# 编码

二进制：0b

十进制：

八进制：

十六进制：0x

**补码：**

# 常量

常量可以使用const来修饰

## const

关键字const修饰的数据类型是指常类型，常类型的变量或对象的值时不能被更新的。

主要作用是：

1. 定义const常量，具有不可变性；
2. 便于进行类型检查，使编译器对处理内容有更多的了解，消除一些隐患；

void func(const int i){…}编译器就会知道i是一个常量，不允许修改

1. 可以避免意义模糊的数字出现，同样可以很方便地进行参数的调整和修改；
2. 可以保护被修饰的东西，放置意外修改，增强程序的健壮性；
3. 为函数重载提供了一个参考；
4. 可以节省空间，避免不必要的内存分配、。

**const和define的区别：**

1. const定义时，定义了常量的类型，更精确一些；
2. define只是简单的文本替换，除了可以定义常量外，还可以用来定义一些简单的函数，有点类似内联函数inline

**const指针**

int me;

const int \*p1 = &me;//p1可变，\*p1不可变，此时不能用\*p1修改，但是p1可以转向

const \*int p1 = &name;

# 变量

## 内存大小

## 初始化

未初始化的局部变量位于栈中，它的位置是不确定的，因此其值也是不确定的。当然，在windows下它的值是0xcccccccc，而“烫”字在MBCS字符集中的值为0xcccccccc，所以我们在Windows系统中经常会出现“烫烫烫”的错误信息。

而静态变量就不一样的，它的地址是确定的，并且存放在了数据段，而程序在运行之前，未初始化数据段的内容可以很方便地统一被初始化为0。

## volatile

一个被定义volatile的变量表示这个变量可能会被意想不到地改变，这样编译器就不会去假设这个变量的值了。换言之，优化器在用到这个变量的时候必须每次都小心地重新读取这个变量的值（即使两次操作在相隔的两行代码），而不是使用保存在寄存器里的备份（这涉及内存可见性）。

注：编译器会“自作聪明”的对代码进行优化，这可能导致结果不是我们想要的，所以要把这样可能随时改变的值设置为volatile类型，然后编译器就会跳过优化，每次都读取最新的数值。

**使用场景：**

1. 并行设备的硬件寄存器；
2. 一个中断服务子程序中会访问到非自动变量；
3. 多线程中被几个任务共享的变量，如互斥锁。

嵌入式程序员经常同硬件、RTOS、中断等打交道，这些都需要volatile变量，否则会带来灾难。

编译器的优化：在本次线程内，当读取一个变量时，为提高读取速度，编译器优化时有时会先把变量读取到一个寄存器中，以后，再读取该变量时，就直接从寄存器取值（读取寄存器的数据非常快）。当变量值在本线程中改变时，会同时把变量的新址copy到寄存器中，以便保持一致。

当变量在因别的线程改变了值，该寄存器的值不会相应改变，从而造成应用程序读取的值和实际的变量值不一致。当该寄存器在因别的线程等而改变了值，源变量的值不会改变，从而造成应用程序读取的值和实际的变量值不一致。

**注意：**

一个参数可以是const也可以是volatile：只读的状态寄存器，它是volatile因为它可能被意想不到地改变，它是const是因为程序不应该试图去修改它。

一个指针可以是volatile：当一个中断服务子程序修改一个指向buffer的指针时。

# 字符/字符串

C语言没有显式的字符串数据类型，因为字符串以字符串常量的形式（比如“abcd”）出现，或者存储于字符数组中（如char str[]={“abcd”}）。

字符串常量很适用于那些程序不会对它们进行修改的字符串，所以其他字符串都必须存储于字符数组或动态分配的内存中。

注：在C语言中没有字符串类型，在C++中有字符串类型，即string。

## 单引号/双引号

单引号引起的一个字符实际代表一个整数，整数值对应于该字符在编译器采用的字符集中的序列值。

双引号引起的字符串代表的是一个指向无名数组起始字符的指针，该数组被双引号之间的字符以及一个额外的二进制值为零的字符’\0’初始化。

## 类型转换

### stroll

### strtoull

### strtoul

## 字符长度

字符串是一串零个或者多个字符，并且以一个位模式位全0的NULL字节结尾。因此，字符串所含的字符内部不能出现NULL字节。

NULL字节是字符串的终止符，但它本身并不是字符串的一部分，所以字符串的长度并不包含NULL字节。

最常用的字符串函数都是“不受限制”的，就是说它们只是通过寻找字符串参数结尾的NULL字节来判断它的长度。这些函数一般都是指定一块内存用于存放结果字符串，在使用这些函数的时候，程序员必须保证结果字符串不会溢出这块内存。

### strlen

函数strlen返回一个size\_t类型的值，这个类型是在头文件stddef.h中定义的，它是一个无符号整数类型，在表达式中使用无符号数可能导致不可预知的结果。

例如：

if(strlen(x)>=strlen(y))……

if(strlen(x)-strlen(y)>=0)……

实际上是不相等的，第1条语句将按照预想的结果执行，第2条语句的结果将永远是真strlen的结果是个无符号数，所以操作符>=左边的表达式也将是无符号数，而无符号数绝对不可能是负的。

表达式中如果同时包含了有符号数和无符号数，可能产生奇怪的结果，原因一样：

if(strlen(x)>=10)……

if(strlen(x)-10>=0)……

注：如果把strlen的返回值强制转换为int，就可以消除这个问题。

### sizeof

## 字符查找

### strchr

### strrchr

### strpbrk

### strstr

### strspn

### strcspn

### strtok

## 字符拷贝

### strcpy

在执行字符串拷贝之前，需要保证dst指向的空间足够大，否则的话，可能会产生意想不到的结果。如果dst指向的空间不足以存储src中的字符串，不要以为只是遗失了src中存储不下的字符串那么简单。函数会将src中的内容拷贝到dst指向空间后面的内存中，而dst所指空间后面的内存是不可知的，有可能已经被其他资源占用了，这样就会破坏原先存储的内容，导致系统崩溃。

因为strcpy在执行字符串拷贝的时候，会从src所指位置开始，检测当前内存单元中存储的数据是否为’\0’，如果不为’\0’，则将这个内存单元中的数据拷贝到dst所指向的内存中。如果src中存储的字符串长度大于dst所申请的内存空间的话，就会产生越界，造成不可预知的后果。

**strcpy与snprintf：**

strcpy操作的对象是字符串，完成从源字符串到目的字符串的拷贝功能；

snprintf操作对象不限于字符串，虽然目的对象是字符串，但是源对象可以是字符串，也可以是任意基本类型的数据。这个函数主要用来实现字符串或基本数据类型到字符串的转换功能。如果源对象是字符串，并且指定%s格式符，也可实现字符串拷贝。

**strcpy与memcpy：**

memcpy内存拷贝，实现将一个内存块的内容复制到另外一个内存块的功能。内存块由其首地址以及长度确定，程序汇总出现的实体对象，不论是什么类型，最终表现就是在内存中占据一个内存区域或块。因此，memcpy的操作对象不局限于某一类型数据。memcpy通常用于同种类型数据或对象之间的拷贝。

总结：对于字符串拷贝，strcpy最合适，效率高且方便。

snprintf需要额外指定格式符并且进行格式转化，麻烦且效率不高。

memcpy高效，但需提供拷贝内存长度，易错且不便。如长度多大，会降低性能。

### strncpy

当src的长度大于dst申请的内存空间的时候，和strcpy不一样。这种情况下与strcpy不同，不会自动在字符串尾部添加结束标识符’\0’。

优点：更加安全，防止溢出

缺点：需要进行格式符设置，麻烦且不高效

### memcpy

### memmove

## 字符连接

### strcat

### strncat

## 字符比较

### strcmp

strcmp函数的字符串参数必须以一个NULL字节结尾，否则，strcmp就可能对参数后面的字节进行比较，这个比较结果不会有任何意义。

### strncmp

## 字符串

### 顺序存储

### 链式存储

# 数组

数组大小必须在**编译期**就作为一个常数确定下来，但C99中引入了变长数组，允许数组的维度是表达式，但在数组分配内存时，其表达式的值可以被求出。

数组下标运算实际上都是通过指针进行的，也就是说a[4]与\*(a+4)是等价的，甚至你会发现和4[a]也是一样的。

数组名一般代表了指向该数组下标为0的元素的指针，并且printf("%s\n",hello)与printf("%s\n",&hello[0])等效。

编译器并不检查程序对数组下标的引用是否在数组的合法范围之内，这种做法的好处在于：不需要浪费时间对一些已知是正确的数组下标进行检查，坏处在于：无效的下标引用无法被检测出来。

