全局变量，静态变量都放在静态区,静态变量不初始化的时候，默认会被初始化0
局部变量，是放在栈区，不初始化，默认值是随机值
sizeof这个操作符计算返回的结果是size_t类型的，是无符号整型的

数据储存

1.//整型储存///
1.数据类型
sizeof(long)>=sizeof(int) ===> 同时满足>=4
2.类型的意义
使用这个类型开辟内存空间的大小（大小决定了使用范围）
如何看待内存空间的视角

字节大小

`char` -- 1
`short` -- 2
`int` -- 4
`long` -- 4/8
`long long` -- 8
`float` -- 4
`double` -- 8

c89标准的

c18

类型的基本归类

1.整型家族
char,short,int,long,long long
2.浮点数家族
float,double
3.构造类型
数组类型，结构体类型，枚举类型，联合类型
4.指针类型
5.空类型
void 表示空类型（无类型）
通常应用于函数的返回类型，函数的参数，指针类型。
5.整型在内存中的储存
一个变量的创建是要在内存中开辟空间的。空间的大小是根据不同的类型而决定的
1.原码，反码，补码
1.正的整数，原码，反码，补码相同
2.负的整数，原码，反码，补码是需要计算的
原码：直接通过正负的形式写出的二进制序列就是原码
反码：原码的符号位不变，其他位按位取法得到的就是反码
补码：反码+1就是补码
eg：int b = -10
//10000000000000000000001010 --- 原码

//11111111111111111111110101 --- 反码
//11111111111111111111110110 --- 补码
整数内存中存放是补码的二进制序列

6.为什么电脑用补码记录

在计算机系统中,数值一律用补码来表示和存储。原因在于,使用补码,可以将符号位和数值域统一处理;同时,加法和减法也可以统一处理(CBU只有加法器)此外,补码与原码相互转换,其运算过程是相同的,不需要额外的硬件电路。

7. 内存储存

eg:0x11 22 33 44

1.大端字节序存储 --- 正着储存(把一个数据的高位字节序的内容存放在低地址处，把低位字节符的内容放在高地址处)

2.小端字节序储存 --- 倒着储存(把一个数据的高位字节序的内容存放在高地址处，把低位字节符的内容放在低地址处)

8.取值范围

有符号的char

范围：-128—127

无符号的char

范围：0—255

%d 是打印有符号的整型

%u 是打印无符号的整数

注：strlen是求字符串的长度，关注的是字符串中\0之前出现多少字符

strlen的返回值为无符号整数

无符号的数与无符号的数相减得到的是一个无符号的数

2.//浮点型在内存中的储存//

1.整型储存与浮点型储存的方式不一样，两者互存可能会出错

2.浮点数的储存规则

3.//指针进阶//

1.字符指针

char* 的指针

char* p = "abcdef";//把字符串首字符a的地址，赋值给了p

p是常量，不能改变

2.指针数组

是数组，是用来存放指针的数组

int* arr2[6];//存放整型指针的数组

char* arr3[5];//存放字符指针的数组

注：*(p+i) --> p[i]*

3.数组指针 -- 指针

整型指针 -- 指向整型的指针(用来存放整型的地址)

字符指针 -- 指向字符的指针(用来存放字符的地址)

数组指针 -- 指向数组的指针(用来存放数组的地址)

int p1[10];//p1是指针数组

int(*p2)[10];//p2是数组指针*

4.数组名

1.数组名通常表示的都是数组首元素的地址

//但是有两个例外：

//1.sizeof(数组名)，这里的数组名表示整个数组，计算的是整个数组的大小(单位是字节)

//2.&函数名，这里的数组名表示的依然是整个数组，所以&数组名取出的是整个数组的地址

*5.int (*p)[5];*

p的类型：int*()[5];*

p是指向一个整型数组的，数组5个元素 int [5]
p+1 -> 跳过一个五个int元素的数组
*6.int (*parr3[10])[5];
parr3应该是存放数组指针的数组
7.数组参数，指针参数
形参的二维数组，行可以省略，列不能省略！！
二维数组的数组名，表示首元素的地址，其实是第一行的地址
8.如果函数的形式参数是二级指针，调用函数的时候可以穿相符合的实参
9.函数指针 -- 指向函数的指针
1.对于函数来说，&函数名和函数名都是函数的地址
2.函数指针
int Add(int x, int y)
{
return x + y;
}
int main()
{

```
    int  (*pf)(int, int) = &Add;
    int ret = (*pf)(2, 3);
    printf("%d\n", ret);
    return 0;
}
3.typedef void(* pf_t)(int);
把void(*)(int)类型重命名为pf_t
```

10.回调函数
回调函数就是一个通过函数指针调用的函数。如果你把函数的指针(地址)作为参数传递给另一个函数,当这个指针被用来调用其所指向的函数时,我们就说这是回调函数。回调函数不是由该函数的实现方直接调用,而是在特定的事件或条件发生时由另外的一方调用的,用于对该事件或条件进行响应。
11.库函数qsort() - 这个函数可以排序任意类型的数据
使用快速排序的思想进行排序
void qsort(void* base,//你要排序的数据的起始位置
size_t num,//待排序的数据元素的个数
size_t width,//待排序的数据元素的大小(单位是字节)
int(*cmp)(const void* el, const void* e2)//函数指针 - 比较函数
);
12.void* 是无具体类型的指针，可以接受任意类型的地址
void* 不能解引用操作，也不能+-整数
13.内存 -----> 内存的单元(1byte) ---> 编号 ---> 地址 --->指针
所以指针就是一个地址而已
sizeof(数组名)，数组名表示整个数组，计算的是整个数组的大小，单位是字节
sizeof(a+0)，a不是单独放在sizeof内部，也没有取地址，所以a就是首元素的地址
sizeof(*a)*，a中的a是数组首元素的地址，*a就是对首元素的地址解引用，找到的就是首元素
&arr --> 数组的地址，是数组的地址就是4/8个字节

虚拟地址 --> 物理地址
printf("%d\n", sizeof(*arr));：这将返回arr指向的元素的大小，即字符的大小。在C语言中，字符的大小是1字节。*
printf("%d\n", sizeof(&arr));：这里&arr是指向整个数组的指 针。不过，sizeof(&arr)给出的是指

针的大小，而不是　　数组的大小。所以，这和sizeof(arr+0)一样，根据系统的不同，结果可能是4字节或8字节。

//a[0]是第一行这个一维数组的数组名,单独放在sizeof内部,a[0]表示第一个整个这个一维数组，sizeof(a[0])计算的就是　　第一行的大小

//a[0]并没有单独放在sizeof内部,也没取地址,a[的就表示首元素的地址，就是第一行这个一维数组的第一个元素的地址

sizeof(a+1)

//a虽然是二维数组的地址,但是并没有单独放在sizeof内部,也没取地址

//a表示首元素的地址,二维数组的首元素是它的第一行,a就是第一行的地址

//a+1就是跳过第一行,表示第二行的地址(是地址就是4/8的字节)

//(a+1)是对第二行地址的解引用，拿到的是第二行

14.

