//==============================2017.06.03 BEGIN ===============================//

变量的存储方式

在一个代码块内(或者是在一个函数头部作为参量)使用修饰符register声明的变量属于寄存器存储类。Register修饰符暗示编译程序相应的变量将被频繁的使用，若可能的话，应将其保存在CPU的寄存器当中，从而加快其存取速度。该类与自动存储类(auto)相似，具有自动存储期、代码块作用域和空连接。如果没有被初始化，其值是不确定的。使用register修饰符有几点限制：

1. register变量必须是能被CPU寄存器所接受的类型，这通常意味着register变量必须是一个单个的值，并且其长度应该小于或者是等于整型的长度。这与处理器的类型相关。
2. 声明register仅仅是一个请求，并不是一个命令，因此该变量最终仍然可能是auto变量，没有存放在寄存器当中。
3. 因为变量有可能存放在寄存器当中，应此不能够使用&运算符来获取register变量的地址，这种用法编译器会报错。
4. 只有局部变量和形参可以声明为register变量，全局变量不行。
5. 实际上有些系统并不把register变量存放在寄存器中，而优化的编译系统则可以自动识别使用频繁的变量而把他们放在寄存器当中。

逗号运算符和表达式

C语言中，逗号也是一种运算符，称为逗号运算符，其功能是将两个表达式链接起来组成一个表达式，例如：

y = (x=a+b), (b+c);

本例中，y是整个逗号表达式的值，也就是表达式2的值，x是第一个表达式的值，对于逗号表达式还有三点要注意：

1. 逗号表达式中的表达式1和表达式2也可以是逗号表达式，如表达式1，（表达式2，表达式3，表达式4）;整个表达式的值等于最后一个表达式的值。
2. 程序中使用逗号表达式，通常是要求逗号表达式内各个表达式的值，并不一定要求整个表达式的值。
3. 并不是在所有出现逗号的地方都组成逗号表达式，如在变量说明当中，函数参数表中逗号只是用做各个变量之间的间隔符。

例如：float x=10.5, y=1.8, z=0;

z= (x+=5, y=x+0.2);

则 z = 15.7 x=15.5 y = 15.7

C中的移位操作

移位操作需要考虑一个特殊的问题，即对于有符号的数而言，符号为的处理。通常有两种方法：

1. 逻辑移位。不考虑符号位的问题，原来的数值右移n位以后左边空出的n个位置用零填充。如10000100右移2位变为00100001.
2. 算数移位，原数值进行了右移操作以后，需要保证符号位不变，一次右移n位以后，左边空出的n个位置用原来数值的符号位来填充，如1000 0100 右移2位后变为1110 0001.

若是左移，则不论是算数移位还是逻辑移位，1000 0100 都变为 0001 0000. C语言标准说明对无符号数所进行的移位操作都是逻辑移位，但是对于有符号数，到底是采用逻辑移位还是算数移位则是取决于编译器，不同的编译器所产生的结果有可能会不同。因此，一个程序若是采用了有符号数的右移操作，那么它是不可移植的。

在C中指针的效率往往高于数组下标。因此，编译器对程序中数组下标的操作全部转换为对指针的偏移量的操作。数组中二维数组名代表的是数组的起始地址。数组名加1，是移动一行元素。二维数组名是一个很特殊的地址，参与运算时以行为单位移动，因此也被称为行地址。如a[2][3],a代表第一行的首地址，a[0](&a[0][0])代表第一行第一列元素的地址；a+1代表第二行的首地址，a[1](&a[1][0])代表第二行第一列的地址。

当一个字符指针初始化为指向一个字符串常量时，不能对字符指针变量的目标赋值(可以对该指针变量赋值吗?)。例如：char \*s = “welcome”;相当于 const char \*s = “welcome”;只是省略了const，因此不能对其进行(\*s)++操作(指针的指向可以改变，但是指针指向的内容不能被改变。那如果指针进行了s++操作，那\*(s++)指向的内容可以被改变吗。)。

指针数组和数组指针

指针数组是指由若干个具有相同存储类型和数据类型的指针变量构成的集合，如int \*p[3]是一个含有三个元素的一维数组，每一个数组元素都是一个一级指针。对于指针数组的数组名，也代表数组的起始地址。由于数组的元素已经是指针，数组名就是数组首元素的地址，因此数组名是指针的地址，也就是多级指针了。

