)
        printf("a[%d] = %c\n", i, a[i]);
    printf("存进b的输入: ");
    fgets(b, 10, stdin);
    printf("这里没有阻塞等待输入，而是直接跳过了\n");
    //printf("%s", a); //下面这句的输出和这句是一样的
    printf("printf(\"%%s\", b)%c==>%s", ';', b);
    return 0;
}
```

运行结果：



```
L-0x0b:04 l$ ./test
存进 a 的输入: abcde
a[0] = a
a[1] = b
a[2] = c
a[3] =
存进 b 的输入: 这里没有阻塞等待输入，而是直接跳过了
printf("%s", b);==>de
L-0x0b:04 l$
```

https://blog.csdn.net/L_0x0b

在这个例子中，输入“ abcde ”之后，数组 a[] 读取走“ abc ”之后，代码运行到第11行的时候并没有停下来等待用户的输入，而是直接读取了还留在缓存区里面的“ de\n ”，读取到 ‘\n’ 之后返回，所以我最后一行的输出代码中并没有加上换行符 ‘\n’，因为数组 b[] 中已经包含有换行符 ‘\n’ 了。

2.1.5 fprintf()

可以向文件输出任意数据

fprintf

```
int fprintf ( FILE * stream, const char * format, ... );
```

CSDN @XYLoveBarbecue

printf

```
int printf ( const char * format, ... );
```

CSDN @XYLoveBarbecue

仔细观察发现fprintf比printf函数唯一多出来的一个参数就是FILE*类型的stream。

如此一来就很好解释fprintf函数的用法了：

stream:接收指向所要输出文件的指针

format:接收将要输出数据的格式（和printf函数一样有%d, %x, %c, %s等等格式类型）

至于后面的...是接收所要输出的数据（如printf函数在格式后输入所要打印的变量一样）

```
struct S
{
    int age;
    char name[20];
    char sex[5];
};

int main()
{
    struct S L = { 20, "张三", "男" };
    //打开
    FILE* pf = fopen("text.txt", "w");
    //判断是否打开成功
    if (pf == NULL)
    {
        perror("fopen");
        return 1;
    }
    //输入数据
    fprintf(pf, "%d %s %s", L.age, L.name, L.sex);
    //关闭
    if (fclose(pf) == EOF)
    {
        //关闭失败
        perror("fclose");
        return 1;
    }
    pf = NULL;
    return 0;
}
```

2.1.6fscanf()

可以读取各种类型的数据的函数

fscanf

```
int fscanf ( FILE * stream, const char * format, ... );
```

CSDN @XYLoveBarbecue

scanf

```
int scanf ( const char * format, ... );
```

CSDN @XYLoveBarbecue

仔细观察发现fscanf比scanf函数唯一多出来的一个参数也是FILE*类型的stream。

那和scanf函数的用法也大同小异了：

stream:接收指向所要输入文件的指针

format:接收将要输入数据的格式（和scanf函数一样有%d, %x, %c, %s等等格式类型）

至于后面的...是接收所要输入的数据的变量（如scanf函数在格式后输入所要改变的变量一样）

运行成功后，该函数返回所读取数据的个数（甚至为零），如果在读取时发生读取错误或到达文件末尾，则会设置正确的指示器（feof 或 ferror）。而且，如果在成功读取任何数据之前发生任何一种情况，则返回 EOF。

注：使用fscanf和scanf函数时一样非字符串类型的所要储存输入数据的变量需要&

```
struct S
{
    int age;
    char name[20];
    char sex[5];
};

int main()
{
    struct S s;
    //打开
    FILE* pf = fopen("text.txt", "r");
    //判断是否打开成功
    if (pf == NULL)
    {
        perror("fopen");
        return 1;
    }
    //输出数据
    fscanf(pf, "%d %s %s", &(s.age), s.name, s.sex);
    printf("%d %s %s", s.age, s.name, s.sex);
    //关闭
    if (fclose(pf) == EOF)
    {
        //关闭失败
        perror("fclose");
        return 1;
    }
    pf = NULL;
    return 0;
}
```

2.1.7fwrite()

我们在向文件输出数据时都是以文本格式输出的，但是fwrite函数可以直接将计算机内存中所存储的二进制数据输出到文件中（此时此文件就是一个二进制文件）

fwrite

`size_t fwrite (const void * ptr, size_t size, size_t count, FILE * stream);` CSDN @XYLoveBarbecue

fwrite函数可以直接将计算机内存中所存储的二进制数据输出到文件中（此时此文件就是一个二进制文件）

ptr:接收指向所要输入数据的指针或地址

size:接收输出的每个数据的大小（以字节为单位）。

count:所要输出数据的个数

stream:接收指向所要输出文件的指针

成功运行返回输出的数据总数，如果此数字与 count 参数不同，则写入错误会阻止函数完成。在这种情况下，将为流设置误差指示器（ferror），如果大小或计数为零，则该函数返回零，并且错误指示器保持不变。

```
struct S
{
    int age;
    char name[20];
    char sex[5];
};

int main()
{
    struct S L = { 20, "李四", "女" };
    //打开
    FILE* pf = fopen("text.txt", "wb");//二进制输出时用wb
    //判断是否打开成功
    if (pf == NULL)
    {
        perror("fopen");
        return 1;
    }
    //输出数据
    fwrite(&L, sizeof(struct S), 1, pf);
    //关闭
    if (fclose(pf) == EOF)
    {
        //关闭失败
        perror("fclose");
        return 1;
    }
    pf = NULL;
    return 0;
}
```

2.1.8fread()

对于二进制的文件数据我们就要使用二进制的方式读取

fread