一个经验：如果下标是从那些一直是正确的值计算得来的，那么就无需检查下标，如果一个用作下标的值是根据某种方法下从用户输入的数据产生而来的，那么在使用之前必须进行检查，以确保在有效范围内。

## 初始化

对于数组的初始化可以采用memeset/bzero或者for循环+下标的方式。

对于在栈上初始化数组的操作：

**在数组较小的情况下，bzero的效率比memset高**；当数组超过一定大小之后，bzero的效率开始比memset低；数组越大，memset的性能优势越明显。而在数组较小的情况下，memset的性能甚至不如直接for循环对数组中的每一个字节置零的方法。

**经验：**在非必要的情况下，只对临时数组的第一个（或前几个）字符置零的初始化方式对比与使用bzero的话，能够明显提高性能。对于在栈上的数组，使用bzero和memset性能差别不大。

## 数组名

绝大多数情况，数组名都代表着指向该数组中下标为0的元素的指针，但是有例外：

int e[4];//假设int为4字节

sizeof(e);

上面的sizeof(e)的值并非4或8（指针占用空间），而是4\*4 = 16。也就是说，当数组名被用作运算符sizeof的参数时，它的计算结果是整个数组的大小，而非第一个元素的指针大小。

再来看下面这种情况：

int temp[5];

char \*p = &temp;

char \*q = temp;

在这里，p和q的值是一样的，含义却不一样，前者是指向数组的指针，而后者是指向该数组中下标为0的元素的指针。因此p+1指向了temp的末尾，而q+1指向了temp的第2个元素。

## 数组长度

int f[] = {1,2,3,4,5,6};

int \*g = f;

size\_t len\_f = sizeof(f)/sizeof(int)//正确计算方法

size\_t len\_g = sizeof(g)/sizeof(int)

上面的len\_f和len\_g的值相等吗？显然并不相等。事实上，只有len\_f得到了数组f的长度，而len\_g的值并没有任何实际意义（g表示指针，而\*g才表示数据，即数组f）。

拓展：int id[sizeof(unsigned long)];该写法正确，sizeof是编译时运算符，在编译期间就确定了，可以看作和机器相关的变量，但是int id[n]这种写法错误，n不确定。

## 数组访问

1. 数组名：

a[m]/\*(a+m)（不能使用\*(a++)）

1. 指针：

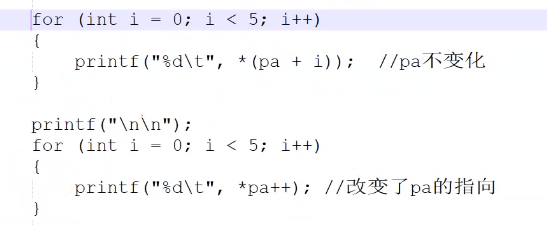
int \*（不能使用int(\*)[5]，int\*每次偏移4Byte，int(\*)[5]每次偏移4\*5Byte）

int a[5] = {1,1,1,1,1};

int \*pa = a;

//pa = &a[0];

此时还可以使用pa++;（不同于\*(a++)，这里相当于申请一个临时指针变量，数组名本身不改变）



## 数组与指针

**数组和指针不相等：**

考虑下面的声明：

int c[4]；//假设int占4字节

int \*d;

对于上面的声明，编译器会给c预留内存空间4\*4字节，并且数组名代表着指向数组第一个元素的指针。但对于d，却只为指针本身保留了内存空间。

所以此时有下面的操作：

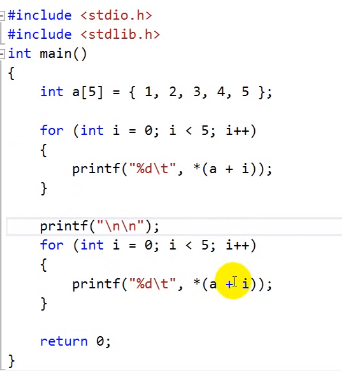
c[3]; //合法

\*(c+3); //合法

**\*d; //不合法，d指向了内存中不确定位置**

**c++； //不合法，一维数组名是指针常量，常量不能被修改掉**

d++； //可通过编译



注：**第一种写法\*(a+i)正确，后面的\*(a++)错误。因为a是一个特殊的指针，即数组名，a++这种表示改变了数组名的指向。**

**另外，下面的两种情况也是不一样的：**

char c[] = "hello";

char \*d = "hello";

前者对字符数组a进行了初始化，后者将d指向了字符串常量。字符串常量存储在只读区，因此有下面的操作：

c[0] = 'H'; //合法，可修改数组内容

\*d = 'H'; //不合法，字符串常量内容不可更改

d[0] = 'H' //不合法

**数组和指针可交换性的总结：**

1. 用a[i]这样的形式对数组进行访问总是被编译器“改写”或解释为像\*(a+1)这样的指针访问；
2. 指针始终是指针。它不可以改写为数组，你可以使用下标形式访问指针，一般都是指针作为函数参数时，而且你知道实际传输给函数的是一个数组；
3. 在特定的上下文中，也就是它作为函数的参数（也只有这种情况），一个数组的声明可以看做是一个指针。作为函数参数的数组始终会被编译器修改成

## 二维数组

如果是固定大小的数组，在C语言中用二维数组是最简单高效的：

const size\_t N = 100, M = 100;

int A[N][M];

对于动态的二维数组（矩形数组），C语言需要C99的可变长数组（VLA）才可以：

void f(size\_t N, size\_t M) {

int A[N][M];

/\* ... \*/

}

而且VLA必须为局部变量，生命周期为作用域之内。

若编译/运行环境不容许使用VLA，或者要控制生命周期的话，C语言并没有在语言层面提供动态二维数组的功能。一般做法是自行分配一块内存，使用时自行计算下标：

void f(size\_t N, size\_t M) {

int \*A = (int\*)malloc(N \* M \* sizeof(int));

int i, j;

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

for (j = 0; j < M; j++)

A[i \* M + j] = /\* ... \*/;

/\* ... \*/

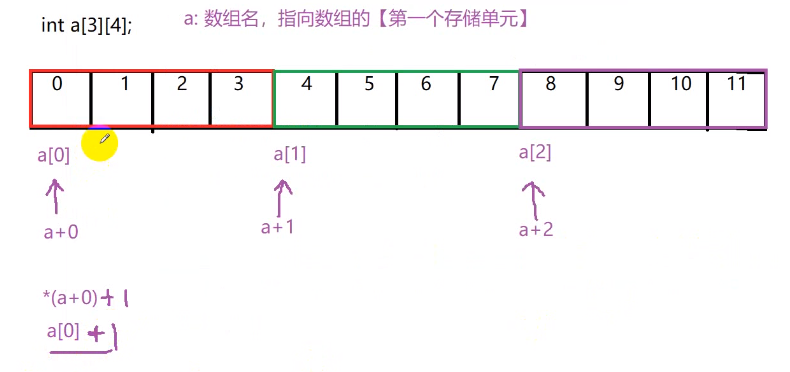
free(A); /\* 也可不释放，返回给调用方 \*/

}

如果不是矩形数组，而是锯齿形数组（jagged array），才会考虑使用二级指针（如int\*\* A）。用二级指针的好处是下标不用自行计算（可写成A[i][j]）。但对矩形数组也用二级指针的话，会有不必要的内存（存储指针的数组）和运行时开销（每次访问元素多一次寻址）。

另外，如果存储的数据是稀疏的（sparse），也会采用专门的数据结构。

二维数组在内存中存储：



二维数组仍然是按照顺序存储，即二维数组中是由多个一维数组组成的，分别对应a[0]，a[1]，a[2]。

a数字名表示数组的第一个存储单元，对于一维数组而言是第一个元素a[0]，但是对于二维数组而言，则表示第一个存储的一维数组，即a[0]。此时a+1偏移16字节（而不是4字节）。

a+0表示a[0]，\*(a+0)表示\*a[0]，即第一个一维数组，\*(a+0)+1等价于a[0]+1，表示第一个一维数组的第二个元素1。

二维数组的访问：

访问m行n列：\*(\*(a+m)+n)

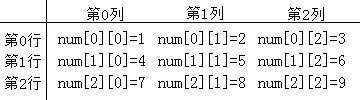
## 多维数组

多维数组的定义格式是：

数据类型 数组名称[常量表达式1][常量表达式2]...[常量表达式n];