数组名的意义:

1.sizeof(数组名),这里的数组名表示整个数组,计算的是整个数组的大小。

2.&数组名,这里的数组名表示整个数组,取出的是整个数组的地址。

3.除此之外所有的数组名都表示首元素的地址。

15.

(unsigned long)p 将指针 p 转换为一个无符号长整型整数。这样做是为了确保在执行算术操作时，指针值被解释为无　　符号整数而不是内存地址。

- 0x1 将无符号长整型整数的值增加 1。

  16.

  指针的运算是基于指针所指向的数据类型的大小的。在这个例子中，ptr1[-1]其实等价于*(ptr1 - 1)，即从ptr1指向的　位置向前移动一个int类型的大小，然后取得该位置上的值。

//字符函数和字符串函数//

1.求字符串长度

strlen

2.长度不受限制的字符串函数

strcpy，strcat，strcmp

3.长度受限制的字符串函数介绍

strncpy，strncat，strncmp

4.字符串查找

strstr，strtok

5.错误信息报告

strerror

6.字符操作

7.内存操作函数

memcpy，memmove，memset，memcmp

注：

1. `strcpy`里的值不可变

8. `strcpy`，`strcat`，`strcmp`都是长度不受限制的字符串函数

9.strncpy,strncat,strncmp都是长度受限制的字符串函数

10.strstr：查找子串的一个函数

11.strerror -- 返回错误码，所对应的错误信息

errno - C语言设置的一个全局的错误码存放的变量

12.memcpy - 只能拷贝字符串(memcpy函数是不用来处理重叠的内存之间的数据拷贝的

13.memmove负责拷贝两块独立空间中的数据(可以实现重叠内存之间的数据拷贝)

14.void是C语言中的一个特殊类型，用于表示指向未指定类型的内存地址的指针。它的主要作用是在编程时提供一种灵活的方式来处理各种类型的数据。通常情况下，当我们不确定某个指针所指向的数据的

确切类型时，就可以使用void类型。例如，在编写通用的数据结构或函数时，可能需要使用void来处理不同类型的数据。然后，通过在运行时进行类型转换，可以将void指针转换为具体的数据类型指针，以便对数据进行操作。使用void*需要小心，因为它使得编译器无法对指针所指向的数据类型进行类型检查，因此在使用时需要确保类型转换的正确性。

自定义类型:结构体，枚举，联合

1.结构体

1.结构体的声明

1.1结构体的基础知识

结构是一些值的集合,这些值称为成员变量。结构的每个成员可以是不同类型的变量。

1.2结构体的声明

1.
```
struct tag
{
  member - list;
}variable-list;
```

2.
```
struct stu
{
  char name[20];
  int age;
}s1，s2;
```

这里的s1和s2是struct stu类型的变量

3.
```
struct stu
{
  char name[20];
  int age;
}s1，s2;//s1，s2是全局变量
int main()
{
  struct stu s3;//s3是局部变量
  return 0;
}
```

1.3特殊的声明

在声明结构的时候，可以不完全的声明

```
//匿名结构体类型

//只能使用一次

struct
{
  int a;
  char b;
```

float c;
}x;
struct
{
int a;
char b;
float c;
}a[20],*p;

1.4结构体的自引用

1.

```
//用结构体来创建一个链表
  struct Node
  {
      int data;
      struct Node* next;
  };
```

2.

```
typedef struct Node
  {
      int data;
      struct Node* next;
  }* linklist;
```

typedef struct Node* 重命名为 linklist

1.5结构体变量的定义和初始化

```
  struct Point
  {
      int x;
      int y;
  }p1 = { 2,3 };
  struct score
  {
      int n;
      char ch;
  };
  struct stu
  {
      char name[20];
      int age;
      struct score s;
  };
  int main()
```

```
{
    struct Point p2 = { 3,4 };
    struct stu s1 = { "zhangsan",20,{100,'q'} };

  printf("%s %d %d %c\n", s1.name, s1.age, s1.s.n, s1.s.ch);

  return 0;
}
```

1.6结构体内存对齐

计算结构体的大小

首先得掌握结构体的对齐规则:

1.第一个成员在与结构体变量偏移量为0的地址处。
2.其他成员变量要对齐到某个数字(对齐数)的整数倍的地址处。
对齐数=编译器默认的一个对齐数与该成员大小的较小值。(VS中默认的值为8。)

3.结构体总大小为最大对齐数(每个成员变量都有一个对齐数)的整数倍。
4.如果嵌套了结构体的情况,嵌套的结构体对齐到自己的最大对齐数的整数倍处,结构体的整体大小就是所有最大对齐数(含嵌套结构体的对齐数)的整数倍。

6.gcc上没有默认对齐数

7.为什么存在内存对齐?

大部分的参考资料都是如是说的:

1.平台原因(移植原因):
不是所有的硬件平台都能访问任意地址上的任意数据的;某些硬件平台只能在某些地址处取某些特定类型的数据,否则抛出硬件异常。

2.性能原因:

数据结构(尤其是栈)应该尽可能地在自然边界上对齐。
原因在于,为了访问未对齐的内存,处理器需要作两次内存访问;而对齐的内存访问仅需要一次访问。

总体来说:

结构体的内存对齐是拿空间来换取时间的做法。

8.如何既满足对齐，又要节省空间呢

让占用空间小的成员尽量集中在一起

1.7修改默认对齐数

用#pragma这个预处理指令

#pragma once(头文件中使用，功能是：防止头文件被多次引用)

1.8结构体传参

在C语言中，`->` 是一个成员访问运算符，通常用于访问结构体或联合体类型的成员，以及指向结构体或联合体类型的指针的成员。它的作用是通过指针来访问结构体或联合体类型的成员，以简化代码书写和提高可读性。

例如，如果有一个结构体 `struct Person` 包含成员 `name` 和 `age`，如果 `personPtr` 是一个指向 `struct Person` 的指针，可以通过 `personPtr->name` 来访问 `name` 成员，通过 `personPtr->age` 来访问 `age` 成员，这样就不需要使用间接访问运算符 `*` 结合结构体成员访问运算符 `.` 来访问结构体成员了。

因此，`->` 的作用是通过指针访问结构体或联合体类型的成员，简化了对指针所指向的对象成员的访问。

```
struct S
{
    int data[1000];
    int num;
};
void print1(struct S ss)
{
    int i = 0;
    for (i = 0; i < 3; i++)
    {
        printf("%d ", ss.data[i]);
    }
    printf("%d\n", ss.num);
}
void print2(const struct S* ps)
{
    int i = 0;
    for (i = 0; i < 3; i++)
    {
        printf("%d ", ps->data[i]);
    }
    printf("%d\n", ps->num);
}
int main()
{
    struct S s = { {1,2,3},100 };
    print1(s);//传值调用
    print2(&s);//传址调用
    return 0;
}
```

上面的print1和print2函数哪个好些?