```
size_t fread ( void * ptr, size_t size, size_t count, FILE * stream );
```

 CSDN @XYLoveBarbecue

ptr:传入指向将要存储数据的变量的指针或地址

size:传入每次从文件读取数据的大小（以字节为单位）

count:传入将要读取的次数

stream:接收指向所要输入文件的指针

该函数运行成功后返回读取数据的次数，否则返回0值。

```
struct S
{
    int age;
    char name[20];
    char sex[5];
};
int main()
{
    struct S s;
    //打开
    FILE* pf = fopen("text.txt", "rb");//二进制输出时用rb
    //判断是否打开成功
    if (pf == NULL)
    {
        perror("fopen");
        return 1;
    }
    //输入数据
    if (fread(&s, sizeof(struct S), 1, pf) != 0)
    {
        printf("%d %s %s", s.age, s.name, s.sex);
    }
    //关闭
    if (fclose(pf) == EOF)
    {
        //关闭失败
        perror("fclose");
        return 1;
    }
    pf = NULL;
    return 0;
}
```

2.1.9标准输入流，标准输出流，标准错误流

stdin——标准输入流——键盘

stdout——标准输出流——屏幕

stderr ——标准错误流——屏幕

系统自动打开三个FILE*类型的流：stdin（接收键盘数据），stdout（接收所要向屏幕输出的数据），stderr（接收所要向屏幕输出的数据）。

2.随机读写

随机读写并不是真的随机读写，而是通过当前的文件指针所指向的位置相对于文件的指针的起始位置的偏移量来读取任意想要读取的位置。

2.1文件指针的偏移量

第一次读：



第二次读：



第三次读：



CSDN @XYLoveBarbecue

2.2fseek()

该函数可以设置文件指针指向的位置及偏移量。

fseek

```
int fseek ( FILE * stream, long int offset, int origin );
```

stream:传入将要改变的文件指针（文件流）

offset:传入指针所要偏移的偏移量

origin:设置传入的stream指针的起始位置（可传入3种参数：SEEK_SET（设置指针指向文件开头）、SEEK_CUR（不改变指针位置）、SEEK_END（设置指针指向文件结束位置））

Constant	Reference position
SEEK_SET	Beginning of file
SEEK_CUR	Current position of the file pointer
SEEK_END	End of file *

该函数运行成功，返回零。否则回非零值。

如果发生读取或写入错误，则设置错误指示器（ferror）。

2.3ftell()

ftell函数可以返回文件指针相对于起始位置的偏移量：

ftell

```
long int ftell ( FILE * stream );
```

stream:传入所需计算偏移量的文件指针

成功后，返回位置指示器的当前值。失败时，返回 -1，并将 **errno** 设置为系统特定的正值。

2.4rewind()

rewind

```
void rewind ( FILE * stream );
```

CSDN @XYLoveBarbecue

该函数可以将所传入的文件指针设置指向文件初始位置：

stream:传入将要改变的文件指针

三、文本文件和二进制文件

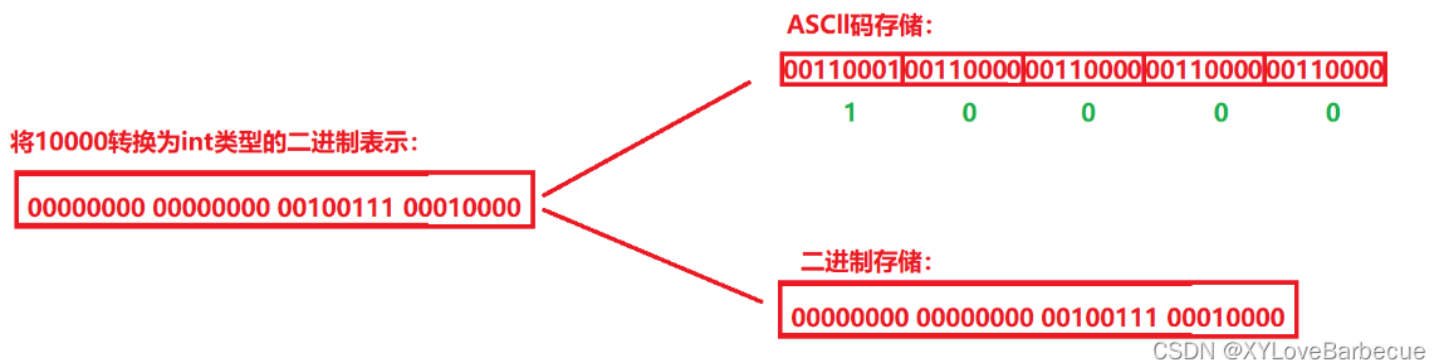
根据数据的组织形式，数据文件被分为文本文件或者二进制文件。

数据在内存中以二进制的形式存储，如果不加转换的输出到外存（磁盘文件等等），就是二进制文件。

如果要求在外存上以ASCII码的形式存储，则需要在存储前转换。以ASCII字符的形式存储的文件就是文本文件。

数据在内存中字符一律以ASCII形式存储，数值型数据既可以用ASCII形式存储，也可以使用二进制形式存储。

我们拿10000这个数据来举例，用ASCII码来存储需要5个字节（因为10000是五位数，每一位数都要用一个字节的ASCII码来表示），而用二进制存储只需要四个字节（一个整型大小为4字节）。



四、文件读取结束的判定

4.1feof()

在我们使用读取文件的函数时，都讲解了其返回值，我们可以通过其返回值来判定对文件的读取是否成功（这里不再一一举例）。

补充一个用来判断文件为什么结束读取的函数：

feof函数

对于文件读取结束之后，我们可以使用feof函数来判断文件是否读取结束的

feof