例如：这样定义了一个名称为num，数据类型为int的二维数组。其中第一个[3]表示第一维下标的长度，就像购物时分类存放的购物；第二个[3]表示第二维下标的长度，就像每个购物袋中的元素。

我们可以把上面的数组看作一个3×3的矩阵，如下图：



多维数组的初始化与一维数组的初始化类似也是分两种：

1、数据类型 数组名称[常量表达式1][常量表达式2]...[常量表达式n] = {{值1,..,值n},{值1,..,值n},...,{值1,..,值n}};

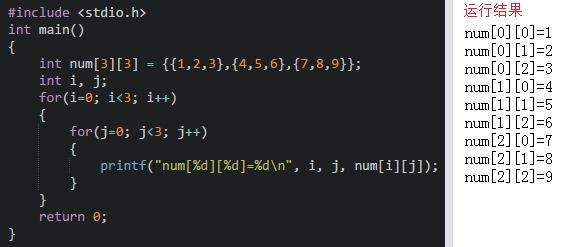
2、数据类型 数组名称[常量表达式1][常量表达式2]...[常量表达式n]; 数组名称[下标1][下标2]...[下标n] = 值;

多维数组初始化要注意以下事项：

1、采用第一种始化时数组声明必须指定列的维数。因为系统会根据数组中元素的总个数来分配空间，当知道元素总个数以及列的维数后，会直接计算出行的维数；

2、采用第二种初始化时数组声明必须同时指定行和列的维数。

多维数组也是存在遍历的，和一维数组遍历一样，也是需要用到循环。不一样的就是多维数组需要采用嵌套循环，如：遍历输出int num[3][3] = {{1,2,3},{4,5,6},{7,8,9}};



注意：多维数组的每一维下标均不能越界

# 链表

## 静态链表

静态链表中所有节点都是在内存中临时生成的，不是new开辟的空间。



## 动态链表



# 结构体

为了表示一些复杂的数据，而普通的基本类型变量无法满足要求。

结构体是用户根据实际需要自己定义的复合数据类型（不是变量）。

结构体的内容可以是任何其他的数据声明：单个数据项、数组、其他结构体、指针、枚举、联合体等。

注：结构体可以放置枚举，枚举中也可以放置结构体。

## 定义

**方法一：**

struct str{

//定义

};//分号不能省略

**方法二：**

还可以在struct关键字后面加一个可选的“结构标签”：

struct str{

//定义

}str\_t;

因此，结构体的通常形式是：

struct 结构标签（可选）{

类型1 标识1;

}变量定义（可选）;

**定义三：**

struct str{

//内容

};

typedef struct str mystr; //定义结构体的一个实例

typedef mystr \*name; //起一个别名，这样就隐藏了结构体这个数据类型

**方法四：**

将前面的都合并到一个表达式中（这是常用的方式）：

typedef struct str{

//内容

}str\_t;

str\_t mystr;

注：typedef是为已有的数据类型起一个名字，比如typedef INT16 int;类似对于结构体struct str{}；可以采用typedef struct str{}str\_t;或者typedef struct str{}\*str\_t;定义一个别名str\_t或者指针\*str\_t。

## 初始化

## 访问

# 联合体

## 定义

联合的所有成员引用的是内存中的相同位置，当你想在不同的时刻把不同的东西存储于同一位置时，就可以使用联合体。

## 初始化

联合变量也可以被初始化，但这个初始值必须是联合体第1个成员的类型，并且它必须位于一对花括号里面。

union{

int a;

char b[2];

}x={5}; //把x.a初始化为5

或者单独赋值：

x.a = 1;

x.c=’a’;

或者一起赋值：

x={1,’a’};

## 访问

# 字节对齐

参考：https://blog.csdn.net/weixin\_41066529/article/details/89496593

1、数据成员对齐规则：结构(struct)(或联合(union))的数据成员，第一个数据成员放在offset为0的地方，以后每个数据成员存储的起始位置要从该成员大小或者成员的子成员大小（只要该成员有子成员，比如说是数组，结构体等）的整数倍开始(比如int在32位机为4字节，则要从4的整数倍地址开始存储。

2、结构体作为成员：如果一个结构里有某些结构体成员，则结构体成员要从其内部最大元素大小的整数倍地址开始存储(struct a里存有struct b，b里有char，int，double等元素，那b应该从8的整数倍开始存储)。

3、收尾工作：结构体的总大小，也就是sizeof的结果，必须是其内部最大成员的整数倍，不足的要补齐。

# 枚举

## 定义

枚举enum是通过一种简单的途径，把一串名字与一串整型值联系在一起。

枚举一般形式：

enum 可选标签{

//内容

}可选变量定义;

注：其中内容是一些标识符的列表，可能有一些整型值赋值给它们。

## 初始化

enum size{

small = 1;

large = 10000;

};

缺省情况下，整型值从0开始。如果对列表中某个标识符进行赋值，那么紧接其后的那个标识符的值就比所赋值大1，以此类推。

枚举与define：

1. define定义的名字一般在编译时被丢弃；
2. 枚举名字则通常一直在调试器中可见（gdb调试的时候可以打印输出枚举的名字），可以在调试代码的时候使用它们。

# 预处理器

## 预定义符号

\_\_FILE\_\_ 进行编译的源文件名

\_\_LINE\_\_ 文件当前行的行号

\_\_DATA\_\_ 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

## #define

优点：

1. 只需要替换一个地方就可以完成全部调用的替换；
2. 省去函数调用的重大系统开销（在编译前就已经做好替换了，比如数据合法性判断和log打印控制都可以采用宏定义）

注：宏只是对程序的文本起作用

**宏定义中的空格**

一个宏如果不带参数，则只需要使用宏名即可，括号无关紧要。第一个空格前的作为待替换的部分，空格厚的是替换的内容。

**宏不是函数**

注意宏中的括号，作用是预防引起与优先级相关的问题，所以最好是多加一些括号，保证优先级是我们需要的。

**宏不是语句**

## typedef

## 条件编译

## #error

## #progma

#progma指令用于因编译器而异的特性。它的语法也是因编译器而异，有些环境可能提供一些#progma指令，允许一些编译选项或者其他任何方式无法实现的一些处理方式。

例如，有些编译器使用#progma指令在编译过程中打开或关闭清单显示，或者把汇编代码插入到C语言中。从本质上讲，#progma是不可移植的。

预处理器忽略它不认识的#progma，两个不同的编译器可能以两种不同的方式解释同一条#progma指令。

# 编译链接



## 静态链接

任何人都可以创建静态或动态的函数库。只需要简单地编译一些不包含main函数的代码，并把编译生成的.o文件用正确的使用工具进行处理——如果是静态库，使用“ar”，如果是动态库，使用“ld”。

静态库被称为archive，它们通过ar（用于archive的实用工具）来创建和更新，静态库约定在它们的文件名中使用“.a”的拓展名。

动态链接库中链接编辑器ld创建，根据约定，动态库的文件拓展名为“.so”，表示“shared object（共享对象）”——每一个链接到该函数库的程序都共享它的一份拷贝，而静态链接相反，每个对象都拥有一份该函数库内容的拷贝。

## 动态链接

动态链接是一种“Just-in-time（JIT）”链接，这意味着程序在运行时必须能够找到它们所需要的函数库。链接器通过库文件名或者路径名植入可执行文件中来做到这一点。这意味着，函数库的路径不能随意移动。

当在一台机器上编译完程序后，把它拿到另一台不同的机器上运行时，也可能出现运行失败的情况。执行程序的机器必须具有所有该程序需要链接的函数库，而且这些函数库必须位于在链接器中所说明的目录。

使用共享函数库的主要原因就是获得ABI的好处——使你的软件不必因为新版本函数库或操作系统的发布而重新链接。附带的一个好处是，能够提高系统的总体性能。

动态链接使单个可执行文件的启动速度稍微受影响，但是动态链接可以从两个方面提升性能：

1. 动态链接可执行文件比功能相同的静态链接可执行文件的体积小，可以节省磁盘空间和虚拟内存，因为函数库只有在需要时才会被映射到进程中；
2. 所有动态链接到某个特定函数库的可执行文件在文件在运行时共享该函数库的一个单独拷贝。如果可执行文件是静态链接的，每个文件都将拥有一份函数库的拷贝，虽然极为浪费。