答案是:首选print2函数。
原因:

函数传参的时候,参数是需要压栈,会有时间和空间上的系统开销。

如果传递一个结构体对象的时候,结构体过大,参数压栈的的系统开销比较大,所以会导致性能的下降。

结论：

结构体传参的时候，要传结构体的地址

2.位段

2.1什么是位段

位段的声明和结构是类似的,有两个不同:

1.位段的成员必须是int、unsigned int 或signed int。(实际上只要是整型就行)

2.位段的成员名后边有一个冒号和一个数字。

比如：

```
struct A
{
    int _a : 2;
    int _b : 5;
    int _c : 10;
    int _d : 30;
};
```

A就是一个位段类型

位段可以节省空间

2.2位段的内存分配

1.位段的成员可以是int unsigned int Signed int或者是char(属于整形家族)类型
2.位段的空间上是按照需要以4个字节(int)或者1个字节(char)的方式来开辟的。
3.位段涉及很多不确定因素,位段是不跨平台的,注重可移植的程序应该避免使用位段。

比如：

```
struct A
{
    //4byte - 32bit

  int _a : 2;
  int _b : 5;
  int _c : 10;
  //还剩15个bit
  //4byte - 32bit
  int _d : 30;

};
```

2.3位段的跨平台问题

1.int 位段被当成有符号数还是无符号数是不确定的。

2.位段中最大位的数目不能确定。(16位机器最大16,32位机器最大32,写成27,在16位机器会出问题。
3.位段中的成员在内存中从左向右分配,还是从右向左分配标准尚未定义。

4.当一个结构包含两个位段,第二个位段成员比较大,无法容纳于第一个位段剩余的位时,是舍弃剩余的位还是利用,这是不确定的。

总结:

跟结构相比,位段可以达到同样的效果,但是可以很好的节省空间,但是有跨平台的问题存在。

2.4位段的应用

3.枚举

枚举顾名思义就是 -- 列举

把可能的取值一一列举

3.1枚举类型的定义

```
enum Day
{
    Mon,
    Tues,
    Wed,
    Thur,
    Fri,
    Sat,
    Sun
};
```

以上定义的enum Day是枚举类型

{}中的内容是枚举类型的可能取值，也叫枚举变量

这些取值可能都是`

```
  enum Color
  {
      RED = 1,
      GREEN = 2,
      BLUE = 4
  };
```

3.2枚举的优点

为什么使用枚举?

我们可以使用#define 定义常量,为什么非要使用枚举?
枚举的优点:

1.增加代码的可读性和可维护性
2.和#define定义的标识符比较枚举有类型检查,更加严谨。
3.防止了命名污染(封装)
4.便于调试
5.使用方便,一次可以定义多个常量

3.3枚举的使用

```
enum Color//颜色
{
    RED = 1,
    GREEN = 2,
    BLUE = 4
};

enum Color clr = GREEN;//只能拿枚举常量给枚举变量赋值，才不会出现类型的差异
clr = 5;//ok??   (no ok)
```

4.联合体(共用体)

4.1联合类型的定义

联合也是一种特殊的自定义类型
这种类型定义的变量也包含一系列的成员,特征是这些成员公用同一块空间(所以联合也叫共用体)。
比如:

```
//联合类型的声明
union Un
{
    char c:
    int i;
};
//联合变量的定义
union Un un;
//计算连个变量的大小
printf("%d\n", sizeof(un));
```

注:

联合相当于合租房，而结构体相当于独立房间

4.2联合的特点

联合的成员是共用同一块内存空间的,这样一个联合变量的大小,至少是最大成员的大小(因为联合至少得有能力保存最大的那个成员)。

```
union Un
{
    int i;
    char c;
};
union Un un;

//下面输出的结果是一样的吗?
printf("%d\n", &(un.i));
printf("%d\n", &(un.c));

//下面输出的结果是什么?
un.i = 0x11223344;
un.c = 0x55;
printf("%x\n", un.i);
```

4.3联合大小的计算

· 联合的大小至少是最大成员的大小。
· 当最大成员大小不是最大对齐数的整数倍的时候,就要对齐到最大对齐数的整数倍。

动态内存管理

malloc,calloc,realloc,free

1.为什么存在动态内存分配

我们已经掌握的内存开辟方式有:

`int val= 20;//在栈空间上开辟四个字节`
`char arr[10]={0};//在栈空间上开辟10个学节的连续空间`

但是上述的开辟空间的方式有两个特点:

1.空间开辟大小是固定的。
2.数组在申明的时候,必须指定数组的长度,它所需要的内存在编译时分配。

但是对于空间的需求,不仅仅是上述的情况。有时候我们需要的空间大小在程序运行的时候才能知道,那数组的编译时开辟空间的方式就不能满足了。
这时候就只能试试动态存开辟了。

2.动态内存函数的介绍

栈区:

局部变量，形式参数

堆区:

malloc,calloc,realloc,free

静态区:

变长数组

```
int main()
 {
        int n = 0;
        scanf("%d",&n);
        int arr2[n];
}
```

2.1malloc和free

C语言提供了一个动态内存开辟的函数:

`void* malloc (size_t size);`

这个函数向内存申请一块连续可用的空间,并返回指向这块空间的指针。

· 如果开辟成功,则返回一个指向开辟好空间的指针。
· 如果开辟失败,则返回一个NULL指针,因此malloc的返回值一定要做检查。
· 返回值的类型是void*,所以malloc函数并不知道开辟空间的类型,具体在使用的时候来决定。
· 如果参数 `size` 为 `0` ,`malloc` 的行为是标准是未定义的,取决于编译器。

C语言提供了另外一个函数free,专门是用来做动态内存的释放和回收的,函数原型如下:

void free (void* ptr);

free函数用来释放动态开辟的内存。

· 如果参数ptr指向的空间不是动态开辟的,那free函数的行为是未定义的。
· 如果参数ptr是NULL指针,则函数什么事都不做。

malloc和free都声明在stdlib.h头文件中。

2.2calloc

C语言还提供了一个函数叫 calloc,calloc函数也用来动态内存分配。原型如下:

void* calloc (size_t num, size_t size);

· 函数的功能是为num个大小为size的元素开辟一块空间,并且把空间的每个字节初始化为0。
· 与函数ma1loc的区别只在于calloc会在返回地址之前把申请的空间的每个字节初始化为全0。

注:

calloc = malloc+memset

2.3realloc

· realloc函数的出现让动态内存管理更加灵活。
· 有时会我们发现过去申请的空间太小了,有时候我们又会觉得申请的空间过大了,那为了合理的时候内存,我们一定会对内存的大小做灵活的调整。那realloc函数就可以做到对动态开辟内存大小的调整。
函数原型如下:

void* realloc (void* ptr, size_t size);