```
int feof ( FILE * stream ); CSDN @XYLoveBarbecue
```

该函数只有一个FILE*类型的参数：

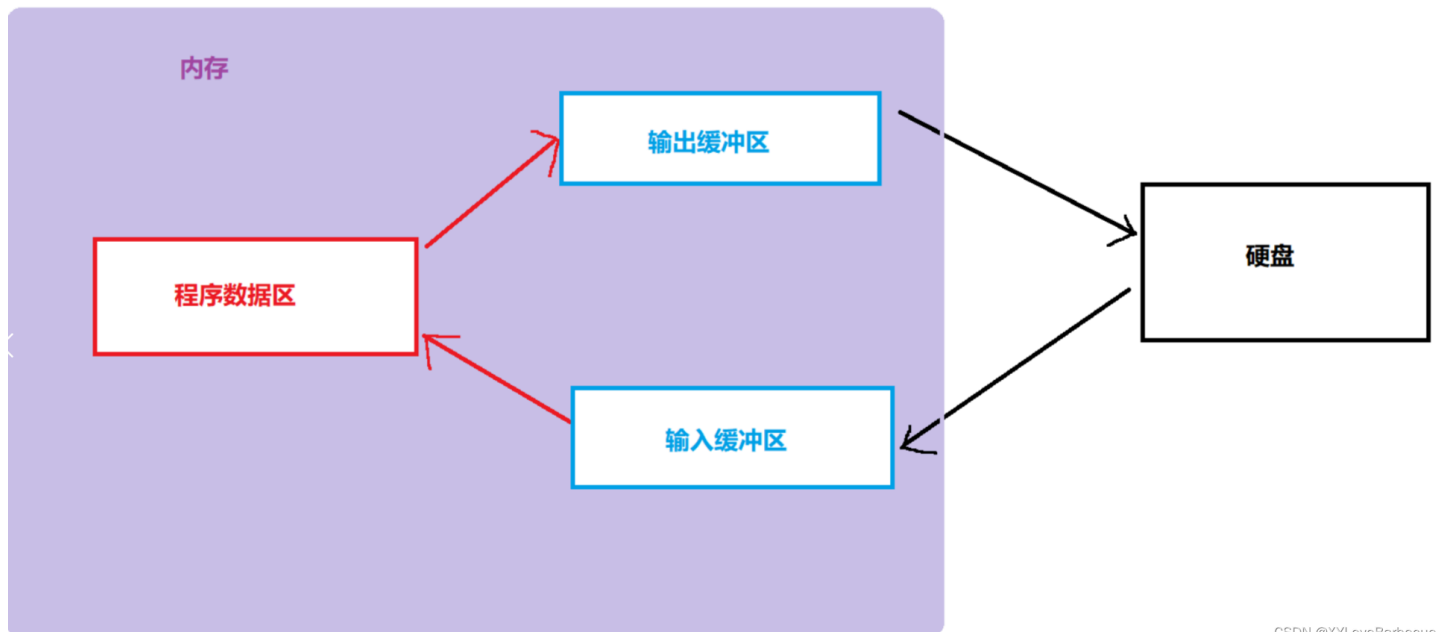
stream:传入所需要判断的文件指针

当传入的文件流是遇到文件末尾而结束读取时该函数返回非0值，其他原因返回0。

五、文件缓冲区

ANSI C标准采用"缓冲文件系统"处理的数据文件的，所谓缓冲文件系统是指系统自动地在内存中为程序中每一个正在使用的文件开辟一块"文件缓冲区"。从内存向磁盘输出数据会先送到内存中的缓冲区，装满缓冲区后才一起送到磁盘上。如果从磁盘向计算机读入数据，则从磁盘文件中读取数据输入到内存缓冲区（充满缓冲区），然后再从缓冲区逐个地将数据送到程序数据区（程序变量等）。缓冲区的大小根据C编译系统决定的。

以下是程序向硬盘输入数据和硬盘向程序输出数据的流程：



CSDN @XYLoveBarbecue