# 作用域

作用于只与变量或者函数的位置有关，没有什么关键字修改作用域的。

# 链接属性

链接属性有3种：external外部、internal内部和none

## external

全局可见，即其他文件都可以链接到这个文件中的函数。

使用关键字extern。

函数的声明extern关键词是可有可无的，因为函数本身不加修饰的话就是extern。但是引用的时候一样需要声明的。

全局变量在外部使用声明时，extern关键字是必须的，如果变量没有extern修饰且没有显式的初始化，同样成为变量的定义，因此此时必须加extern，而编译器在此标记存储空间在执行时加载内并初始化为0。而局部变量的声明不能有extern的修饰，且局部变量在运行时才在堆栈部分分配内存。

C++支持重载，而C不支持，C++并不能直接调用C代码写好的接口，因此**如果你的C代码想要能够被C调用，也想被C++调用，那么别忘了extern "C"**。

## static

在本函数或者本文件内有效。

关键字external和static用于在声明中修改标识符的链接属性，如果某个声明在正常情况下具有external链接属性，在其前面加上static就变为了internal。

# 存储类型

在定义变量的时候我们通常使用的是局部变量、全局变量，但是局部变量作用范围仅限函数或者结构内部，且执行完毕生命周期结束，全局变量虽然生命周期伴随程序，但是多线程中并不安全。所以，还需要了解变量的不同存储类型和作用域（生命周期）控制方法。

有三个地方可以用于存储变量：普通内存、运行时堆栈、硬件寄存器。

变量的缺省存储类型取决于它的声明位置。凡是在代码声明的变量总是存储于静态内存中，全局变量或者static修改的局部变量属于这一类。

## auto

这是C语言默认的存储类别。自动变量（auto variable）声明在利用该变量的函数中，而声明时所用的关键字正式auto。这种变量会在函数被调用时自动创建，并于函数退出后自动销毁。因此，对于声明该变量的函数而言，它是个私有变量，或者叫做局部变量。

一般而言，main函数中的局部变量在整个在整个程序运行期间都是正常存在的，只不过其活跃范围仅限于main罢了。如果递归地调用同一个函数，那么每次调用时所创建的auto变量都是不相同的（因此是局部的，下一次调用前上一次的变量已经销毁）。此外，还可以在代码块（block）中定义自动变量，这些变量只在声明该变量的代码块里有效。

## extern

这种变量声明于所有函数之外。在整个程序运行期间它们始终存在，并保持活跃。这种变量也称作全局变量（global variable），其默认值为0。

extern变量与局部变量不同，程序里的每个函数都可以访问它。如果局部变量与全局变量同名，那么采用局部变量的声明。

有时我们用extern关键字来声明这种变量，该变量从声明它的地方开始，一直到程序末尾，均保持可见。

## 静态内存区

static修饰的变量，全局变量

静态变量的值会一直保留到程序结束为止。它是用static关键字来声明的。

这种变量可能是外部变量，也可能是局部变量，具体情况要根据声明的地点来确定。静态变量只在程序编译的时候初始化一次。

### 内部静态变量

内部静态变量是指声明在函数里的静态变量。这种变量的作用域一直延续到定义它的那个程序结束为止。此类变量与auto变量几乎完全相同，只不过它们在函数执行完之后依然存在（或者说存活），直到

### 外部静态变量

### 静态函数

**硬件寄存器**

register类型的变量存储在计算机的寄存器中。由于访问寄存器要比访问内存快，因此，把频繁用到的变量放在寄存器里，可以提高程序的执行速度。

因为寄存器只能容纳很少的几个变量，所以我们使用必须谨慎：究竟应该把哪几个变量放在寄存器中。另一方面，如果寄存器满额，那么C 语言会自动把register形式的变量转换为非register的形式。

不要将全局变量声明为register，因为那样做有可能会使该变量在程序运行的过程中始终占据着寄存器。

## 堆栈

**栈溢出**

栈溢出指的是程序向栈中某个变量中写入的字节数超过了这个变量本身所申请的字节数，因而导致栈中与其相邻的变量的值被改变。

栈溢出的原因：

1、局部数组过大。当函数内部的数组过大时，有可能导致堆栈溢出。局部变量是存储在栈中的，因此这个很好理解。解决这类问题的办法有两个，一是增大栈空间,二是改用动态分配，使用堆（heap）而不是栈（stack）。

2、递归调用层次太多。递归函数在运行时会执行压栈操作，当压栈次数太多时，也会导致堆栈溢出。

3、指针或数组越界。这种情况最常见，例如进行字符串拷贝，或处理用户输入等等。

栈溢出例子：

#include <stdio.h>

#include <string.h>

int main(int argc, char\* argv[]) {

char buf[256];

strcpy(buf,argv[1]);

printf("Input:%s\n",buf);

return 0;

}

上述代码中的strcpy(buf,argv[1]);这一行发生了缓冲区溢出错误，因为源缓冲区内容是用户输入的。

# 内存分配

## malloc/free

malloc需要手动初始化（或者调用calloc），参数是需要分配的字节数，分配的是一块连续的内存（适合数组存储以及指针的偏移）。当一块以前使用的内存不再使用时，调用free函数手动把它释放，归还给内存池供以后用。

返回指向被分配内存块起始位置的指针，void类型的指针（这样可以存储任何类型的数据），有些编译器，尤其是老式的编译器，可能要求你在转换时使用强制类型转换。

操作系统无法向malloc提供更多的内存，malloc就返回一个NULL指针。因此，每个从malloc返回的指针都进行检查，确保它并非NULL是非常重要的！

动态分配的内存必须整块一起释放，不允许释放一块内存的一部分，但是realloc函数可以缩小一块内存分配的内存，有效地释放它尾部的部分内存。

free参数必须要么是NULL，要么是malloc、calloc、realloc返回的值，向free传递一个NULL参数不会产生任何效果。

传递给free一个指针，让它释放一块并非动态分配的内存可能导致程序立即终止或者在晚些的时候终止。

注意：

1. 不要访问已经被free函数释放的内存（如果存在多个指针复制的时候，有可能会忘记已经释放了）；
2. 确保程序中所有使用这块内存的地方在这块内存被释放之前停止对它的使用。

## calloc/relloc

calloc在返回指向内存的指针前把它初始化为0，calloc的参数不是malloc那样的字节数，而是所需元素的数量和每个元素的字节数，根据这些就能计算出所有的待分配的内存。

realloc用于修改一个原先已经分配大小的内存块。增大内存时元数据仍然有效，减小时尾部数据被拿掉。如果原来的内存块大小无法改变，realloc将分配另一块正确大小的内存，并将原来那块内存的内容复制到新的块上。因此，在使用realloc之后，你就不能再使用指向旧内存的指针，而是应该用reallloc返回的新指针。

## memset/bzero

memset是对内存进行清零，与bzero类似，在数组较小的情况下，bzero的性能比memset高。当数组超过一定大小后，bzero效率开始比memset低，数组越大，memset优越越明显。在数组较小的情况下，memset性能设置不如直接for循环数组中每一个字节置零的方法。

关于字符数组的初始化，在项目的压力测试中，发现性能明显下降，变怀疑在程序中的若干临时字符数组的初始化（使用bzero）身上。于是修改为首个字符置零的方式而非全部置零的方式初始化，响应得到明显的提升。原来在mp3检索的每一条结果都要进行bzero对临时数组初始化，每一个请求需要30次的bzero对临时数组的置零。于是想到了，在非必要的情况下，只对临时数组的第一个（或前几个）字符置零的初始化方式对比与使用bzero的话，能够明显提高性能。

在此之外，又想起另外两种对数组所有字节都置零的方式，顺便比较一下他们之间的性能，写个简单的程序如下：

#include <stdio.h>

#include <sys/time.h>

#include <string.h>

#define TIMEDIFF(s, e) (((e.tv\_sec)-(s.tv\_sec))\*1000000 + (e.tv\_usec) - (s.tv\_usec))

int main()