· ptr是要调整的内存地址
· size 调整之后新大小

· 返回值为调整之后的内存起始位置。
· 这个函数调整原内存空间大小的基础上,还会将原来内存中的数据移动到新的空间。

· realloc在调整内存空间的是存在两种情况:
○ 情况1:原有空间之后有足够大的空间

o 情况2:原有空间之后没有足够大的空间

情况1
当是情况1的时候,要扩展内存就直接原有内存之后直接追加空间,原来空间的数据不发生变化。
情况2
当是情况2的时候,原有空间之后没有足够多的空间时,扩展的方法是:在堆空间上另找一个合适大小的连续空间来使用。这样函数返回的是一个新的内存地址。
由于上述的两种情况,realloc函数的使用就要注意一些。

eg:

```
int main()
{
    realloc(NULL, 40);//malloc(40);
    return 0;
}
```

内存池:

是一种用于管理内存分配和释放的技术。它通常用于提高内存分配和释放的效率，减少频繁的系统调用和内存碎片化。内存池会预先分配一块连续的内存空间，然后根据需要将这块内存空间切割成多个小块，称为内存块或内存池块。应用程序可以从内存池中请求内存块，并在不再需要时将其释放回内存池，而不是直接向操作系统请求内存。

内存池的主要优点包括：

1. **减少内存碎片化**：内存池可以避免频繁的内存分配和释放，从而减少内存碎片化的发生，提高内存的利用率。
2. **提高性能**：内存池预先分配了一定数量的内存块，避免了频繁的系统调用，从而提高了内存分配和释放的性能。
3. **降低内存泄漏的风险**：内存池的管理机制可以更好地控制内存的分配和释放，减少了内存泄漏的可能性。
4. **定制化内存分配策略**：内存池可以根据具体的需求定制化内存分配策略，例如可以实现自定义的内存分配算法或者内存对齐方式。

内存池通常用于对频繁进行内存分配和释放的场景，比如网络服务器、数据库等高性能应用程序中。它可以帮助提高系统的稳定性和性能表现。

3.常见的动态内存错误

3.1对NULL指针的解引用操作

```
void test()
{
    int* p = (int*)malloc(INT_MAX/4);
    *p = 20;//如果p的值是NULL，就会有问题
    free(p);
}
```

3.2对动态开辟空间的越界访问

```
void test()

{

int i = 0;
int *p = (int *)malloc(10*sizeof(int));
if(NULL == p)

{

•   exit(EXIT_FAILURE);

}

for(i=0; i <= 10; i++)

{

•   *(p+i)=i;//当i是10的时候越界访问

}

free(p);

}
```

3.3对非动态开辟内存使用free释放

```
void test()

{

int a = 10;
int *p = &a;
free(p);//ok?

}
```

3.4使用free释放一块动态开辟内存的一部分

```
int main()
{
    int* p = (int*)malloc(40);
    if (p == NULL)
    {
        return 1;
    }
    //使用
    int i = 0;
    for (i = 0; i <= 10; i++)
    {
        *p = i;//正确的写法：p[i]或*(p+i)
        p++;
    }
    //释放
    free(p);
    p = NULL;
    return 0;
}
```

3.5对同一块动态内存多次释放

```
int main()
{
    int* p = (int*)malloc(40);
    free(p);
    //p = NULL; 正确写法
    free(p);
    return 0;
}
```

3.6动态开辟内存忘记释放(内存泄漏)

```
void test()
{
    int* p = (int*)malloc(100);
    //....
    int flag = 0;
    scanf("%d", &flag);
    if (flag == 5)
```

```
    {
        return;
    }
    free(p);
    p = NULL;
}
int main()
{
    test();
    return 0;
}
```

忘记释放不再使用的动态开辟的空间会造成内存泄漏

切记:

动态开辟的空间一定要释放，并且正确释放

6.柔性数组

也许你从来没有听说过柔性数组(flexible array)这个概念,但是它确实是存在的。
C99中,结构中的最后一个元素允许是未知大小的数组,这就叫做『柔性数组』成员。

例如:

```
typedef struct st_type
{
    int i;
    int a[0];//柔性数组成员
}type_a;
```

6.1柔性数组的特点

· 结构中的柔性数组成员前面必须至少一个其他成员。
· sizeof返回的这种结构大小不包括柔性数组的内存。
· 包含柔性数组成员的结构用malloc()函数进行内存的动态分配,并且分配的内存应该大于结构的大小,以适应柔性数组的预期大小。

6.2柔性数组的使用

```
p->a[i] = i;

//代码1
int i = 0;
type_a *p= (type_a*)malloc(sizeof(type_a)+100*sizeof(int));
//业务处理
p->i = 100;
for(i=0; i<100; i++)
{
    p->a[i] = i;
}
free(p);
```

这样的柔性数组成员a,相当于获得了100个整型元素的连续空间

6.3柔性数组的优势

第一个好处是:方便内存释放

如果我们的代码是在一个给别人用的函数中,你在里面做了二次内存分配,并把整个结构体返回给用户。用户调用free可以释放结构体,但是用户并不知道这个结构体内的成员也需要free,所以你不能指望用户来发现这个事。所以,如果我们把结构体的内存以及其成员要的内存一次性分配好了,并返回给用户一个结构体指针,用户做一次free就可以把所有的内存也给释放掉。

第二个好处是:这样有利于访问速度.

连续的内存有益于提高访问速度,也有益于减少内存碎片。(其实,我个人觉得也没多高了,反正你跑不了要用做偏移量的加法来寻址)

注:

柔性数组只能在程序中存在一个

7.文件

7.1为什么使用文件

我们前面学习结构体时,写了通讯录的程序,当通讯录运行起来的时候,可以给通讯录中增加、删除数据,此时数据是存放在内存中,当程序退出的时候,通讯录中的数据自然就不存在了,等下次运行通讯录程序的时候,数据又得重新录入,如果使用这样的通讯录就很难受。

我们在想既然是通讯录就应该把信息记录下来,只有我们自己选择删除数据的时候,数据才不复存在。这就涉及到了数据持久化的问题,我们一般数据持久化的方法有,把数据存放在磁盘文件、存放到数据库等方式。

使用文件我们可以将数据直接存放在电脑的硬盘上,做到了数据的持久化。

7.2什么是文件

磁盘上的文件是文件。

但是在程序设计中,我们一般谈的文件有两种:程序文件、数据文件(从文件功能的角度来分类的)。

7.2.1程序文件

包括源程序文件(后缀为.c),目标文件(windows环境后缀为.obj),可执行程序(windows环境后缀为.exe)。

7.2.2数据文件

文件的内容不一定是程序,而是程序运行时读写的数据,比如程序运行需要从中读取数据的文件,或者输出内容的文件。

7.2.3文件名

个文件要有一个唯一的文件标识,以便用户识别和引用。

文件名包含3部分:文件路径+文件名主干+文件后缀

例如:c:\code\test.txt

为了方便起见,文件标识常被称为文件名。

7.3文件的打开和关闭

7.3.1文件指针

缓冲文件系统中,关键的概念是“文件类型指针”,简称“文件指针”。

每个被使用的文件都在内存中开辟了一个相应的文件信息区,用来存放文件的相关信息(如文件的名字,文件状态及文件当前的位置等)。这些信息是保存在一个结构体变量中的。该结构体类型是有系统声明的,取名FILE.