数组指针也被称为行指针，定义 int (\*p)[n]，首先说明p是一个指针，指向一个整形的一维数组，这个一维数组的长度是n，也可以说这个n是p的步长。即执行p+1时，p要跨过n个整形数据的长度。若要将一个二维数组赋值给一个指针如

int a[3][4];

Int (\*p)[4]; //该语义是定义一个数组指针，指向含有四个元素的一维数组。

p=a; // 将该二维数组的首地址赋值给p，即a[0]或&a[0][0]

p++; //该语句执行过后，也就是 p=p+1;p跨过行a[0][]指向了行a[1][]

所以数组指针也称为指向一维数组的指针，亦称为行指针。

Attribute机制

GNU C的一大特点就是\_\_attribute\_\_机制，\_\_attribute\_\_可以设置函数属性(function attribute)、变量属性(variable attribute)和类型属性(type attribute)。\_\_attribute\_\_后面会紧跟一对圆括弧，括弧里面是相应的\_\_attribute\_\_参数。

1. 函数属性(function attribute)

函数属性可以帮助开发者把一些特性添加到函数声明当中，从而使得编译器在错误检查方面功能更加强大。\_\_attribute\_\_机制也很容易同非GNU应用程序做到兼容。GNU C需要使用-Wall选项来激活该功能，这是控制警告信息的好方法。

位域

C语言提供一种称为“位域”或“位段”的数据结构。所谓的”位域”是指把一个字节中的二进位划分为几个不同的区域，并说明每一个区域的位数。每一个域有一个域名，允许在程序当中按域名进行操作。这样就可以把几个不同的对象用一个字节的二进制域来表示。

位域的定义与结构体的定义相似，一般如下：

Struct 位域结构名

{

位域列表；

}；

例如:

Struct data{

Unsigned int a:2;

Unsigned int b:3;

Unsigned int c:3;}; 其中a表示data的低两位，b表示data的3-5位，c表示data的6-8位。

关于位域需要注意的问题：

1. 个位域必须存储在同一个字节当中，不能够跨越两个字节。如：

struct data{

Unsigned int a:2;

Unsigned int b:4;

Unsigned int c:3;};

其中a、b占据了六个字节，域名c占3个字节，这样一个完整的字节就没法存储a、b、c，从就得跨两个字节，这是不允许的。当一个字节所剩空间不够存放另外一个位域的时候，应该从下一单元起存放该位域。也可以有意的使某个位域从下一个单元开始。如下：

Struct data{

Unsigned int a:2;

Unsigned int b:3;

Unsigned int :0;

Unsigned int c:3;};这样修改以后，剩下的两位用0来填充，表示不用，c会从第二个字节开始存储。

1. 位域的占用位数不能超过8个二进制位。(位域不能跨字节当然长度不能大于一个字节的长度。)
2. 允许位域无域名(如上面的填充位就没有域名)。

枚举

任意两个枚举成员不能有相同的名称，每一个枚举成员均具有相关联的常数值，此值的类型就是int类型，每一个枚举成员的常数值必须要在该枚举的基础类型的范围之内。允许多个枚举成员有相同的值，没有显示赋值的枚举成员的值，总是前一个枚举成员的值加1.如：

Enum TimeOfDay{

Morning = 1,

Afternoon,

Evening = 1}; 枚举成员都是常量，应此一旦定义了枚举类型，在程序中，就不能再使用赋值语句对其进行赋值。

//==============================2017.06.03 END =================================//

c语言中的数组名和指针之间有一个重要的区别，必须要记住的是指针是变量,故p=a,p++,p--,都是有意义的操作,而数组名是常量,不是变量,因此表达式a=p,a++,a--都是非法的操作,但是&a是存在的，但是为什么编译器会对&a操作给出警告呢,但程序还是能够正常运行。Int \*p=&a这样赋值给出警告信息是因为两端的数据类型不匹配造成的,左边的p是指针类型，是一个int \*类型,而右端是一个整形的地址类型，应该将其转换为(int \*)类型.即

int \*p=(int \*)&a;

对于二维数组而言,编译系统必须能够分辨一维数组和二维数组,a[0]会被作为二维数组存储的首地址,显示的表示就是&a[0][0],“p=a;”是一维数组的表示方法,如果用在二维数组中，编译器就会给出警告信息，但是程序还是能够正常运行,在二维数组中我们可以使用“p=&a[0][0]”,但是更推荐使用”p=a[0]”的方式.

