# 1.volatile:

声明变量是容易变化的，程序在编译时不能对变量优化（放到寄存器中），每次需要使用这个变量时都需要直接从内存中取。一般在中断服务程序中改变变量值。

# 2.static:

(1)修饰变量：无论全局还是局部都存放在内存静态区（和全局变量一样，默认初始化为0），全局静态变量的作用域为：定义开始往下的该文件的所有函数。局部静态变量：只在定义的函数内有效。作用域都是在该文件。

（2）修饰函数：static含义不是指存储方式，而是指函数的作用域局限于定义的文件（内部函数）

函数指针：每一个函数都占用一段内存单元，它们有一个起始地址，指向函数入口地址的指针称为函数指针。注意：函数指针的本质是一个指针变量，且指针指向的函数的入口地址

指向函数的指针变量的一般定义形式为：

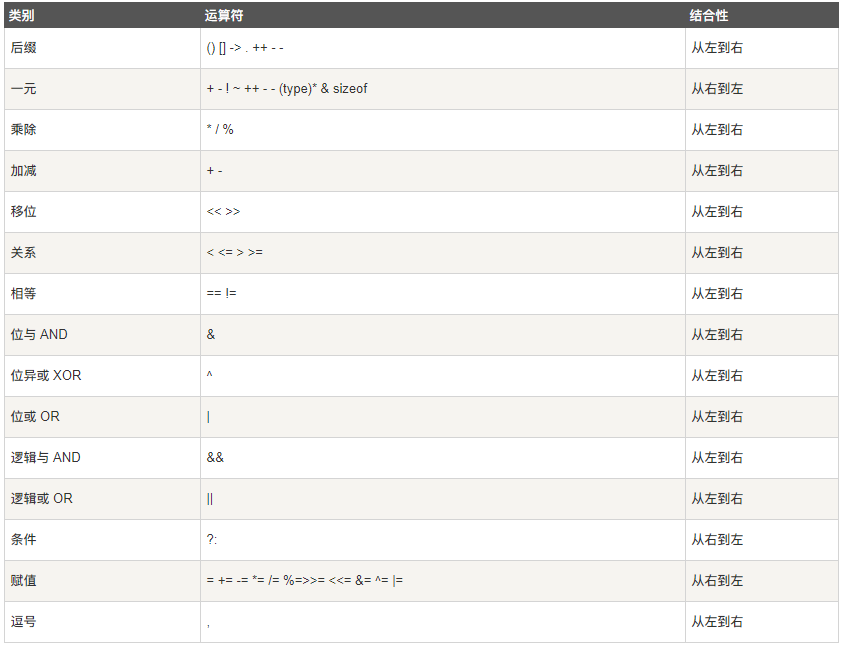
  数据类型   (\*指针变量名)   (参数表);

1. int (\*p)(int a, int b);  //p是一个指向函数的指针变量，函数返回类型为整型
2. int \*p(int a, int b);     //p是函数名，函数返回类型为整型指针
3. int max(int x, int y) { return x+y; }
5. void main()
6. {
7. //定义一个函数指针
8. int (\*p)(int a, int b);
9. p = max; //正确
10. p = &max; //正确
12. //通过函数指针定义变量
13. p(1,2); //正确
14. (\*p)(1,2); //同样正确
15. }

函数指针类型：通过typedef 定义函数指针类型

1. int max(int x, int y)
2. {
3. return x > y ? x : y;
4. }
5. //定义函数指针类型
6. typedef int(\*pFun)(int, int)//可以用来声明该类型的变量
8. int main()
9. {
10. //通过函数指针类型定义函数指针
11. pFun fun = &max;
12. fun(1,2);
13. return 0;
14. }