不同的C编译器的FILE类型包含的内容不完全相同,但是大同小异。

每当打开一个文件的时候,系统会根据文件的情况自动创建一个FILE结构的变量,并填充其中的信息,使用者不必关心细节。

一般都是通过一个FILE的指针来维护这个FILE结构的变量,这样使用起来更加方便。

下面我们可以创建一个FILE*的指针变量:

```
FILE* pf;//文件指针变量
```

定义pf是一个指向FILE类型数据的指针变量。可以使pf指向某个文件的文件信息区(是一个结构体变量)。通过该文件信息区中的信息就能够访问该文件。也就是说,通过文件指针变量能够找到与它关联的文件。

7.3文件的打开和关闭

文件在读写之前应该先打开文件,在使用结束之后应该关闭文件。

在编写程序的时候,在打开文件的同时,都会返回一个FILE*的指针变量指向该文件,也相当于建立了指针和文件的关系。

ANSIC规定使用fopen函数来打开文件,fclose来关闭文件。

//打开文件

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

//关闭文件

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

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，建议一个新的文件	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

7.4文件的顺序读写

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

任何一个C程序,只要运行起来就会默认打开3个流:
FILE* stdin -标准输入流(键盘)
FILE* stdout-标准输出流(屏幕)
FILE* stderr - 标准错误流(屏幕)

7.4.1对比一组函数

```
scanf/fscanf/sscanf
printf/fprintf/sprintf
```

`scanf`是针对标准输入的格式化输入语句
`prinf`是针对标准输出的格式化输出语句

`fscanf`是针对所有输入流的格式化输入语句
`fprintf`是针对所有输出流的格式化输出语句

`sscanf` 从一个字符串中转化处一个格式化的数据
`sprintf`是把一个格式化的数据转化成字符串

7.5文件的随机读写

7.5.1 fseek

根据文件指针的位置和偏移量来定位文件指针

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

例子:

```
/* fseek example */
#include <stdio.h>
```

```
int main ()
{

FILE * pFile;
pFile = fopen ( "example.txt", "wb" );
fputs ( "This is an apple.", pFile );
fseek ( pFile, 9, SEEK_SET);
fputs (" sam", pFile );
fclose ( pFile );
return 0;

}
```

7.5.2 ftell

返回文件指针相对于起始位置的偏移量

`long int ftell(FILE* stream);`

例子:

```
/* ftell example : getting size of a file */
#include <stdio.h>

int main ()
{
FILE * pFile;
long size;

pFile = fopen ("myfile.txt","rb");
if (pFile == NULL) perras ("Error opening file");
else
{
fseek (pFile, 0, SEEK_END);// non-portable
size=ftell (pFile);
fclose (pFile);
printf ("size of myfile.txt: %ld bytes.\n",size);
}
return 0;
}
```

7.5.3 rewind

让文件指针的位置回到文件的起始位置

`void rewind(FILE* stream);`

例子:

```
/* rewind example */
int main ()
```

```
{
int n;
FILE * pFile;
char buffer [27];

pFile = fopen ("myfile.txt","w+");
for ( n='A' ; n <= 'z' ; n++)
fputc (n, pFile);
rewind (pFile);
fread (buffer,1,26,pFile);
fclose (pFile);
buffer[26]='\0';
puts (buffer);
return 0;
}
```

7.6文本文件和二进制文件

根据数据的组织形式,数据文件被称为文本文件或者二进制文件。

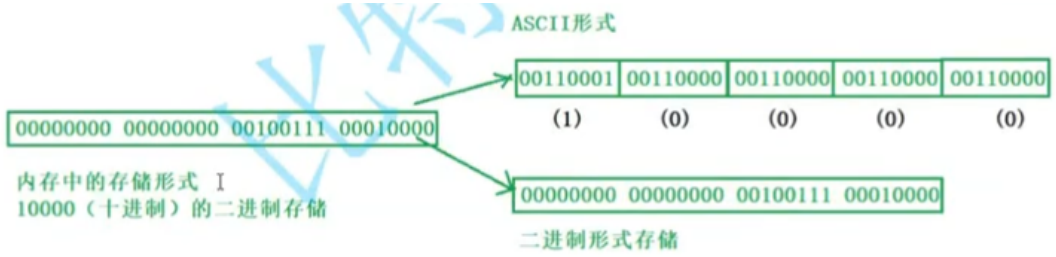
数据在内存中以二进制的形式存储,如果不加转换的输出到外存,就是二进制文件。

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。

一个数据在内存中是怎么存储的呢?

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。

如有整数10000,如果以ASCII码的形式输出到磁盘,则磁盘中占用5个字节(每个字符一个字节),而二进制形式输出,则在磁盘上只占4个字节(VS2013测试)。



例如:

```
int main()
{
    int a = 10000;
    FILE* pf = fopen("test.txt", "wb");
    fwrite(&a, 4, 1, pf);//二进制的形式写到文件中
    fclose(pf);
    pf = NULL;
    return 0;
}
```

7.7文件读取结束的判定

7.7.1被错误使用的feof

牢记:在文件读取过程中,不能用feof函数的返回值直接用来判断文件的是否结束。

而是**应用于当文件读取结束的时候,判断是读取失败结束,还是遇到文件尾结束。**

1.文本文件读取是否结束,判断返回值是否为`EOF(fgetc)`,或者`NULL(fgets)`

例如:

- `fgetc`判断是否为`EOF`.
- `fgets`判断返回值是否为`NULL`.

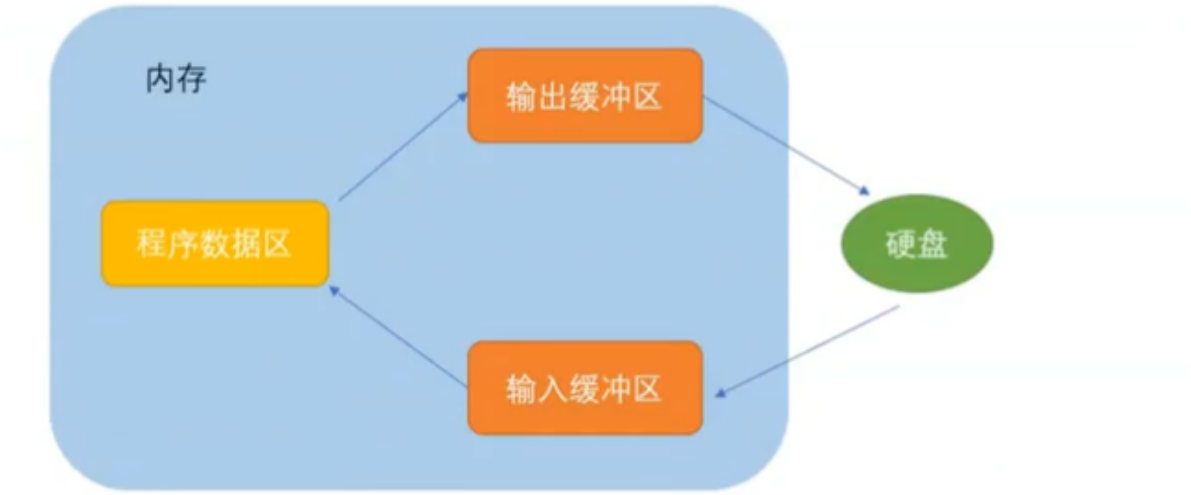
2.二进制文件的读取结束判断,判断返回值是否小于实际要读的个数。

例如:

- `fread`判断返回值是否小于实际要读的个数。

7.8文件缓冲区

ANSIC 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。



系统调用:操作系统提供的接口

这里可以得出一个结论:
因为有缓冲区的存在,C语言在操作文件的时候,需要做刷新缓冲区或者在文件操作结束的时候关闭文件。

如果不做,可能导致读写文件的问题。

8.程序环境和预处理

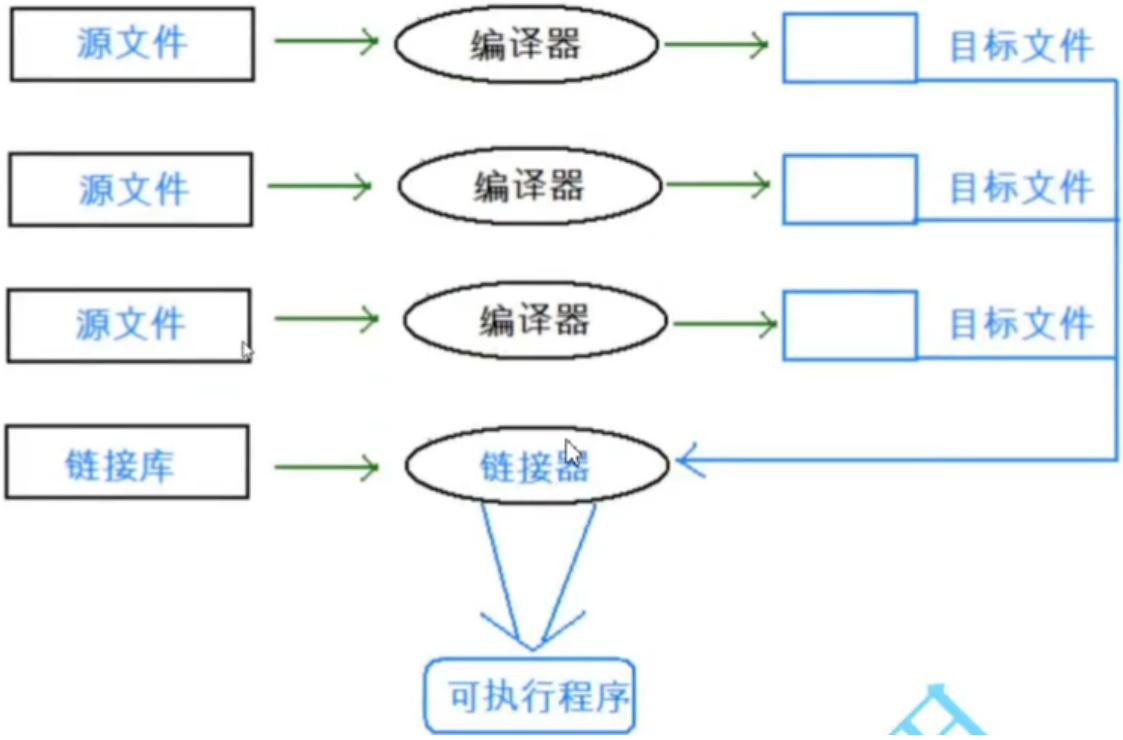
1.程序的翻译环境和执行环境

在ANSI C的任何一种实现中,存在两个不同的环境。

第1种是翻译环境,在这个环境中源代码被转换为可执行的机器指令。
第2种是执行环境,它用于实际执行代码。

2.详解编译+链接

2.1翻译环境



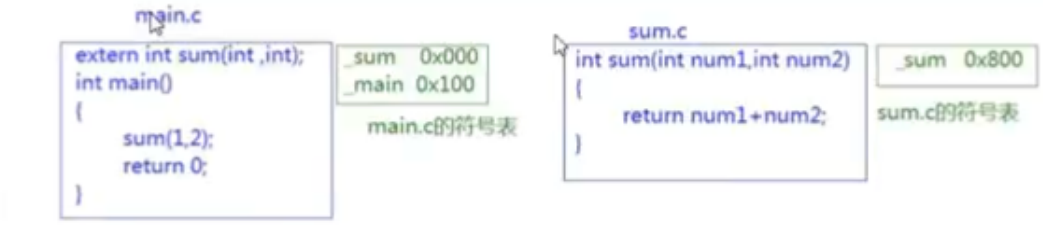
- 组成一个程序的每个源文件通过编译过程分别转换成目标代码（object code）。
- 每个目标文件由链接器（linker）捆绑在一起，形成一个单一而完整的可执行程序。
- 链接器同时也会引入标准C函数库中任何被该程序所用到的函数，而且它可以搜索程序员个人的程序库，将其需要的函数也链接到程序中。

2.2各项处理

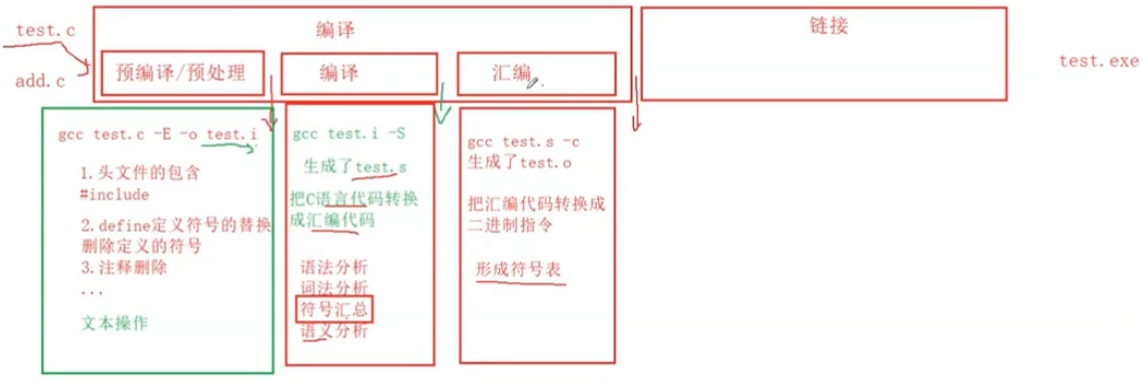
test.c

```c
#include <stdio.h>
int main()
{
    int i = 0;
    for(i=0; i<10; i++)
    {
        printf("%d ", i);
    }
    return 0;
}
```