{

struct timeval s, e;

char a[1024], b[1024\*1024], c[1024\*1024\*4];

gettimeofday(&s, NULL);

bzero(a, sizeof(a));

gettimeofday(&e, NULL);

printf("bzero 1k: %d\n", TIMEDIFF(s, e));

gettimeofday(&s, NULL);

bzero(b, sizeof(b));

gettimeofday(&e, NULL);

printf("bzero 1m: %d\n", TIMEDIFF(s, e));

gettimeofday(&s, NULL);

bzero(c, sizeof(c));

gettimeofday(&e, NULL);

printf("bzero 4M: %d\n", TIMEDIFF(s, e));

gettimeofday(&s, NULL);

memset(a, 0, sizeof(a));

gettimeofday(&e, NULL);

printf("memset 1k: %d\n", TIMEDIFF(s, e));

gettimeofday(&s, NULL);

memset(b, 0, sizeof(b));

gettimeofday(&e, NULL);

printf("memset 1M: %d\n", TIMEDIFF(s, e));

gettimeofday(&s, NULL);

memset(c, 0, sizeof(c));

gettimeofday(&e, NULL);

printf("memset 4M: %d\n", TIMEDIFF(s, e));

gettimeofday(&s, NULL);

for(int i=0; i<sizeof(a); ++i)

a[i]=0;

gettimeofday(&e, NULL);

printf("for 1k: %d\n", TIMEDIFF(s, e));

gettimeofday(&s, NULL);

for(int i=0; i<sizeof(b); ++i)

b[i]=0;

gettimeofday(&e, NULL);

printf("for 1M: %d\n", TIMEDIFF(s, e));

gettimeofday(&s, NULL);

memset(c, 0, sizeof(c));

gettimeofday(&e, NULL);

printf("memset 4M: %d\n", TIMEDIFF(s, e));

gettimeofday(&s, NULL);

for(int i=0; i<sizeof(a); ++i)

a[i]=0;

gettimeofday(&e, NULL);

printf("for 1k: %d\n", TIMEDIFF(s, e));

gettimeofday(&s, NULL);

for(int i=0; i<sizeof(b); ++i)

b[i]=0;

gettimeofday(&e, NULL);

printf("for 1M: %d\n", TIMEDIFF(s, e));

gettimeofday(&s, NULL);

for(int i=0; i<sizeof(c); ++i)

c[i]=0;

gettimeofday(&e, NULL);

printf("for 4M: %d\n", TIMEDIFF(s, e));

}

运行的结果基本上是，在数组较小的情况下，bzero的效率比memset高；当数组超过一定大小之后，bzero的效率开始比memset低；数组越大，memset的性能优势越明显。而在数组较小的情况下，memset的性能甚至不如直接for循环对数组中的每一个字节置零的方法。

以下的运行结果的数值单位是微秒（gettimeofday的默认单位）。

第一次运行：

bzero 1k: 6

bzero 1m: 2168

bzero 4M: 9136

memset 1k: 11

memset 1M: 1303

memset 4M: 5483

for 1k: 12

for 1M: 4934

for 4M: 21313

再一次运行：

bzero 1k: 6

bzero 1m: 2160

bzero 4M: 9067

memset 1k: 17

memset 1M: 1257

memset 4M: 5115

for 1k: 11

for 1M: 4866

for 4M: 19201

此后，又写了个小程序，测试在堆上的数组中，bzero和memset的效率，发现两者差不多。可能由于，里面原来的数据就比较有规则，不管是否先对数组置一随机值。（malloc开辟字符数组空间时，会清零的。）

#include <stdio.h>

#include <string.h>

#include <sys/time.h>

#include <stdlib.h>

#include <time.h>

#define TIMEDIFF(s, e) (((e.tv\_sec)-(s.tv\_sec))\*1000000 + (e.tv\_usec) - (s.tv\_usec))

int main()

{

srand(time(NULL));

char \*array;

struct timeval s, e;

int tb, tm;

for(int i=1; i<1024\*1024\*1024; i\*=2)

{

array=(char\*)malloc(i);

memset(array, rand()%256, i);

gettimeofday(&s, NULL);

bzero(array, i);

gettimeofday(&e, NULL);

tb=TIMEDIFF(s, e);

free(array);

array=(char\*)malloc(i);

memset(array, rand()%256, i);

gettimeofday(&s, NULL);

memset(array, 0, i);

gettimeofday(&e, NULL);

tm=TIMEDIFF(s, e);

free(array);

printf("array size: %d \tbzero time: %d \tmemset time: %d \tbzero>memset?: %d\n", i, tb, tm, (tb>tm));

}

}

运行结果：

array size: 1 bzero time: 28 memset time: 1 bzero>memset?: 1

array size: 2 bzero time: 1 memset time: 1 bzero>memset?: 0

array size: 4 bzero time: 1 memset time: 1 bzero>memset?: 0

array size: 8 bzero time: 1 memset time: 1 bzero>memset?: 0

array size: 16 bzero time: 1 memset time: 1 bzero>memset?: 0

array size: 32 bzero time: 1 memset time: 1 bzero>memset?: 0

array size: 64 bzero time: 1 memset time: 0 bzero>memset?: 1

array size: 128 bzero time: 1 memset time: 1 bzero>memset?: 0

array size: 256 bzero time: 1 memset time: 1 bzero>memset?: 0

array size: 512 bzero time: 1 memset time: 1 bzero>memset?: 0

array size: 1024 bzero time: 1 memset time: 1 bzero>memset?: 0

array size: 2048 bzero time: 1 memset time: 1 bzero>memset?: 0

array size: 4096 bzero time: 2 memset time: 2 bzero>memset?: 0

array size: 8192 bzero time: 2 memset time: 2 bzero>memset?: 0

array size: 16384 bzero time: 5 memset time: 6 bzero>memset?: 0

array size: 32768 bzero time: 9 memset time: 8 bzero>memset?: 1

array size: 65536 bzero time: 27 memset time: 24 bzero>memset?: 1

array size: 131072 bzero time: 81 memset time: 68 bzero>memset?: 1

array size: 262144 bzero time: 190 memset time: 169 bzero>memset?: 1

array size: 524288 bzero time: 447 memset time: 393 bzero>memset?: 1

array size: 1048576 bzero time: 996 memset time: 973 bzero>memset?: 1

array size: 2097152 bzero time: 2258 memset time: 2272 bzero>memset?: 0

array size: 4194304 bzero time: 4821 memset time: 4799 bzero>memset?: 1

array size: 8388608 bzero time: 9797 memset time: 9799 bzero>memset?: 0

array size: 16777216 bzero time: 19764 memset time: 19737 bzero>memset?: 1

array size: 33554432 bzero time: 39687 memset time: 39675 bzero>memset?: 1

array size: 67108864 bzero time: 79907 memset time: 79324 bzero>memset?: 1

array size: 134217728 bzero time: 158956 memset time: 158775 bzero>memset?: 1

array size: 268435456 bzero time: 318247 memset time: 318632 bzero>memset?: 0

array size: 536870912 bzero time: 638536 memset time: 638883 bzero>memset?: 0

## memcpy

## mememove

## memcmp

## memchr

# 指针

指针只是一个指向的作用，最终落地的还是具体的数据类型，比如int、char等等。

## 普通指针

一般在使用结构体的时候，声明的是指针类型的变量，如果是一个基本类型，则直接声明普通变量即可。

## 双指针

例子：

i = ‘a’;

\*pi = ‘a’;

\*\*pi = ‘a’;

在一条简单的对i赋值的操作就可以完成任务的情况下，为什么还要用更加复杂的涉及间接访问的方法呢？

这是因为赋值操作并不总是可行，例如链表的插入。在那些函数中，我们无法使用简单赋值，因为变量名在函数的作用域内部是未知的。函数拥有的只是一个指向需要修改的内存位置的指针，所以要对该指针进行间接访问操作以访问需要修改的变量。

## 函数指针

### 概述

对于函数指针来说，它存储的就是函数的地址。函数也是有地址的，函数实际上由载入内存的一些指令组成，而指向函数的指针存储了函数指令的起始地址。

函数名是常量指针，不可以被赋值（func=func1非法），可以使用函数指针操作。

### 声明

声明普通类型指针时，需要指明指针所指向的数据类型，而声明函数指针时，也就要指明指针所指向的函数类型，即需要指明函数的返回类型和形参类型。例如对于下面的函数原型：

int sum(int,int);

它是一个返回值为int类型，参数是两个int类型的函数，那么如何声明该类型函数的指针呢？很简单，将函数名替换成(\*pf)形式即可，即我们把sum替换成(\*fp)即可，fp为函数指针名，结果如下：

int （\*fp）(int,int);

这样就声明了和sum函数类型相同的函数指针fp。这里说明两点，第一，\*和fp为一体，说明了fp为指针类型，第二，\*fp需要用括号括起来，否则就会变成下面的情况：

int \*fp(int,int);