宏 与 const

在c语言中,宏(define)是一个重要的内容。无参数的宏作为常量,而带参数的宏可以提供比函数调用更加高的效率,但是自从推出了const以后，就建议使用const来代替无参数的宏的来定义常量。例如：

#define NUM 3 //常常使用如下定义来替代

Const int NUM=3;

在宏的定义中，一行只能够定义一个宏，以下的用法是错误的：

#define NUM 3, LENGTH 4

下面的这种的宏定义：

#define CH “AB” “CD” “EF”

则CH的值变为字符串ABCDEF

2.1 有参数的宏定义：

在宏的定义当中，最好使用括号将每一个参数都括起来，以免引起与优先级有关的问题,整个结果的表达式最好也用括号括起来，以防当宏用于一个更大的表达式时可能会出现同样的问题。例如：

#define ABS(x) x>0?x:-x

Printf(“%d”,ABS(2-5));

以上的宏定义应该修改为：#define ABS(x) ((x)>0?(x):-(x))

有如下的两个宏定义：

#define PSQR(x) printf(“the square of x is %d.\n”,(x)\*(x))

#define PSQR(x) printf(“the square of “#x” is %d.\n”,(x)\*(x))

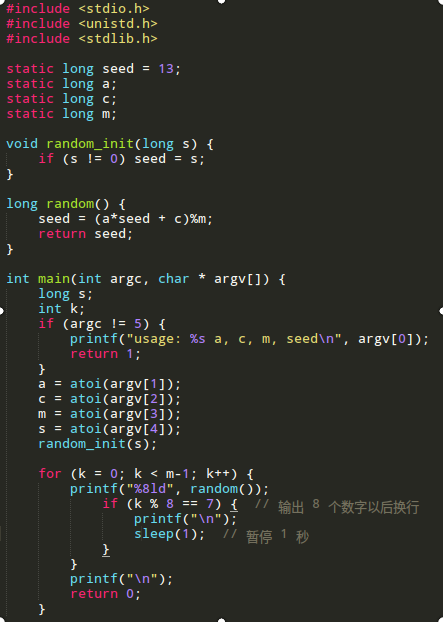
我们的目标是希望在使用宏的时候能够将宏体当中的x替换为宏参数，但是第一个宏定义是不能够完成这样的目标的，因为引号当中的x会被当作是普通的文本来进行处理,我们使用第二种方法就可以实现目标.

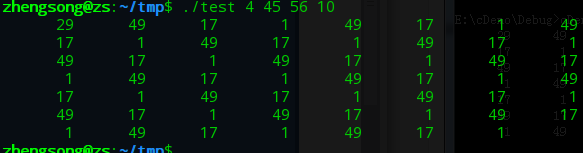
线性同余法产生伪随机数

线性同余法产生随机数序列的公式为：

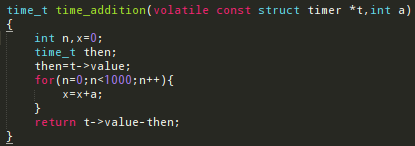
x[k+1] = (a\*x[k] + c) % m

参数 a, c, m 都是由用户来设定的，并和一个种子数（例如 x[0]）一起作为命令行参数传递给生成的程序。





1. 一个变量既可以被声明为变量又可以被声明为常量吗？————一个变量可以既被声明为变量又被声明为常量。常量是指程序中没有任何的代码来改变其值，但是并不是意味着其值在这一段代码之外不能被改变。常量的声明使用const关键字；当const和volatile一起使用的时候，可以将一个变量既声明为变量又声明为常量。Volatile关键字的含义是在编译器事别的范围外变量可以被改变；const关键字的含义是一个不可以被改变的变量。下面的例子可以演示：