```
int main()
{
    //打开
    FILE* pf = fopen("text.txt", "wb");
    if (pf == NULL)
    {
        perror("fopen");
        return 1;
    }
    //存入
    int a = 10000;
    fwrite(&a, sizeof(int), 1, pf);
    printf("此20秒数据在文件缓冲区内，打开文件是没有数据的\n");
    Sleep(20000); //睡眠10秒，睡眠时打开文件可以观察
    fflush(pf); //此函数可以刷新缓冲区中的数据，使其存入硬盘文件中
    printf("此20秒数据从文件缓冲区内读入到文件中，打开文件是有数据的\n");
    Sleep(20000);
    //关闭
    fclose(pf); //fclose()也会清理缓冲区，因此前面也要加Sleep()
    pf = NULL;
    return 0;
}
```

预处理



一、预定义符号

C语言设置了一些预定义符号，可以直接使用，预定义符号也是在预处理期间处理的。

1. **FILE**：进行编译的源文件
2. **LINE**：文件当前的行号
3. **DATE**：文件被编译的日期
4. **TIME**：文件被编译的时间

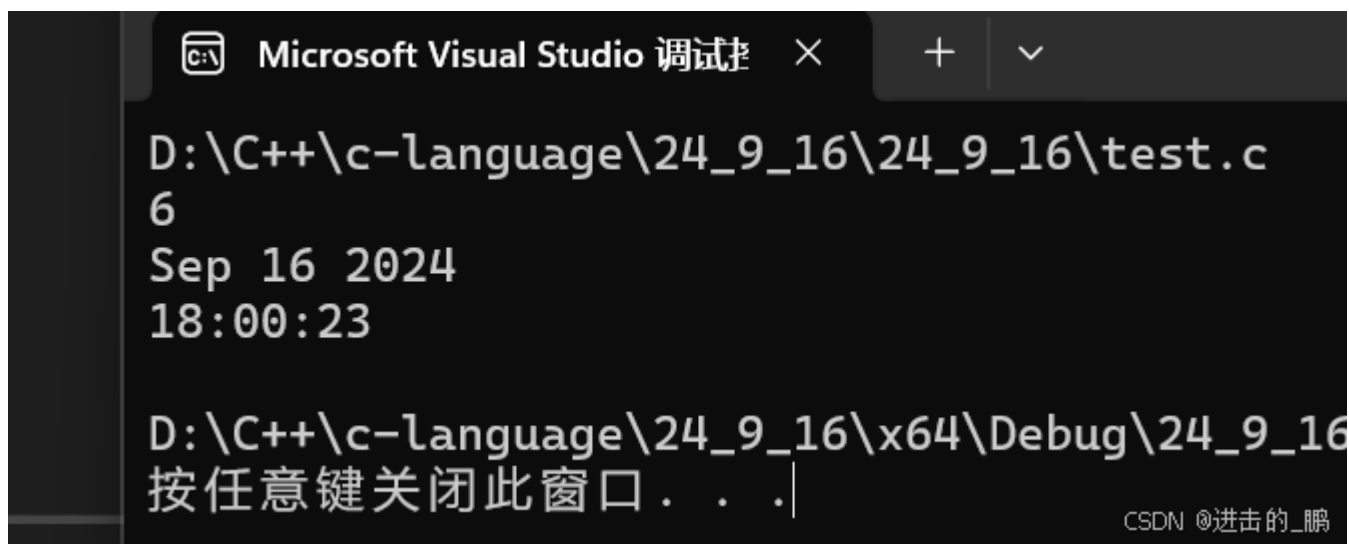
5. **STDC**: 如果编译器遵循 ANSI C , 其值为 1 , 否则未定义 (VS不支持该符号)

演示:

```
#include <stdio.h>

int main() {
    printf("%s\n", __FILE__); //进行编译的源文件
    printf("%d\n", __LINE__); //文件当前的行号
    printf("%s\n", __DATE__); //文件被编译的日期
    printf("%s\n", __TIME__); //文件被编译的时间
    return 0;
}
```

运行结果:

The image shows a screenshot of the Microsoft Visual Studio debug console. The title bar at the top reads "Microsoft Visual Studio 调试" with standard window controls. The console output is as follows:
D:\C++\c-language\24_9_16\24_9_16\test.c
6
Sep 16 2024
18:00:23

D:\C++\c-language\24_9_16\x64\Debug\24_9_16
按任意键关闭此窗口 . . .
In the bottom right corner of the console window, the text "CSDN @进击的_鹏" is visible.

二、#define 定义常量

基本语法:

```
#define name stuff
```

- name: 标识符名字, 记住必须是连续的, 不能有空格
- stuff: 可以是数字, 也可以是字符, 它会在预处理阶段替换掉 name

演示:

```
#include <stdio.h>

#define M 100    //定义一个常量

int main() {
    printf("%d\n", M);
    return 0;
}
```

运行结果:



****解析:****首先, 该过程发生在编译与链接中的编译的预处理阶段, 该过程会把定义的常量标识符直接替换成常量, 如上, 代码在预处理阶段就会把 M 替换成 100。

注意1: #define 定义后面不要加 分号, 因为这个分号也会在预处理阶段替换掉标识符, 这样就会多一个分号, 导致编译报错

****注意2:****如果我们使用 #define 定义一个较长的符号时, 我们需要换行时不能直接回车, 而是需要用到续行符: ***

如:

```
#include <stdio.h>

#define DEBUG_PRINT printf("file:%s\tline:%d\tdate:%s\ttime:%s\n" , __FILE__ , __LINE__ , __DATE__ , __TIME__ )
```

我们应该使用下面这种方式换行:

```
#define DEBUG_PRINT printf("file:%s\tline:%d\tdate:%s\ttime:%s\n" , \
                           __FILE__ , \
                           __LINE__ , \
                           __DATE__ , \
                           __TIME__ )
```

****切记:****续行符 \ 后面要直接回车, 不能有空格之类的东西。

三、#define定义宏

#define 机制包括了一个规定，允许把参数替换到文本中，这种实现通常称为宏（macro）或定义宏（definemacro）。

宏的声明：

```
#define name( parament-list ) stuff
```

- name：宏的名字
- parament-list：参数表，传参的参数名
- stuff：宏实现的内容

****注意：参数列表的左括号必须与name紧邻，如果两者之间有任何空白存在，参数列表就会被解释为stuff的一部分。**

****举例：使用宏实现一个平方的计算**

```
#include <stdio.h>

//定义一个宏
#define SQUARE(x) ((x)*(x))//括号是为了保证优先级

int main()
{
    int m = 0;
    scanf("%d", &m);
    int ret = SQUARE(m);
    printf("%d\n", ret);

    return 0;
}
```

运行结果：



****解析：****简单来说，宏就是把参数替换到 stuff 里，在预处理阶段，上述 `int ret = SQUARE(m);` 会被替换为 `int ret = ((m)*(m));`，然后在程序运行时被计算。

****提问：****为什么要使用那么多括号？

****我们看以下场景：****实现一个计算2倍的宏

```
#include <stdio.h>

//定义一个宏，假设不使用括号
#define DOUBLE(x) x+x

int main()
{
    int ret = 10 * DOUBLE(5 + 1);

    //进过预处理阶段，会变为以下形式
    //int ret = 10 * 5+1+5+1

    printf("%d\n", ret);

    return 0;
}
```

运行结果：



****解析：****很明显这是一个错误的结果，怎么得到正确的结果呢，这就体现了括号的作用

以下为正确修改后的代码：