这种情况下，它声明了一个参数为两个int类型，返回值为int类型的指针的函数，而不再是一个函数指针了。

在经常使用函数指针之后，我们很快就会发现，每次声明函数指针都要带上长长的形参和返回值，非常不便。这个时候，我们应该想到使用typedef，即为某类型的函数指针起一个别名，使用起来就方便许多了。例如，对于前面提到的函数可以使用下面的方式声明：

typedef int (\*myFun)(int,int);//为该函数指针类型起一个新的名字

myFun f1; //声明myFun类型的函数指针f1

上面的myFun就是一个函数指针类型，在其他地方就可以很方便地用来声明变量了。typedef的使用不在本文的讨论范围，但是特别强调一句，typedef中声明的类型在变量名的位置出现，理解了这一句，也就很容易使用typedef了。因而下面的方式是错误的：

typedef myFun (int)(int,int); //错误

typedef (int)(int,int) \*myFun; //错误

### 赋值

赋值也很简单，既然是指针，将对应指针类型赋给它既可。例如：

#include<stdio.h>

int test(int a,int b)

{

/\*do something\*/

return 0

}

typedef int(\*fp)(int,int);

int main(void)

{

fp f1 = test; //表达式1

fp f2 = &test;//表达式2

printf("%p\n",f1);

printf("%p\n",f2);

return 0;

}

在这里，声明了返回类型为int，接受两个int类型参数的函数指针f1和f2，分别给它们进行了赋值。表达式1和表达式2在作用上并没有什么区别。因为函数名在被使用时总是由编译器把它转换为函数指针，而前面加上&不过显式的说明了这一点罢了。

### 调用

调用也很容易，把它看成一个普通的函数名即可：

#include<stdio.h>

int test(int a,int b)

{

/\*do something\*/

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

return 0

}

typedef int(\*fp)(int,int);

int main(void)

{

fp f = test;

f(1,2);//表达式1

(\*f)(3,4);//表达式2

return 0;

}

在函数指针后面加括号，并传入参数即可调用，其中表达式1和表达式2似乎都可以成功调用，但是哪个是正确的呢？ANSI C认为这两种形式等价。

### 作用

函数指针的应用场景比较多，以库函数qsort排序函数为例，它的原型如下：

void qsort(void \*base,size\_t nmemb,size\_t size , int(\*compar)(const void \*,const void \*));

看起来很复杂对不对？拆开来看如下：

void qsort(void \*base, size\_t nmemb, size\_t size, );

拿掉第四个参数后，很容易理解，它是一个无返回值的函数，接受4个参数，第一个是void\*类型，代表原始数组，第二个是size\_t类型，代表数据数量，第三个是size\_t类型，代表单个数据占用空间大小，而第四个参数是函数指针。这第四个参数，即函数指针指向的是什么类型呢？

int(\*compar)(const void \*,const void \*)

很显然，这是一个接受两个const void\*类型入参，返回值为int的函数指针。

到这里也就很清楚了。这个参数告诉qsort，应该使用哪个函数来比较元素，即只要我们告诉qsort比较大小的规则，它就可以帮我们对任意数据类型的数组进行排序。

在这里函数指针作为了参数，而他同样可以作为返回值，创建数组，作为结构体成员变量等等，

### 总结

函数指针与其他指针类型无本质差异，不过它指向的是函数的地址罢了。

声明函数指针需要指明函数的返回类型和形参类型。

函数名在被使用时总是由编译器把它转换为函数指针。

要想声明函数指针，只需写出函数原型，然后将函数名用(\*fp)代替即可。这里fp是声明的函数指针变量。

typedef中声明的类型在变量名的位置出现。

## 指针引用

# 字节对齐

## 概述

计算机中内存大小的基本单位是字节（byte），理论上来讲，可以从任意地址访问某种基本数据类型，但是实际上，计算机并非逐字节大小读写内存，而是以2,4,或8的 倍数的字节块来读写内存，如此一来就会对基本数据类型的合法地址作出一些限制，即它的地址必须是2，4或8的倍数。那么就要求各种数据类型按照一定的规则在空间上排列，这就是对齐。

## 准则

总的来说，字节对齐有以下准则：

结构体变量的首地址能够被其对齐字节数大小所整除。

结构体每个成员相对结构体首地址的偏移都是成员大小的整数倍，如不满足，对前一个成员填充字节以满足。

结构体的总大小为结构体对齐字节数大小的整数倍，如不满足，最后填充字节以满足。

我们通过一个小例子来说明是如何对齐的。

考虑下面的程序

/\*================================================================

\* Copyright (C) 2018 Ltd. All rights reserved.

\*

\* 文件名称：testByteAlign.c

\* 创 建 者：shouwang

\* 创建日期：2018年09月15日

\* 描 述：

\*

================================================================\*/

#include<stdio.h>

#include<stdint.h>

struct test

{

int a;

char b;

int c;

short d;

};

int main(int argc,char \*argv)

{

/\*在32位和64位的机器上，size\_t的大小不同\*/

printf("the size of struct test is %zu\n",sizeof(struct test));

return 0;

}

编译成32位程序并运行（默认四字节自然对齐），可以看到，结构体test 的大小为16字节，而不是11字节（a占4字节，b占1字节，c占4字节，d占2字节）

#64位机器上编译32位程序可能需要安装一个库

#sudo apt-get install gcc-multilib

gcc -m32 -o testByteAlign testByteAlign.c #编译程序

chmod +x testByteAlign #赋执行权限

./testByteAlign #运行

the size of struct test is 16

实际上，结构体test的成员在内存中可能是像下面这样分布的（数值为偏移量）

未对齐时:

0~3 4 5~9 10~11

a b c d

对齐时:

0~3 4 5~7 8~11 12~13 14~15

a b 填充内容 c d 填充内容

从上面可以看出，c的偏移为5，不满足对齐要求（它的偏移量应该能够被sizeof（int）大小整除），因此在b后面填充了3个字节，使得c的偏移为8。在b后面填充后，d已经满足对齐要求了，为什么最后还要填充字节呢？或者说，为什么需要满足第三条准则呢？

考虑下面的声明

struct teArray[2];

我们不难知道，teArray[0]的d如果不填充字节，那么teArray[1]的a偏移为14，不满足对齐要求，因此d后面也需要填充字节。

## 编程

实际上，字节对齐的细节都由编译器来完成，我们不需要特意进行字节的对齐，但并不意味着我们不需要关注字节对齐的问题。

### 空间存储

还是考虑前面的结构体test，其占用空间大小为16字节，但是如果我们换一种声明方式，调整变量的顺序，重新运行程序，最后发现结构体test占用大小为12字节

struct test

{

int a;

char b;

short d;

int c;

};

空间存储情况如下,b和c存储在了一个字节快中:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0~3 | 4 | 5 | 6~7 | 8~11 |
| a | b | 填充内容 | c | d |

也就是说，如果我们在设计结构的时候，合理调整成员的位置，可以大大节省存储空间。

### 跨平台通信

由于不同平台对齐方式可能不同，如此一来，同样的结构在不同的平台其大小可能不同，在无意识的情况下，互相发送的数据可能出现错乱，甚至引发严重的问题。因此，为了不同处理器之间能够正确的处理消息，我们有两种可选的处理方法。

1字节对齐

自己对结构进行字节填充

我们可以使用伪指令#pragma pack(n)（n为字节对齐数）来使得结构间一字节对齐。

同样是前面的程序，如果在结构体test的前面加上伪指令，即如下:

#pragma pack(1) /\*1字节对齐\*/

struct test

{

int a;

char b;

int c;

short d;

};

#pragma pack()/\*还原默认对齐\*/

在这样的声明下，任何平台结构体test的大小都为11字节，这样做能够保证跨平台的结构大小一致，同时还节省了空间，但不幸的是，降低了效率。

当然了对于单个结构体，gcc还有如下的方法，使其1字节对齐

struct test

{

int a;

char b;

int c;

short d;

}\_\_attribute\_\_ ((packed));

注:

\_\_attribute\_\_((aligned (n)))，让所作用的结构成员对齐在n字节自然边界上。如果结构中有成员的长度大于n，则按照最大成员的长度来对齐。

\_\_attribute\_\_ ((packed))，取消结构在编译过程中的优化对齐，也可以认为是1字节对齐。

除了前面的1字节对齐，还可以进行人为的填充，即test结构体声明如下:

struct test

{

int a;

char b;

char reserve[3];

int c;

short d;

char reserve1[2];