# 3.运算符优先级



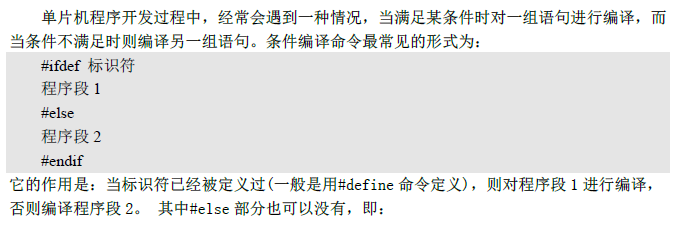
# 4.位操作

A = 00111000，A<<2==11100000,A>>2==00001110

GPIOx->BSRR = (((uint32\_t)0x01) << pinpos);//将BSRR的第pinpos位置1

GPIOA->ODR|=1<<5; //PA.5输出高,不改变其他位

ifdef条件编译：



# 5.Extern 关键字

extern变量声明：

extern可置于变量（全局变量？）或者函数前，可以置于变量或者函数前，以表示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。这里面要注意，对于extern申明变量可以多次，但定义只有一次。在我们的代码中你会看到看到这样的语句：

extern u16 USART\_RX\_STA;

这个语句是声明USART\_RX\_STA变量在其他文件中已经定义了，在这里要使用到。所以，你肯定可以找到在某个地方有变量定义的句：

u16 USART\_RX\_STA;

声明和定义的区别：有无分配内存

**对于变量**：

extern int a; // 声明一个全局变量 a

int a; // 定义一个全局变量 a（默认省略extern）

extern int a =0 ; // 定义一个全局变量 a 并给初值。

int a =0; // 定义一个全局变量 a, 并给初值，

在另一个文件要引用一个全局变量时必须要声明，extern int a。

**对于函数：**对于函数也一样，也是定义和声明，定义的时候用extern，说明这个函数是可以被外部引用的，声明的时候用extern说明这是一个声明。 但由于函数的定义和声明是有区别的，定义函数要有函数体，声明函数没有函数体(还有以分号结尾)，所以函数定义和声明时都可以将extern省略掉，反正其他文件也是知道这个函数是在其他地方定义的，所以不加extern也行。两者如此不同，所以省略了extern也不会有问题。

总结：(a)对于一个文件中调用另一个文件的全局变量，因为全局变量一般定义在原文件.c中，我们不能用#include包含源文件而只能包含头文件，所以常用的方法是用extern int a来声明外部变量。 另外一种方法是可以是在a.c文件中定义了全局变量int global\_num ，可以在对应的a.h头文件中写extern int global\_num ，这样其他源文件可以通过include a.h来声明她是外部变量就可以了。

（b）还有变量和函数的不同举例 **int fun();** 和 **extern int fun();** 都是声明(定义要有实现体)。 用 **extern int fun()** 只是更明确指明是声明而已。而 **int a;** 是定义 **extern int a;** 是声明。

# 6.结构体

struct tag{

Member-list

Member-list

}variable-list;

tag为结构体名可有可无，variable-list为该结构体的变量（在声明结构体的同时定义了变量，要分配内存了），或者在声明结构体之后再声明变量。

结构体的成员可以包含其他结构体，也可以包含指向自己结构体类型的指针，而通常这种指针的应用是为了实现一些更高级的数据结构如链表和树等。

常用typedef struct{memberlist}simple; ,用typedef为该结构体取别名，然后再定义变量simple a,b;

例如：我们初始化一个串口

typedef struct

{

uint32\_t USART\_BaudRate;

uint16\_t USART\_WordLength;

uint16\_t USART\_StopBits;

uint16\_t USART\_Parity;

uint16\_t USART\_Mode;

uint16\_t USART\_HardwareFlowControl;

} USART\_InitTypeDef;

void USART\_Init(USART\_TypeDef\* USARTx, USART\_InitTypeDef\* USART\_InitStruct);

# 7.预处理和头文件

1．头文件由编译器自带的头文件（例如stdio.h这些编译器看的懂，但还是会去相应的路径找）和自定义的头文件组成。系统头文件引用格式为：#include <file.h>，指示编译器去系统目录去的标准列表中去找相应的文件。程序员自定义的：#include “file.h”,去包含当前文件的目录寻找。

2.引用头文件相当于复制该头文件下面的内容到引用的文件中，因此不能多次引用同一个头文件，否则编译器会处理两次头文件的内容（同时会造成头文件下声明的函数或者声明定义的变量等多次重复）。可以用以下方法避免：