```
#include <stdio.h>

//定义一个宏，假设不使用括号
#define DOUBLE(x) ((x)+(x))

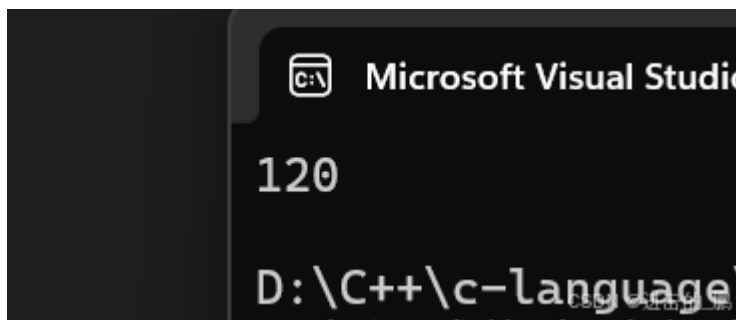
int main()
{
    int ret = 10 * DOUBLE(5 + 1);

    //进过预处理阶段，会变为以下形式
    //int ret = 10 * ((5+1)+(5+1))

    printf("%d\n", ret);

    return 0;
}
```

运行结果：



****提醒：****所以用于对数值表达式进行求值的宏定义都应该用这种方式加上括号，避免在使用宏时由于参数中的操作符或邻近操作符之间不可预料的相互作用。***总之就是不要吝啬小括号***

宏的使用与函数有点类似，但是宏是没有类型检查的，它只是完成一个替换，适用于一些简单重复的功能的实现

带有副作用的宏参数

当宏参数在宏的定义中出现超过一次的时候，如果参数带有副作用，那么你在使用这个宏的时候就可能出现危险，导致不可预测的后果。副作用就是表达式求值的时候出现的永久性效果。

什么是副作用？例如：

1. `y = x + 1;` // 不带副作用
2. `y = x++;` // 带有副作用

以上两种方式效果相同，可是第二种会修改 `x` 本身的值，这就是副作用。

MAX 宏可以证明具有副作用的参数所引起的问题：

```
#include <stdio.h>

//定义一个比较大小的宏
#define MAX(a, b) ((a) > (b) ? (a) : (b))

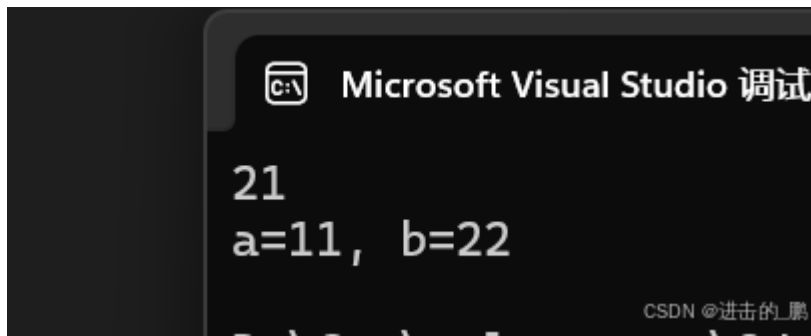
int main()
{
    int a = 10;
    int b = 20;
    int c = MAX(a++, b++);

    //经过预处理阶段后，上一条代码会变为以下形式
    //int c = ((a++) > (b++) ? (a++) : (b++));

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

    return 0;
}
```

运行结果：



****解析：****因为宏只是完成替换，所以上述b的值会被修改两次，但如果是函数的话，传入相同的参数，a和b只会被修改一次，因此我们使用宏的时候需要注意这些问题

宏替换的规则

在程序中扩展#define定义符号和宏时，需要涉及几个步骤。

1. 在调用宏时，首先对参数进行检查，看看是否包含任何由#define定义的符号。如果是，它们首先被替换。
2. 替换文本随后被插入到程序中原来文本的位置。对于宏，参数名被他们的值所替换。
3. 最后，再次对结果文件进行扫描，看看它是否包含任何由#define定义的符号。如果是，就重复上述处理过程。

注意：

1. 宏参数和#define定义中可以出现其他#define定义的符号。但是对于宏，不能出现递归。
2. 当预处理器搜索#define定义的符号的时候，字符串常量的内容并不被搜索。

四、宏与函数的对比

宏通常被应用于执行简单的运算。

比如在两个数中找出较大的一个时，写成下面的宏，更有优势一些。

```
#define MAX(a, b) ((a)>(b)?(a):(b))
```

函数：

```
int MAX(int a, int b)
{
    return a > b ? a : b;
}
```

那为什么不用函数来完成这个任务？

原因有二：

1. 用于调用函数和从函数返回的代码可能比实际执行这个小型计算工作所需要的时间更多。所以宏比函数在程序的规模和速度方面更胜一筹。
2. 更为重要的是函数的参数必须声明为特定的类型。所以函数只能在类型合适的表达式上使用。反之 这个宏怎可以适用于整形、长整型、浮点型等可以用于 > 来比较的类型。宏的参数是类型无关的。

和函数相比宏的劣势：

1. 每次使用宏的时候，一份宏定义的代码将插入到程序中。除非宏比较短，否则可能大幅度增加程序的长度。
2. 宏是没法调试的。
3. 宏由于类型无关，也就不够严谨。
4. 宏可能会带来运算符优先级的的问题，导致程容易出现错。

宏的参数可以出现类型，但是函数做不到。

例如：

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

//定义一个宏用于申请空间
#define MALLOC(num, type) (type*)malloc(num * sizeof(type))

int main()
{
    int* p = MALLOC(10, int);
    //预处理阶段上述代码会转换为以下代码：
    //int* p = (int*)malloc(10 * sizeof(int));

    return 0;
}
```