};

访问效率高，但并不节省空间，同时扩展性不是很好，例如，当字节对齐有变化时，需要填充的字节数可能就会发生变化。

## 总结

虽然我们不需要具体关心字节对齐的细节，但是如果不关注字节对齐的问题，可能会在编程中遇到难以理解或解决的问题。因此针对字节对齐，总结了以下处理建议：

结构体成员合理安排位置，以节省空间

跨平台数据结构可考虑1字节对齐，节省空间但影响访问效率

跨平台数据结构人为进行字节填充，提高访问效率但不节省空间

本地数据采用默认对齐，以提高访问效率

# 字符I/O

## 非格式化

### getchar

从标准输入读，从流中读取下一个字符，并且把它作为函数返回值返回。如果流中不存在更多的字符，则返回常量值EOF（定义为一个整数）。

### getc

### fgetc

从文件中读取字符。一般读取的字符存到字符数组的某个位置，然后还需要手动设置字符数组的结束标志位。

### putchar

### putc

### fputc

### gets

### fgets

### puts

### fputs

## 格式化

### scanf

### fscanf

### ssanf

### printf

printf结果输出到标准输出，即控制台，在日志打印的时候需要。

### fprintf

fprintf可以使用任意输出流，把它的结果作为一个NULL结尾的字符串存储到指定的buffer缓冲区（而不是写到流中）。

### sprint

sprint一般的用法是格式化输出到字符数组中。

printf、fprintf和sprintf区别：

printf结果输出到标准输出，fprintf可以使用任意输出流，把它的结果作为一个NULL结尾的字符串存储到指定的bugger缓冲区，sprintf缓冲区的大小并不是sprintf函数的一个参数，所以输出结果很长溢出缓冲区时，就可能改写缓冲区后面内存位置中的数据，杜绝该问题的方法：

### snprintf

# 二进制I/O

## read

## write

## fread

## fwite

# 文件

一般带有前缀f的是缓冲文件系统函数，比如fopen、fgets等。

## 缓冲文件系统

缓冲文件系统是在内存开辟一个“缓冲区”，为程序中的每一个文件使用。当执行读文件的操作时，从磁盘文件中将数据先读入内存的“缓冲区”，装满后再从内存“缓冲区”依次读入接收的变量。执行写文件的操作时，先将数据写入内存“缓冲区”，待内存“缓冲区”装满后再写入文件。

由此可知，内存“缓冲区”的大小，影响着实际操作外存的次数，内存“缓冲区”越大，则操作外存的次数越小，执行速度越快、效率越高。一般而言，文件“缓冲区”的大小随机器而定。

## 非缓冲文件系统

缓冲文件系统是借助文件结构体指针对文件进行管理，通过文件指针来对文件进行访问，既可以读写字符、字符串、格式化数据，也可以读写二进制数据。

## open

## close

## read

## write

## getc

## getchar

## putc

## putchar

## fopen

## freopen

## feof

## fclose

## fread

## fwrite

## fgetc

## fgets

## fputc

## fputs

## fseek

## ftell

## fcntl

## ftruncate

## rewind

# 指令

## system

## popen

在Linux中通过system()来执行一个shell命令，open()也是执行shell命令，并且通过管道和shell命令进行通信。System和popen给我们处理了fork、exec、waitpid等一系列的处理流程，让我们只需要关注最后的执行结果（函数的返回值）即可。

system和popen都是执行类似的流程，大致是fork🡪exec🡪return。但是system在执行期间调用进程会一直等待shell命令执行完成（waitpid等待子进程结束）才返回；而popen无需等待shell命令执行完成就返回。可以这样理解，system为串行执行，在执行期间调用进程放弃了“控制权”，popen为并行执行。

popen中的子进程没人给他“收尸”，如果你没有在调用popen后调用pclose，那么这个子进程可能变成“僵尸进程”。

在system中对SIGCHLD、SIGINT、SIGQUIT等都做了处理，但是在popen中没有对信号做任何处理。

SIGCHLD是子进程退出的时候发给父进程的一个信号，system中为什么要屏蔽SIGCHLD信号？总结一句就是：system调用能够及时的退出并且能够正确的获取子进程的退出状态（成功回收子进程）。

open没有屏蔽SIGCHLD，主要原因是SIGCHLD是并行的。如果我们在调用popen的时候屏蔽了SIGCHLD，那么如果在调用popen和pclose之间调用进程又创建了其他的子进程并且调用进程注册了SIGCHLD信号处理句柄来处理子进程的回收工作（waitpid），那么这个回收工作会一直阻塞到pclose调用。这也意味着如果调用进程在pclose之前执行了一个wait操作的话，就可以获取到popen创建的子进程的状态，这样在调用pclose的时候就会回收（waitpid）子进程失败，返回-1，同时设置errno为ECHLD，表示pclose无法获取子进程状态。

注：在特权（setuid、setgid）进程中千万注意不要使用system和popen。

# 函数

## 声明

在C语言中，一般是在.h文件中声明函数原型，在.c文件中定义函数。

C语言中的对象必须有且只有一个定义，但是它可以有多个extern声明。

## typedef

在声明函数的时候往往需要用到typedef关键字，它为一种类型引入新的名字，而不是为变量分配空间。在某些方面，typedef类似于宏文本替换——它并没有引入新类型，而是为现有类型取个新名字，二者不同之处具体体现在：

1. 可以用其他类型说明符对宏类型名进行拓展，但对typedef所定义的类型名却不能这样做，如下：

#define peach int

unsigned peach i;//没问题

typedef int banana;

unsigned banana I;;//错误，非法

1. 在连续的几个变量声明中，用typedef定义的类型能够保证声明中所有的变量均为同一类型，而用#define定义的类型则无法保证，如下：

#define int\_ptr int\*

int\_ptr chalk,sheese;

经过宏拓展，第2行变为：

int \*chalk,sheese;

这使得chalk和cheese成为不同的类型，chalk是一个指向int的指针，而cheese则是一个int，与之相反，typedef如下：

typedef char\* char\_ptr;

char\_ptr A,B;

A和B类型相同，虽然前面的类型名变了，但是它们的类型相同，都是指向char的指针。

一般情况下，typedef用于简洁地标识指向其他东西的指针。

不要为了方便对结构体使用typedef，这样做唯一的好处是能使你不必书写struct关键字，但这个关键字可以向你提示一些信息，不应该把它省掉。

typedef应该用在：

1. 数组、结构、指针以及函数的组合类型；
2. 可移植类型。比如当你需要一种至少20比特的类型时，可以对它进行typedef操作，当代码移植到不同的平台时，要选择正确的类型，如short、int、long，只要在typedef中进行修改即可，不需对每个声明都修改。

## 变长参数

### 基本原理

变长参数实现的基本原理：

对于x86来说，函数参数入栈顺序为从右往左，因此，在知道第一个参数地址之后，我们能够通过地址偏移获取其他参数，虽然x86-64在实现上略有不同，但`对于开发者使用来说，实现变长参数函数没有32位和64位的区别。

它主要通过一个类型（va\_list）和三个宏（va\_start、va\_arg、va\_end）来实现

va\_list：存储参数的类型信息，32位和64位实现不一样。

void va\_start ( va\_list ap, paramN );

**参数：**

ap: 可变参数列表地址

paramN: 确定的参数

功能：初始化可变参数列表，会把paraN之后的参数放入ap中

type va\_arg ( va\_list ap, type );

功能：返回下一个参数的值。

void va\_end ( va\_list ap );

功能：完成清理工作。

可变参数函数实现的步骤如下:

1、在函数中创建一个va\_list类型变量

2、使用va\_start对其进行初始化

3、使用va\_arg访问参数值

4、使用va\_end完成清理工作

### 注意事项

变长参数实现注意事项

１、…前的参数可以有1个或多个，但前一个必须是确定类型。

２、传入参数会可能会出现类型提升。

３、va\_arg的type类型不能是char,short int,float等类型，否则取值不正确，原因为第２点。

４、va\_arg不能往回取参数，但可以使用va\_copy拷贝va\_list，以备后用。

５、变长参数类型注意做好检查，例如可以采用printf的占位符方式等等。

６、即便printf有类型检查，但也要注意参数匹配，例如，将int类型匹配%s打印，将会出现严重问题。

７、当传入参数个数少于使用的个数时，可能会出现严重问题，当传入参数大于使用的个数时，多出的参数不会被处理使用。

８、注意字节对齐问题。

## main函数

# 输入输出函数

# 标准函数库

# 进程

## 概述

进程是程序执行时的一个实例，是系统进行资源分配的基本单位。所有与该进程有关的资源，都被记录在进程控制块(PCB)中。以表示该进程拥有这些资源或正在使用它们。另外，进程也是抢占处理机的调度单位，它拥有一个完整的虚拟地址空间。当进程发生调度时，不同的进程拥有不同的虚拟地址空间，而同一进程内的不同线程共享同一地址空间。