该程序的作用是计算出从0加到1000所需要的时间，结构变量t即被声明为变量，又被声明为常量。用户可以看到没有任何代码来改变t->value的值，然而对于时间来说，在硬件设备上他是一个随时都在变化的量，即t->value的值时刻都在改变。如果去掉结构体的volatile修饰符，那么return的返回值将是0.对于变量而言，const是静态的修饰，当volatile和const一起使用时，volatile是用来说明一个变量不仅可以由程序代码改变也可以由硬件、操作系统及其他正在运行的线程改变所改变。上述的程序中因为使用了const修饰，一般代码不可以改变指针指向的内容，有因为加上了volatile的修饰，所以可以改变指针指向的内容。

1. 数据溢出————C语言中的变量包括4种基本的数据类型，即int、char、float、double，数据类型所占的字节随着机器的不同而不同。溢出是程序当中的潜在威胁，不容易被发现，因此在编程的时候要格外的小心，要应用合适的类型变量，避免发生溢出。
2. 数据的类型转换————不同类型的数据进行运算的时候低级别的类型会自动的转换为高级别的类型进行运算，当把高精度的类型强制转换为低精度的类型的时候会产生精度的丢失。
3. 声明的变量和定义的变量有什么不同？————声明的变量是指向编译器描述其类型，但是并没有为其分配任何的空间，定义的变量是指在变量声明的同时给其分配了存储空间来保存变量的值。既声明的变量只有名称，没有指向任何的存储单元，定义的变量指向了一个固定的存储单元。 每一次引用一个变量的时候，引用变量的语句之前必须要出现该变量的声明，该生命可以直接出现在源文件当中，也可以出现在一个头文件当中，因此变量的声明可以在头文件内进行，再通过#include语句对其进行引用。
4. 如何使用goto语句————goto是无条件转向语句，可以插入程序的任何地方，实现程序的无条件的任意转移，既跳转到程序中的其他语句处继续执行。

为了优化程序的设计环境，提高编程效率，ANSI C标准允许在C源程序中加入一些“预处理命令”。预处理命令主要包括宏定义、头文件定义、以及条件变异预处理等。

1. 如何理解C语言的宏？如何使用宏？————在C中用指定的标志符代表一段代码，该标志符被称为“宏“，标识符所代表的那一段代码被称为宏体。程序中使用宏被称为”宏引用“或者是”宏调用“。在编译预处理的时候用宏体去替换源程序中出现的所有”宏名“，将宏名替换为宏体的过程称为”宏替换“或者是”宏展开“。宏是一种预处理指令，通过#define指令来定义。C中的宏可以带参数，也可以不带参数，按照其参数的不同将宏分为有参数和无参数两种形式。
2. :无参数的宏定义：#define 标志宏 宏体 // #define PI 3.1415926
3. :有参数的宏定义：#define 宏名(形参表) 宏体

例如：#define CIRCLE(R,L,S) L=2\*PI\*R;S=PI\*R\*R

在宏的定义中不允许在宏的定义行尾加分号。否则宏展开的时候会将分号作为宏体的一个字符，用于替换宏名。

在宏体中如果使用字符串运算符（#），展开时该运算符作用的元素将被转换为使用引号括起来的字符串，同时与其前面的字符串连接。如下：  
#define BOOK(n) “c/c++”#n // BOOK(book)->被替换为”c/c++book“

在替换字符串中如果使用并接运算符(##),在展开宏时，将两个字符合并为一个字符串。

宏定义命令#include出现再文件的外部，通常情况下放在文件的开头处。

宏名的有效范围是从定义命令之后到当前的文件结束

进行宏定义时，可以引用已经存在的宏名

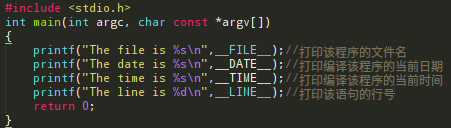
宏展开时不会对宏定义进行任何的检查，若有错误，只能在编译源程序的时候发现

使用宏的几点好处：

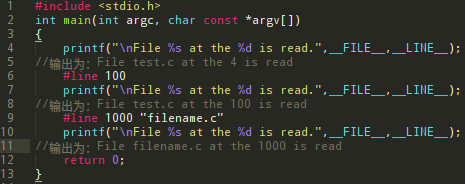
1. .最大的好处是提高程序的可维护性。
2. .宏省去了相同代码的重复书写。
3. .只需定义一次。可以使用多次，因此使用宏能提高程序的编码速度。
4. .宏的参数可以为任意的类型，因此在编码的时候不必考虑宏的参数类型。
5. 标准预定义宏包括哪些————如下：