宏和函数的一个对比：

属性	#define定义宏	函数
代码长度	每次使用时，宏代码都会被插入到程序中。除了非常小的宏之外，程序的长度会大幅度增长	函数代码只出现于一个地方；每次使用函数时，都调用那个地方的同一份代码
执行速度	更快	存在函数的调用和返回的额外开销，所以对慢一些
操作符优先级	宏参数的求值是在所有周围表达式的上下文环境里，除非加上括号，否则邻近操作符的优先级可能会产生不可预料的后果，所以建议宏在书写的时候多写括号。	函数参数只在函数调用求值一次的结果值传递给函数。表达式的求值结果容易预测。
带有副作用的参数	参数可能被替换到宏体中的多个位置，如果宏的参数被多次计算，带有副作用的参数求值可能会产生不可预料的结果。	函数参数只在传参的时候求值一次，结果容易控制。
参数类型	宏的参数与类型无关，只要对参数的操作是合法的，它就可以使用于任何参数类型。	函数的参数是与类型有关的，如果参数类型不同，就需要不同的函数，即使他们的任务是不同的。
调试	宏是不方便调试的	函数是可以逐语句调试的
递归	宏是不能递归的	函数是可以递归的

CSDN @进击的_鹏

五、#和## 运算符

1.# 运算符

#运算符将宏的一个参数转换为字符串字面量。它仅允许出现在带参数的宏的替换列表中。

#运算符所执行的操作可以理解为”字符串化“。

**例如： **我们有以下代码：

```
#include <stdio.h>

int main()
{
    int a = 10;
    printf("the value of a is %d\n", a);

    int b = 20;
    printf("the value of b is %d\n", b);

    float f = 6.6f;
    printf("the value of f is %.1f\n", f);

    return 0;
}
```

运行结果:



****提问:** **上述用于打印 the value of a is 10 这串代码能否使用宏或者函数封装起来?

****答案:** **使用宏可以, 函数做不到。

函数为啥做不到呢?

```
void print(int a)
{
    printf("the value of a is % d\n", a);
}
```

原因:

1. 首先函数不能打印多类型数据

2. 其次, the value of a is 10, 中的 a 是没办法替换的, 我们的要求是 a 能根据参数名变化的。而函数中的 参数变量名 是跟字符没有联系的。因此函数做不到这一效果。

使用宏来实现:

首先我们需要了解 printf 的一个机制:

```
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    printf("Hello"" World\n");

    return 0;
}
```

运行结果:



****解析:****如图和代码, printf 其实会自动拼接字符串, 也就是说, printf 中有多个字符串, 最终都会拼接成一条字符串输出打印。

根据 printf 的这条性质, 加上 `*#` 运算符可以把宏的一个参数转换为字符串字面量*, 我们就可以以下宏:

```
#include <stdio.h>

#define PRINT(format, n) printf("the value of "#n" is "format"\n", n)

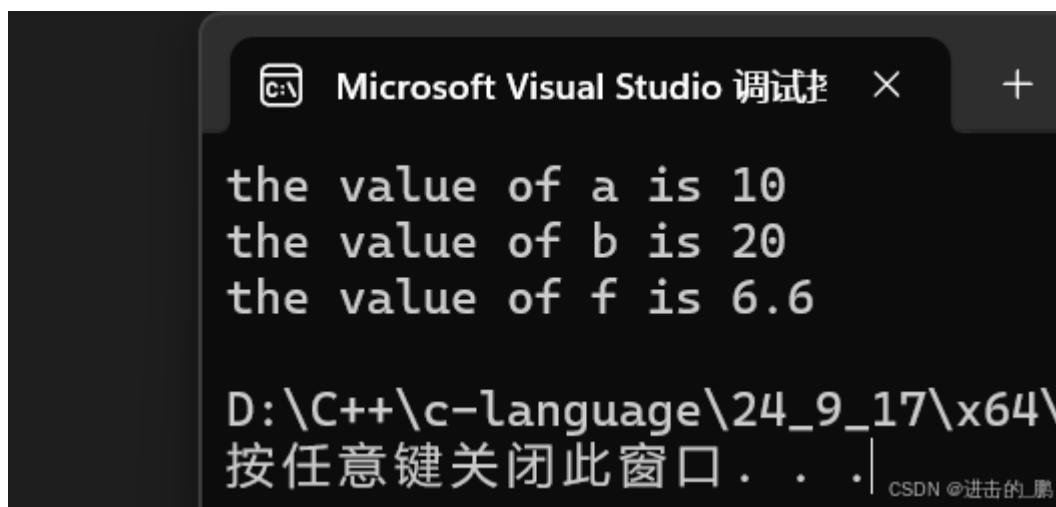
int main()
{
    int a = 10;
    PRINT("%d", a);
    //示例预处理之后的效果:
    //printf("the value of " "a" " is " "%d" "\n", n);

    int b = 20;
    PRINT("%d", b);

    float f = 6.6f;
    PRINT("%.1f", f);

    return 0;
}
```

运行结果:



```
Microsoft Visual Studio 调试 × +

the value of a is 10
the value of b is 20
the value of f is 6.6

D:\C++\c-language\24_9_17\x64\
按任意键关闭此窗口. . . | CSDN @进击的_鹏
```

2.## 运算符

可以把位于它两边的符号合成一个符号，它允许[宏定义](#)从分离的文本片段创建标识符。**## 被称为记号粘合**。这样的连接必须产生一个合法的标识符。否则其结果就是未定义的。