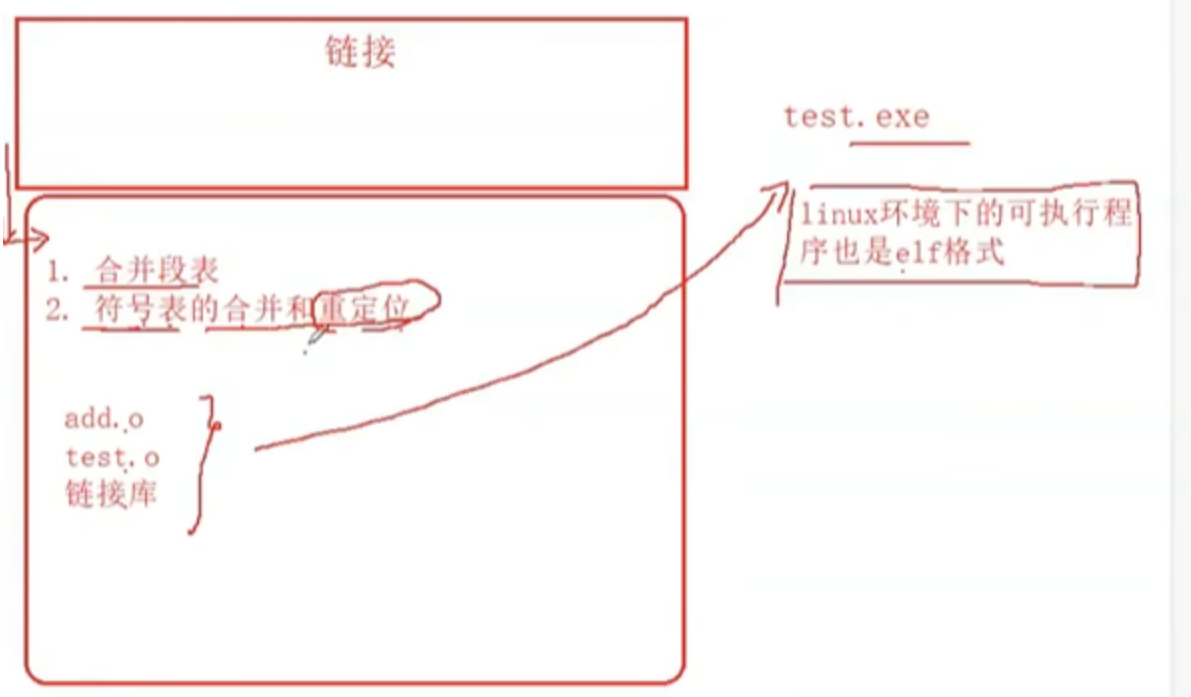
1. 预处理 选项 `gcc -E test.c -o test.i`
   预处理完成之后就停下来，预处理之后产生的结果都放在test.i文件中。
2. 编译 选项 `gcc -S test.c`
   编译完成之后就停下来，结果保存在test.s中。
3. 汇编 `gcc -c test.c`
   汇编完成之后就停下来，结果保存在test.o中。



2.3编译



2.4链接



3.程序的运行环境

1.程序必须载入内存中。在有操作系统的环境中:一般这个由操作系统完成。在独立的环境中,程序的载入必须由手工安排,也可能是通过可执行代码置入只读内存来完成。

2.程序的执行便开始。接着便调用main函数。

3.开始执行程序代码。这个时候程序将使用一个运行是堆栈(stack),存储函数的局部变量和返回地址。程序同时也可以使用静态(static)内存,存储于静态内存中的变量在程序的整个执行过程一直保留他们的值。

4.终止程序。正常终止main函数;也有可能是意外终止。

4.预处理详解

4.1 预定义符号

```
__FILE__   //进行编译的源文件
__LINE__   //文件当前的行号
__DATE__   //文件被编译的日期
__TIME__   //文件被编译的时间
__STDC__   //如果编译器遵循ANSI C，其值为1，否则未定义
```

这些预定义符号都是语言内置的

例:

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

int main()
{
    int i = 0;
    FILE* pf = fopen("log.txt", "w");
    if (pf = NULL)
    {
        perror("fopen");
        return 1;
    }
    for (i = 0; i < 10; i++)
    {
        fprintf("file:%s line=%d data=%s time:%s i=%d\n",__FILE__
,__LINE__,__DATE__,__TIME__,i);
    }
    fclose(pf);
    return 0;
}
```

4.2 #define

4.2.1 #define定义标识符

```
语法:
    #define name stuff
```

eg:

```c
#define MAX 1000
#define STR "hello bit"
int main()
{
    printf("%d\n", MAX);
    printf("%s\n", STR);
    return 0;
}
```

提问：#define后是否要加上;?

解答：建议不要加上；，这样容易导致问题

4.2.2 #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，这种实现通常被称为宏

下面时宏的申明方式：

`#define name( parament-list ) stuff`

其中的parament-list是一个由逗号隔开的符号表，它们可能出现在stuff中

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

如：

```c
#define SQUARE(X) X*X
```

在宏接收到一个参数x。

如果在上述声明之后，你把

```c
SQARE(5);
```

置于程序中，预处理器就会用下面这个表达式替换上面的表达式

```c
5 * 5
```

eg:

```c
#define SQUARE(x) x*x
int main()
{
    int r = SQUARE(5+1);
    //int r = 5 + 1 * 5 + 1;
    printf("%d\n", r);
    return 0;
}
```

4.2.3 #define的替换规则

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定义的变量。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。

4.2.4 #和##

如何把参数插入到字符串中?

首先我们看看这样的代码：

```
char* p = "hello ""bit\n";
printf("hello"," bit\n");
printf("%s", p);
```

这里输出的是不是 hello bit ?

答案是确定的：是。

我们发现字符串是有自动连接的特点的。

1. 那我们是不是可以写这样的代码？：

```
#define PRINT(FORMAT, VALUE)\
 printf("the value is "FORMAT"\n", VALUE);
...
PRINT("%d", 10);
```

这里只有当字符串作为宏参数的时候才可以把字符串放在字符串中。

1. 另外一个技巧是： 使用 # ，把一个宏参数变成对应的字符串。 比如：

```
int i = 10;
#define PRINT(FORMAT, VALUE)\
 printf("the value of " #VALUE "is "FORMAT "\n", VALUE);
...
PRINT("%d", i+3);//产生了什么效果?
```

代码中的 #VALUE 会预处理器处理为：

"VALUE" .

最终的输出的结果应该是：

the value of i+3 is 13

的作用

##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。

```
#define ADD_TO_SUM(num, value) \
 sum##num += value;
...
ADD_TO_SUM(5, 10);//作用是：给sum5增加10.
```

注：

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。

4.2.5 带副作用的宏参数

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。

例如：