|  |  |
| --- | --- |
| 宏名 | 作用 |
| \_\_LINE\_\_ | 记录当前宏所在源文件中的行号 |
| \_\_FILE\_\_ | 记录当前源文件的名称 |
| \_\_DATE\_\_ | 记录当前的日期 |
| \_\_TIME\_\_ | 记录当前的时间 |
| \_\_STDC\_\_ | 判断是否进行ANSI C编译，如果为1将定义该宏，否则不定义。只允许在定义中使用 |
| \_\_cplusplus | 判断是C++还是C，如果使用的是C编译器，该宏不被定义。只允许在定义中使用 |

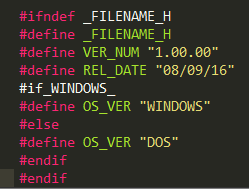
预定义的宏在编译的过程中都是常量，虽然许多宏定义的用法因编译器而异，但是\_\_FILE\_\_和\_\_LINE\_\_在所有的编译器中都是一致的。上表中列出的预定义宏不允许由#define、#undef指令定义或者取消。域定义宏的宏名应该以下划线加大写字母或者是两个下划线开始。例如：



1. 如何改变预处理器的行计数？————C预处理器提供了#line预处理，允许用户设置当前的行号。使用#line预处理的例子如下：

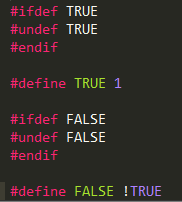


1. 宏与函数的区别————调用函数时，先将调用函数的地址放入堆栈，函数执行完成以后再将调用函数的地址返回给程序，最后回调调用原点；而宏展开是使用取代标志直接将宏名替换。 每一次调用函数都要进行现场保护和现场恢复，因此需要较多的执行时间，但是函数调用不会使得目标程序变长，而宏展开是在编译时进行的。不会占用运行时间，因此宏比一般的函数调用节省了函数调用时间，但是每一次引用宏都会使目标程序增大一次，并且宏没有类型转换的功能。宏调用的参数是没有类型的，并且宏调用是没有返回值的。
2. C语言的头文件————预处理器会将头文件的内容在源程序包含指令处展开，头文件中函数的定义在其他的源文件中会同样的出现，导致函数被重复定义。解决此问题的方法是所有可执行的代码、对象以及函数的定义都应该在.c的源文件（实施文件）中，头文件中只是存在其声明。具体的做法就是在头文件中为全局变量的声明增加extern修饰符，并在相应的C源文件中定义对象或则是函数。 头文件的目的是定义库和任何使用该库程序之间的接口。所谓的接口就是两个分离部分之间的公共边界。
3. 如何避免多次包含同一个文件————通过引入预编译头文件(precompiledheader,即头文件和相关的依赖都已经以一种编译好的状态存储起来)技术，文件包含的命令可以嵌套。但是文件嵌套也会导致一些问题，例如多个头文件相互引用，导致编译进入死循环。解决的办法有两个，一：使用宏保证头文件只被包含一次。二：将头文件的引用放入源文件中，即尽量将#include头文件放入源文件中。

使用#ifndef和#define指令可以避免多次包含一个头文件。在创建头文件时使用#define指令可以定义唯一的标志符名称，然后通过#ifndef指令检查该标志符名称是否被定义。若标志符名称已被定义，则说明该头文件已经被包含，如下：  


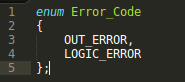
自定义头文件时，应该要避免与预编译器提供的头文件发生冲突，可采用的方法是在定义标志符时不使用前缀，即不使用下划线。

1. 头文件使用规则————(1):避免多重包含；(2):前置声明是为了降低编译依赖，防止修改一个头文件引发的多尼诺效应(如多个相关联的文件，修改其中一个文件，其他文件也跟着改变)。(3)合理使用内联函数可以提高代码的执行效率。
2. 如何重写一个定义好的宏————



1. 使用枚举和使用#define定义常量有什么不同？————(1)使用#define占用内存少，因为#define定义的常量将会直接进入源代码，而无需在内存中分配变量空间。