#ifndef head\_file

#define head\_file

File

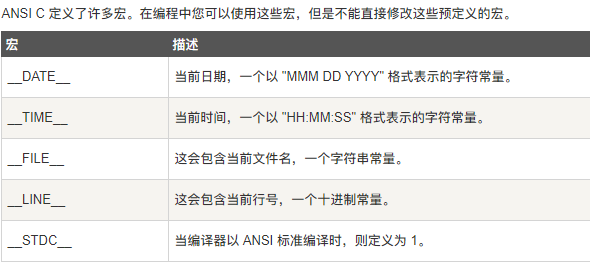
#endif

在这个过程中如果已经有一个文件引用了这个头文件，那么就定义了head\_file，file里面的内容就会复制到引用的文件中，此时如果有另一个文件想引用该头文件，那么预处理器就不会将该头文件的内容复制过去。

3.预处理器cpp,文本替换，预处理指令有：

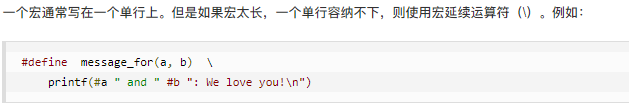


4.预定义宏



5.预处理运算符：

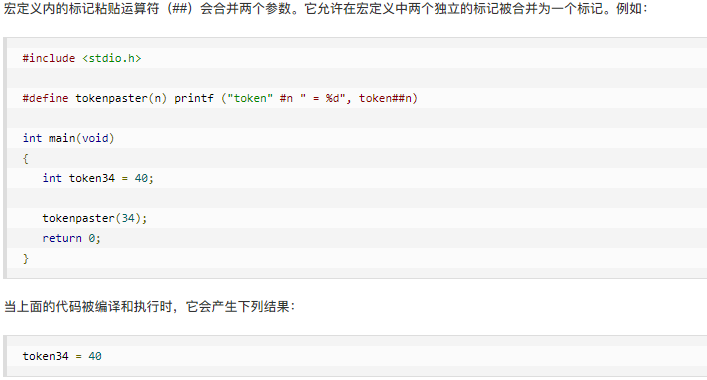
宏延续运算符\,



字符串常量化#



标记粘贴##



对于多线程任务：

重入一般可以理解为一个函数在同时多次调用，例如操作系统在进程调度过程中，或者单片机、处理器等的中断的时候会发生重入的现象。

可重入的定义源于单线程环境。在单线程环境下，一段代码在执行中可能被硬件中断，并转而调用中断服务程序（ISR）。在本次调用中断处理函数之前，有可能中断处理函数已经在执行。因此，任何中断处理函数都应该是可重入的。

可重入（reentrant）函数可以由多于一个任务并发使用，而不必担心数据错误。相反， 不可重入（non-reentrant）函数不能由超过一个任务所共享，除非能确保函数的互斥（或者使用信号量，或者在代码的关键部分禁用中断）。可重入 函数可以在任意时刻被中断，稍后再继续运行，不会丢失数据。可重入函数要么使用本地变量，要么在使用全局变量时保护自己的数据。

可重入函数：

不为连续的调用持有静态数据。    
不返回指向静态数据的指针；所有数据都由函数的调用者提供。    
使用本地数据，或者通过制作全局数据的本地拷贝来保护全局数据。  
如果必须访问全局变量，记住利用互斥信号量来保护全局变量。  
绝不调用任何不可重入函数。