```
x+1;//不带副作用
x++;//带有副作用
```

MAX宏可以证明具有副作用的参数所引起的问题。

```
#define MAX(a, b) ( (a) > (b) ? (a) : (b) )
...
x = 5;
y = 8;
z = MAX(x++, y++);
printf("x=%d y=%d z=%d\n", x, y, z);//输出的结果是什么？
```

这里我们得知道预处理器处理之后的结果是什么：

```
z = ( (x++) > (y++) ? (x++) : (y++));
```

所以输出的结果是：

```
x=6 y=10 z=9
```

4.2.6 宏与函数对比

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。

```
#define MAX(a, b) ((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。宏是类型无关的。

当然和宏相比函数也有劣势的地方：

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问题，导致程容易出现错。

宏有时候可以做函数做不到的事情。比如：宏的参数可以出现类型，但是函数做不到。

```
#define MALLOC(num, type)\
 (type *)malloc(num * sizeof(type))
...
//使用
MALLOC(10, int);//类型作为参数
//预处理器替换之后：
(int *)malloc(10 * sizeof(int));
```

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；每次使用这个函数时，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，所以相对慢一些
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些括号。	函数参数只在函数调用的时候求值一次，它的结果值传递给函数。表达式的求值结果更容易预测。
带有副作用的参数	参数可能被替换到宏体中的多个位置，所以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，结果更容易控制。
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，如果参数的类型不同，就需要不同的函数，即使他们执行的任务是不同的。
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

4.2.7 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： 把宏名全部大写 函数名不要全部大写

4.3 #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。
```

4.4 命令行定义

许多C 的编译器提供了一种能力，允许在命令行中定义符号。用于启动编译过程。 例如：当我们根据同一个源文件要编译出不同的一个程序的不同版本的时候，这个特性有点用处。（假 定某个程序中声明了一个某个长度的数组，如果机器内存有限，我们需要一个很小的数组，但是另外一 个机器内存大写，我们需要一个数组能够大写。）

```
#include <stdio.h>
int main()
{
    int array [ARRAY_SIZE];
    int i = 0;
    for(i = 0; i< ARRAY_SIZE; i ++)
   {
        array[i] = i;
   }
    for(i = 0; i< ARRAY_SIZE; i ++)
   {
        printf("%d " ,array[i]);
   }
    printf("\n" );
    return 0;
}
```

编译指令：

```
gcc -D ARRAY_SIZE=10 programe.c
```

4.5 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。

比如说：

调试性的代码，删除可惜，保留又碍事，所以我们可以选择性的编译

```
#include <stdio.h>
#define __DEBUG__
int main()
{
 int i = 0;
 int arr[10] = {0};
 for(i=0; i<10; i++)
 {
 arr[i] = i;
 #ifdef __DEBUG__
 printf("%d\n", arr[i]);//为了观察数组是否赋值成功。
 #endif //__DEBUG__
 }
 return 0;
}
```

常见的条件编译指令：

```
1.
#if 常量表达式
 //...
#endif
//常量表达式由预处理器求值。
如：
#define __DEBUG__ 1
#if __DEBUG__
 //..
#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
3.判断是否被定义
#if defined(symbol)
#ifdef symbol
#if !defined(symbol)
#ifndef symbol
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
#end if
```

4.6 头文件包含

我们已经知道， #include 指令可以使另外一个文件被编译。就像它实际出现于 #include 指令的地方 一样。

这种替换的方式很简单：

预处理器先删除这条指令，并用包含文件的内容替换。 这样一个源文件被包含10次，那就实际被编译10次。

4.6.1 头文件被包含的方式：

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。

Linux环境的标准头文件的路径：

```
/usr/include
```

VS环境的标准头文件的路径：

```
C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\include
```

注意按照自己的安装路径去找。

库文件包含

```
#include <filename.h>
```

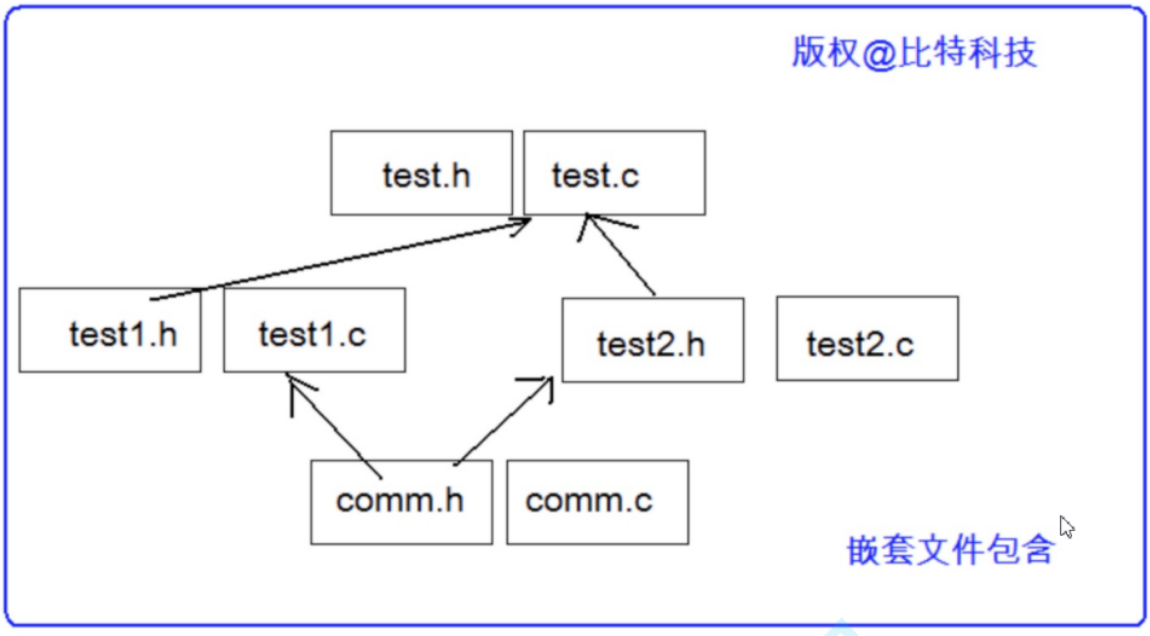
查找头文件直接去标准路径下去查找，如果找不到就提示编译错误。

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。

4.6.2 嵌套文件包含



comm.h和comm.c是公共模块。

test1.h和test1.c使用了公共模块。

test2.h和test2.c使用了公共模块。

test.h和test.c使用了test1模块和test2模块。

这样最终程序中就会出现两份comm.h的内容。这样就造成了文件内容的重复。

如何解决这个问题?

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   //__TEST_H__
```

或者:

```
#pragma once
```

就可以避免头文件的重复引入。

注:

`#include <stdio.h>`

<>:直接在提供的库里面去查找

`#include "test.h"`

"":1.先去代码所在的路径下查找

2.如果上面找不到，再去库目录下查找

4.6.3 其他预处理指令

```
#error
#pragma
#line
```