而使用枚举会使得程序更加的容易维护，因为枚举常量是由编译程序自动生成的，而标识符常量必须由用户手动赋值。





在上面的代码中，使用枚举OUT\_ERROR和LOGIC\_ERROR两个枚举常量会被编译程序自动的赋值为0和1。若使用define语句则需要右边的代码。

使用const来定义常量可以定义局部常量，函数体内通过const定义的常量为局部常量，作用仅限于该函数体。而使用#define定义的常量，即使是在函数体内其作用的范围也是从定义点开始，直到文件结束（除非用#undef取消）。

1. 判断字符是何种类型的？————

|  |  |
| --- | --- |
| 函数 | 作用 |
| Isalnum() | 判断是否为字母或数字，是返回非零值，不是返回0 |
| Isalpha() | 是否为字母 |
| Isascii() | 是否包含ascii |
| Iscntrl() | 是否为控制字符 |
| Isdigit() | 是否为数字 |
| Isgraph() | 是否为图形字符 |
| Isprint() | 是否可打印 |
| Ispunct() | 是否为标点符号 |
| Isspace() | 是否包含空白符号 |
| Isxdigit() | 是否为十六进制值 |

这些用来判断字符是何种类型的函数，他们的参数应该是单个的字符或者是用来存储单个字符的变量。

1. 如何使用外部变量以及外部静态变量————外部变量也称为全局变量，它的作用域是从定义位置的开始到文件末尾结束。采用extern来声明全局变量可以扩大其作用域。全局变量可以在一个或多个文件中声明。如 extern int x；表明使用外部的全局变量x。

外部静态变量也称为静态全局变量。在C语言中，全局变量都采用静态存储方式，即在程序编译时对相应的全局变量分配固定的存储单元。Static声明的全局变量不仅是限定全局变量使用静态存储，而且声明全局变量只能够用于当前的文件。

1. 创建支持多参数多类型的函数————va\_list()是C中解决变参问题的一组宏，用法如下：
2. .首先在函数中定义va\_list类型的变量，该变量是指向参数的指针。
3. .然后使用va\_list()宏初始化va\_list变量，第一个参数是va\_list类型变量，第二个参数表示在第一个参数中的某个指定位置，第二个参数是一个固定值。
4. 然后利用va\_arg()宏返回可变参数，它的第二个参数是返回值类型
5. 最后用va\_end()宏结束可变参数的获取。如果函数有多个可变参数，依次调用va\_arg()宏获取各个参数。

Va\_list()宏在编译器中的处理如下：

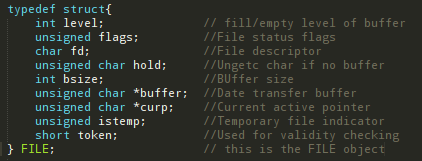
1. .在运行va\_list(ap,v)以后，ap指向堆栈的地址。
2. Va\_list()取得类型t的可变参数的值，此时首先执行apt=sizeof(t类型)，让ap指向下一个参数的地址。然后返回ap-sizeof(t类型)指定的t类型指针，这正是第一个可变参数在堆栈中的地址。最后用\*取得这个地址内存放的数据。
3. Va\_end()，此宏定义为ap=((char \*)0),使得ap不在指向堆栈。

需要主意的是由于参数的地址用于va\_list()宏，因此参数不能够被声明为寄存器变量、函数或者是数组类型。因为va\_start/va\_arg/va\_end是宏的定义，所以可变参数的类型和个数完全在该函数中由程序代码控制，即如果实现智能识别可变参数则必须在程序中判断。同时编译器对可变参数的函数原型检查不够严格，所以需要谨慎使用这种函数。



1. 读写文件时文本模式和二进制模式有什么区别？————文件流可以分为两种类型，即文本流和二进制流(1).使用二进制模式打开文件时，文件本身的内容与用户编写程序时使用的函数读到的内容完全相同（或者说是和磁盘上的内容完全相同）。若是文本模式。操作系统在将文件内容传给上层程序（库函数或者是用户程序）的时候，或者说是上层程序通过操作系统向文件写入内容的时候，操作系统都会预先进行一层预处理（或者说是转义），具体的过程依赖于操作系统的实现。最常见的就是将回车符“\r\n”解释为“\n”（读出时）。将“\n”解释为“\r\n”（）写入时。