******比如：******写一个函数求2个数的较大值的时候，不同的数据类型就得写不同的函数。

```
#include <stdio.h>

int int_max(int a, int b)
{
    return a > b ? a : b;
}

float float_max(float a, float b)
{
    return a > b ? a : b;
}

int main()
{
    printf("%d\n", int_max(3, 4));
    printf("%.1f\n", float_max(5.9f, 6.1f));

    return 0;
}
```

运行结果：



****提问：****如果只用函数，那么我们每次换个类型比较，就需要重新写一个对应类型的函数，有没有什么办法简化这一过程呢？

答案：有的，使用宏结合 **##运算符** 就可以制作一个函数模版，简化这一过程

```
#include <stdio.h>

//宏定义，type##_max预处理之后将会合成一个符号
#define GENERIC_MAX(type) \
type type##_max(type x, type y)\
{\
    return (x > y ? x : y);\
}

//定义一个比较整形大小的函数
GENERIC_MAX(int) //注意不要加分号
//示例预处理之后：
//int int_max(int x, int y)
//{
//    return (x > y ? x : y);
//}

//定义一个比较浮点型大小的函数
GENERIC_MAX(float)

//定义一个比较字符型大小的函数
GENERIC_MAX(char)

int main()
{
    printf("%d\n", int_max(3, 4));
    printf("%.1f\n", float_max(5.9f, 6.1f));
    printf("%c\n", char_max('a', 'b'));

    return 0;
}
```

运行结果：

The image shows a screenshot of the Microsoft Visual Studio console window. The title bar at the top reads "Microsoft Visual Studio 调试" (Microsoft Visual Studio Debug) with a close button on the right. The console output displays three lines of text: "4", "6.1", and "b". Below the output, the file path "D:\C++\c-language\24_9_17\x6" is visible. At the bottom of the console, there is a prompt "按任意键关闭此窗口..." (Press any key to close this window...) and a small logo for "CSDN @进击的_鹏".

****提醒：**在实际开发过程中##使用的很少，很难取出非常贴切的例子。主要是让我们看到别人这样写的时候我们能够看懂。

六、命名约定

一般来讲函数的宏的使用语法很相似。所以语言本身没法帮我们区分二者。

那我们平时的一个习惯是：

1. 把宏名全部大写
2. 函数名不要全部大写

七、#undef

这条指令用于移除一个宏定义。

```
#undef NAME  
// 如果现存的一个名字需要被重新定义，那么它的旧名字首先要被移除。
```

示例：

```
#include <stdio.h>  
  
//定义一个常量  
#define M 100  
  
int main()  
{  
    printf("%d\n", M);  
  
    //移除对M的宏定义  
    #undef M  
  
    printf("%d\n", M);  
  
    return 0;  
}
```

运行结果：



八、条件编译

在编译一个程序的时候我们如果要将一条语句（一组语句）编译或者放弃是很方便的。因为我们有条件编译指令。

常见的条件编译指令：

1. #if #endif

```
#if 常量表达式
//...
#endif
```

功能：常量表达式成立，指向 #if 与 #endif 之间的所有代码，不成立则不执行

示例：

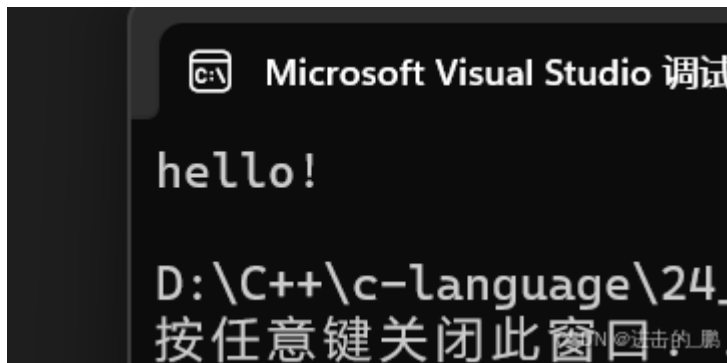
```
#include <stdio.h>

//定义一个常量
#define M 2

int main()
{
    //条件指令，条件成立执行与 endif 之间的代码，否则不执行
    #if M == 2
        printf("hello!\n");
    #endif

    return 0;
}
```

运行结果：



*****注意：****一定是*常量表达式*，不能是自己定义的变量，可以是#define定义的常量，因为变量是在程序编译后运行时生成的，而预处理指令是在预处理阶段执行的。*

因为表达式为假不执行代码，所以可以用来注释多行代码：

```
#if 0

#include <stdio.h>

int main()
{
    printf("hello!\n");

    return 0;
}

#endif
```

2.多个分支的条件编译

```
#if 常量表达式
//...
#elif 常量表达式
//...
#else
//...
#endif
```

****功能：*****效果与if else if else相似，只不过是在预处理阶段执行且必须是常量表达式*

示例：

```
#include <stdio.h>

//定义一个常量
#define M 4

int main()
{
    #if M == 1
        printf("1\n");
    #elif M == 2
        printf("2\n");
    #elif M == 3 //#elif可以有多个
        printf("3\n");
    #else
        printf("%d\n", M);
    #endif

    return 0;
}
```

运行结果:



3.判断是否被定义

这两种效果一样，下面一种是简写

```
#if defined(symbol)
```

```
#ifdef symbol
```

****功能：** **检查 **symbol** 是否用 **#define** 定义过。

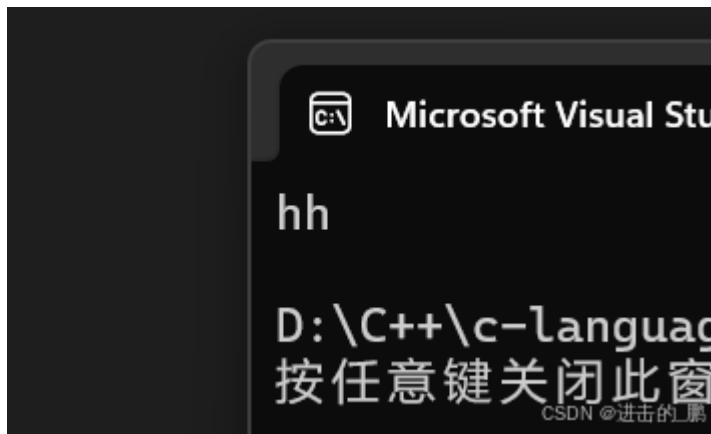

```
#include <stdio.h>

//定义一个常量
#define M 4

int main()
{
#ifdef M
    printf("hh\n");
#endif

    return 0;
}
```

运行结果：



****注意：****同样要与 #endif 搭配使用

判断是否被定义还有一种相反的写法：

同样也有两种写法：

```
#if !defined(symbol)
```

```
#ifndef symbol
```

****功能：****用法相同，效果相反，判断是否没有被定义

示例：

```
#include <stdio.h>

int main()
{
#ifdef M
    printf("hh\n");
#endif

    return 0;
}
```

运行结果:



4. 嵌套指令

```
#if defined(OS_UNIX)
#ifdef OPTION1
    unix_version_option1();
#endif
#ifdef OPTION2
    unix_version_option2();
#endif
#elif defined(OS_MSDOS)
#ifdef OPTION2
    msdos_version_option2();
#endif
#endif
```

嵌套使用，与分支语句形式一样，这里不再举例

九、头文件的包含

1.头文件被包含的方式：

经常写头文件的朋友应该都知道，我们包含头文件的方式有两种，那么这两种包含方式有什么区别呢？答案：**区别在于查找策略不同**

1.本地文件包含

```
#include "filename"
```

****查找策略：****先在源文件所在目录下查找，如果该头文件未找到，编译器就像查找库函数头文件一样在标准位置查找头文件。如果找不到就提示编译错误。

2.库文件包含

```
#include
```

****查找策略：****查找头文件直接去标准路径下去查找，如果找不到就提示编译错误。

这样是不是可以说，对于库文件也可以使用 " " 的形式包含？

答案是肯定的，可以，**但是这样做查找的效率就低些，当然这样也不容易区分是库文件还是本地文件了，所以不建议。**

2.嵌套文件包含

首先我们要知道一个常识：就是如果没有任何指令的情况下，一个头文件被包含了多次，那么它就会被编译多次。也就是说如果一个头文件被包含10次，那就实际被编译10次，如果重复包含，对编译的压力就比较大。

如下：

test.h头文件：

```
void test();

struct Stu
{
    int id;
    char name[20];
};
```

test.c:

```
#include "test.h"
#include "test.h"
#include "test.h"
#include "test.h"
#include "test.h"

int main()
{

    return 0;
}
```

****解释:****如果直接这样写，test.c 文件中将 test.h 包含5次，那么 test.h 文件的内容将会被拷贝5份在 test.c中。如果 test.h 文件比较大，这样预处理后代码量会剧增。如果工程比较大，有公共使用的头文件，被大家都能使用，又不做任何的处理，那么后果真的不堪设想。

虽然实际中不会这样直接包含5次，当在一些多文件情况下，我们可能不得已间接包含多次，所以如何解决呢？答案：*条件编译。*

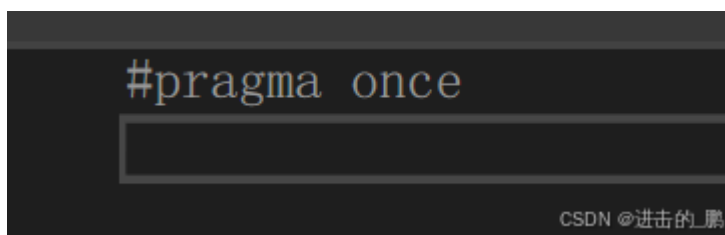
每个头文件的开头写：

```
#ifndef __TEST_H__
#define __TEST_H__
    //头文件的内容
#endif
```

解析：#ifndef **TEST_H** 用于判断 **TEST_H** 是否没有被定义，如果重复包含这里就会被判断已经被定义过，那么整个头文件内容将不会执行。如果没有定义，就 #define **TEST_H** 定义一个该名称。用于下次判断。

当然还有一个简写的方法，使用VS的经常可以看到

```
#pragma once
```



其它预处理指令

预处理名称	意义
#define	宏定义
#undef	撤销已定义过的宏定义
#include	使编译程序将另一源文件嵌入到带有#include 的源文件中
#if	#if 的一般含义是如果#if 后面的常量表达式为 true，则编译它与#endif 之间的代码，否则跳过这些代码。命令#endif 标识一个#if 块的结束。#else 的功能有点像 C 语言中的 else，#else 建立另一种选择（在#if 失败的情况下）。#elif 命令意义与 else if 相同，它形成一个 if else-if 阶梯状语句，可以进行多种编译选择
#else	
#elif	
#endif	
#ifdef	#ifdef 与#ifndef 命令分别表示“如果有定义”及“如果无定义”，是条件编译另一种方法
#ifndef	
#line	改变当前行数和文件名称，它们是在编译程序中预定义的标识符。 命令基本形式如下： #line number["filename"];
#error	编译程序时，只要遇到#error 就会生成一个编译错误提示信息，并停止编译
#pragma	为实现时定义的命令，它允许向编译程序传送各种指令例如，编译程序可能有一种选择，它支持对程序执行的跟踪。可用#pragma 指定一个跟踪选择