线程，有时也被称为轻量级进程，是程序执行流的最小单元，是进程中的一个实体，是被系统独立调度和分派的基本单位。与进程不同，线程与资源分配无关，线程自己不拥有系统资源，它属于某一个进程，并与进程内的其他线程一起共享进程的资源。线程只由相关堆栈（系统栈或用户栈）寄存器和线程控制表TCB组成。

## 创建

### fork

创建一个和当前进程映像一样的进程可以通过fork( )系统调用：

#include <sys/types.h>

#include <unistd.h>

pid\_t fork(void);

成功调用fork( )会创建一个新的进程，它几乎与调用fork( )的进程一模一样，这两个进程都会继续运行。在子进程中，成功的fork( )调用会返回0。在父进程中fork( )返回子进程的pid。如果出现错误，fork( )返回一个负值。

最常见的fork( )用法是创建一个新的进程，然后使用exec( )载入二进制映像，替换当前进程的映像。这种情况下，派生（fork）了新的进程，而这个子进程会执行一个新的二进制可执行文件的映像。这种“派生加执行”的方式是很常见的。

在早期的Unix系统中，创建进程比较原始。当调用fork时，内核会把所有的内部数据结构复制一份，复制进程的页表项，然后把父进程的地址空间中的内容逐页的复制到子进程的地址空间中。但从内核角度来说，逐页的复制方式是十分耗时的。现代的Unix系统采取了更多的优化，例如Linux，采用了写时复制的方法，而不是对父进程空间进程整体复制。

### vfork

在实现写时复制之前，Unix的设计者们就一直很关注在fork后立刻执行exec所造成的地址空间的浪费。BSD的开发者们在3.0的BSD系统中引入了vfork( )系统调用。

#include <sys/types.h>

#include <unistd.h>

pid\_t vfork(void);

除了子进程必须要立刻执行一次对exec的系统调用，或者调用\_exit( )退出，对vfork( )的成功调用所产生的结果和fork( )是一样的。vfork( )会挂起父进程直到子进程终止或者运行了一个新的可执行文件的映像。通过这样的方式，vfork( )避免了地址空间的按页复制。在这个过程中，父进程和子进程共享相同的地址空间和页表项。实际上vfork( )只完成了一件事：复制内部的内核数据结构。因此，子进程也就不能修改地址空间中的任何内存。

vfork( )是一个历史遗留产物，Linux本不应该实现它。需要注意的是，即使增加了写时复制，vfork( )也要比fork( )快，因为它没有进行页表项的复制。然而，写时复制的出现减少了对于替换fork( )争论。实际上，直到2.2.0内核，vfork( )只是一个封装过的fork( )。因为对vfork( )的需求要小于fork( )，所以vfork( )的这种实现方式是可行的。

补充知识点：写时复制

Linux采用了写时复制的方法，以减少fork时对父进程空间进程整体复制带来的开销。

写时复制是一种采取了惰性优化方法来避免复制时的系统开销。它的前提很简单：如果有多个进程要读取它们自己的那部门资源的副本，那么复制是不必要的。每个进程只要保存一个指向这个资源的指针就可以了。只要没有进程要去修改自己的“副本”，就存在着这样的幻觉：每个进程好像独占那个资源。从而就避免了复制带来的负担。如果一个进程要修改自己的那份资源“副本”，那么就会复制那份资源，并把复制的那份提供给进程。不过其中的复制对进程来说是透明的。这个进程就可以修改复制后的资源了，同时其他的进程仍然共享那份没有修改过的资源。所以这就是名称的由来：在写入时进行复制。

写时复制的主要好处在于：如果进程从来就不需要修改资源，则不需要进行复制。惰性算法的好处就在于它们尽量推迟代价高昂的操作，直到必要的时刻才会去执行。

在使用虚拟内存的情况下，写时复制（Copy-On-Write）是以页为基础进行的。所以，只要进程不修改它全部的地址空间，那么就不必复制整个地址空间。在fork( )调用结束后，父进程和子进程都相信它们有一个自己的地址空间，但实际上它们共享父进程的原始页，接下来这些页又可以被其他的父进程或子进程共享。

写时复制在内核中的实现非常简单。与内核页相关的数据结构可以被标记为只读和写时复制。如果有进程试图修改一个页，就会产生一个缺页中断。内核处理缺页中断的方式就是对该页进行一次透明复制。这时会清除页面的COW属性，表示着它不再被共享。

现代的计算机系统结构中都在内存管理单元（MMU）提供了硬件级别的写时复制支持，所以实现是很容易的。

在调用fork( )时，写时复制是有很大优势的。因为大量的fork之后都会跟着执行exec，那么复制整个父进程地址空间中的内容到子进程的地址空间完全是在浪费时间：如果子进程立刻执行一个新的二进制可执行文件的映像，它先前的地址空间就会被交换出去。写时复制可以对这种情况进行优化。

### 区别

1、fork( )的子进程拷贝父进程的数据段和代码段；vfork( )的子进程与父进程共享数据段；

2、**fork( )的父子进程的执行次序不确定；vfork( )保证子进程先运行，在调用exec或exit之前与父进程数据是共享的，在它调用exec或exit之后父进程才可能被调度运行**。

3、vfork( )保证子进程先运行，在它调用exec或exit之后父进程才可能被调度运行。如果在调用这两个函数之前子进程依赖于父进程的进一步动作，则会导致死锁。

4、当需要改变共享数据段中变量的值，则拷贝父进程。

## 进程与线程

### 联系

通常在一个进程中可以包含若干个线程，它们可以利用进程所拥有的资源。但是，一个线程只属于一个进程。进程间相互独立，同一进程的各线程间共享。某进程内的线程在其它进程不可见。而且需要注意的是，线程不是一个可执行的实体。

### 区别

进行和线程之间的差异可以从下面几个方面来阐述：

**调度**：在引入线程的操作系统中，线程是调度和分配的基本单位，进程是资源拥有的基本单位。把传统进程的两个属性分开，线程便能轻装运行，从而可显著地提高系统的并发程度。在同一进程中，线程的切换不会引起进程的切换；在由一个进程中的线程切换到另一个进程中的线程时，才会引起进程的切换。

**并发性**：在引入线程的操作系统中，不仅进程之间可以并发执行，而且在一个进程中的多个线程之间亦可并发执行，因而使操作系统具有更好的并发性，从而能更有效地使用系统资源和提高系统吞吐量。

**拥有资源**：不论是传统的操作系统，还是设有线程的操作系统，进程都是拥有资源的一个独立单位，它可以拥有自己的资源。一般地说，线程自己不拥有系统资源（只有一些必不可少的资源，但它可以访问其隶属进程的资源。

**系统开销**：由于在创建或撤消进程时，系统都要为之分配或回收资源，因此，操作系统所付出的开销将显著地大于在创建或撤消线程时的开销。进程切换的开销也远大于线程切换的开销。

**通信**：进程间通信IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性，因此共享简单。但是线程的数据同步要比进程略复杂。

### 总结

从上面的分析可以看到，似乎线程有很多优势，比如，数据共享效率高，可应对并发操作，有效利用等待时间等等，但是多线程的编程比多进程要复杂，同时，多进程的可靠性较好，因为进程间不会相互影响。实际情况还是需要自己分析拿捏的。但是一般来说，实际应用中常常采用“进程+线程”结合的方式，而不是非此即彼，因为它们两者没有绝对的好与不好，而是适合于不同场景。

# 多线程

# 标准函数库

# C与C++

从实用角度将，C++是ANSI C的一个超集，基本兼容ANSI C，不过C语言有些特性在C++中并不支持。

面向对象的特点是继承和动态绑定。

C++通过类的派生支持继承，通过虚拟函数支持动态绑定，虚拟函数同乐一种封装类体系实现细节的方法。

封装：在C语言中，头文件是非常脆弱的封装实体（这是因为它的组合形式是纯词法意义上的，编译器并不知道头文件是一个语义单元）。

继承：C语言中没有这个特性。