通常，文本流用来读写标准的文本文件，或者是将字符输出到屏幕或者是打印机，或者是接受键盘的输入；而二进制流用来读写二进制文件，例如：图形、字处理文档、读取鼠标输入以及读写调制解调器。二进制流比文本流节省空间，并且不对某一些特殊字符进行转换，因此使用二进制流可以加快流的读写速度，从而提高效率。对于含有大量数字信息的数字流，可以采用二进制的方式；对于大量的字符信息的流，则采用文本流的方式。



1. 为什么要尽量减少文件的I/O操作？————与计算机的CPU和存储器的电子速度相比较，磁盘的运行速度是相当的慢的。因此，用户应当要尽量的减少程序中必须进行的磁盘I/O数量，对于文件操作而言，打开文件可能是最耗时间的。因此用户应该要先检查程序，确认是否有打开程序的必要性。如果I/O模型完善，则可以大大提升性能，但是I/O仍然是相当的慢的。建议用户使用多线程，某些线程专门用于I/O操作。但是多线程不会大规模提供I/O速度，即使是多路复用也不可能达到，因此多线程提高I/O的速度有限。
2. 数组与动态存储那个更好？＿＿首先，数组更易于理解和使用．然后，因为编译器给数组分配空间,程序没有与动态数组存储分配有关的运行时开销.所以程序使用数组可能在执行速度上更快一些. 系统在编译的时候首先确定数组的长度,然后把该数组分配到数据区.而动态存储是在程序运行的时候才确定其大小的,而且其长度可以改变,系统把它分配到堆栈区.因此在实际应用的时候应该根据需要选择使用数组还是动态存储.
3. 如何使用位字段结构?\_\_\_\_\_\_有些变量的返回值包含的位具有特定的含义,许多函数通过使用位字段结构来减少函数中变量的个数,从而减少分配的内存.当数值的位具有特殊的含义的时候,C语言可以使用按位运算符获取数值(指定位).位字段结构是一种特殊的结构,如果需要按位访问一个字节或者是字的多个位时,位结构比按位运算符更简单,而且使用位字段结构可以节省存储空间.

在使用位字段时,要注意声明位字段结构的成员必须是unsigned int类型,同时也必须要定义一个结构来指定相应位的含义.

1. 指针函数和函数指针分别是什么?--------
2. 函数指针:函数指针是指向函数的指针变量,因而函数指针本身首先应该是指针变量,C语言在编译的时候,每一个函数都有一个入口地址,该入口地址就是函数指针指向的地址.有了指向函数的指针变量以后,就可以使用该指针变量调用函数,函数指针有两个用途:即调用函数和作为函数的参数(典型的如回调函数).

格式: 数据类型标识符 (\*指针变量名)(参数);

(2)指针函数:返回值是一个指针类型的函数.

1. malloc()和calloc()函数的区别?---------malloc和calloc的区别在于malloc函数指定所需要的字节数,而calloc函数指定特定长度的所需元素的个数.calloc的格式如下:

Void \*calloc(size\_t number\_of\_items, size\_t item\_size);其中number\_of\_items指定要为多少个元素分配内存,items\_size参数以字节的形式指定每个元素的长度.同时malloc函数获得的内存空间不会被初始化,要使用memset来将其初始化为0,而使用calloc分配的内存空间会被初始化为0;

1. 如何从堆栈中分配内存?--------程序可以利用alloca函数从堆栈中分配内存,该函数的格式如下: void \*alloca(size\_t number\_of\_bytes);

使用alloca()函数分配的内存空间不需要使用free()去释放,因为free()函数使用的是堆而alloca()函数使用的是堆栈.当alloca()函数结束的时候,程序将会自动的释放这些内存.

1. 如何改变被分配的内存区域的大小?---------在程序中可以使用realloc()函数来改变已经分配的内存区域的大小,格式为: void \*realloc(void \*block,size\_t desired\_bytes);

其中block是指向被分配的内存的指针;desired\_bytes是新的内存区域的大小.通过调用realloc()函数可以减小或者是扩展某个内存区域.执行成功将返回一个指向该内存区域的起始处的指针,否则返回NULL;