**全局变量：**在整个工程文件内都有效；“在函数外定义的变量”，即从定义变量的位置到本源文件结束都有效。由于同一文件中的所有函数都能引用全局变量的值，因此如果在一个函数中改变了全局变量的值， 就能影响到其他函数中全局变量的值。

1. 不必要的常量经常使用

可变参数函数：

标准库提供的一些参数的数目可以有变化的函数。例如我们很熟悉的printf，它需要有一个格式串，还应根据需要为它提供任意多个“其他参数”。这种函数被称作“具有变长度参数表的函数”，或简称为“变参数函数”。我们写程序中有时也可能需要定义这种函数。要定义这类函数，就必须使用标准头文件<stdarg.h>，使用该文件提供的一套机制，并需要按照规定的定义方式工作。本节介绍这个头文件提供的有关功能，它们的意义和使用，并用例子说明这类函数的定义方法。  
         一个变参数函数至少需要有一个普通参数，其普通参数可以具有任何类型。在函数定义中，这种函数的最后一个普通参数除了一般的用途之外，还有其他特殊用途。下面从一个例子开始说明有关的问题。  
        假设我们想定义一个函数sum，它可以用任意多个整数类型的表达式作为参数进行调用，希望sum能求出这些参数的和。这时我们应该将sum定义为一个只有一个普通参数，并具有变长度参数表的函数，这个函数的头部应该是（函数原型与此类似）：  
int sum(int n, ...)  
        我们实际上要求在函数调用时，从第一个参数n得到被求和的表达式个数，从其余参数得到被求和的表达式。在参数表最后连续写三个圆点符号，说明这个函数具有可变数目的参数。凡参数表具有这种形式（最后写三个圆点），就表示定义的是一个变参数函数。注意，这样的三个圆点只能放在参数表最后，在所有普通参数之后。  
        为了能在变参数函数里取得并处理不定个数的“其他参数”，头文件<stdarg.h>提供了一套机制。这里提供了一个特殊类型va\_list。在每个变参数函数的函数体里必须定义一个va\_list类型的局部变量，它将成为访问由三个圆点所代表的实际参数的媒介。下面假设函数sum里所用的va\_list类型的变量的名字是vap。在能够用vap访问实际参数之前，必须首先用“函数”va\_start做这个变量初始化。函数va\_start的类型特征可以大致描述为：  
va\_start(va\_list vap, 最后一个普通参数)  
实际上va\_start通常并不是函数，而是用宏定义实现的一种功能。在函数sum里对vap初始化的语句应当写为：  
va\_start(vap, n);  
在完成这个初始化之后，我们就可以通过另一个宏va\_arg访问函数调用的各个实际参数了。宏va\_arg的类型特征可以大致地描述为：  
类型 va\_arg(va\_list vap, 类型名)  
        在调用宏va\_arg时必须提供有关实参的实际类型，这一类型也将成为这个宏调用的返回值类型。对va\_arg的调用不仅返回了一个实际参数的值（“当前”实际参数的值），同时还完成了某种更新操作，使对这个宏va\_arg的下次调用能得到下一个实际参数。对于我们的例子，其中对宏va\_arg的一次调用应当写为：  
v = va\_arg(vap, int);  
这里假定v是一个有定义的int类型变量。  
        在变参数函数的定义里，函数退出之前必须做一次结束动作。这个动作通过对局部的va\_list变量调用宏va\_end完成。这个宏的类型特征大致是：  
void va\_end(va\_list vap);  
下面是函数sum的完整定义，从中可以看到各有关部分的写法：  
int sum(int n, ...) {  
      va\_list vap;  
       int i, s = 0;  
       va\_start(vap, n);  
       for (i = 0; i < n; i++) s += va\_arg(vap, int);  
       va\_end(vap);  
       return s;  
}  
这里首先定义了va\_list变量vap，而后对它初始化。循环中通过va\_arg取得顺序的各个实参的值，并将它们加入总和。最后调用va\_end结束。  
下面是调用这个函数的几个例子：  
k = sum(3, 5+8, 7, 26\*4);  
m = sum(4, k, k\*(k-15), 27, (k\*k)/30);  
         在编写和使用具有可变数目参数的函数时，有几个问题值得注意。首先，虽然在上面描述了头文件所提供的几个宏的“类型特征”，实际上这仅仅是为了说明问题。因为实际上我们没办法写出来有关的类型，系统在预处理时进行宏展开，编译时即使发现错误，也无法提供关于这些宏调用的错误信息。所以，在使用这些宏的时候必须特别注意类型的正确性，系统通常无法自动识别和处理其中的类型转换问题。  
        第二：调用va\_arg将更新被操作的va\_list变量（如在上例的vap），使下次调用可以得到下一个参数。在执行这个操作时，va\_arg并不知道实际有几个参数，也不知道参数的实际类型，它只是按给定的类型完成工作。因此，写程序的人应在变参数函数的定义里注意控制对实际参数的处理过程。上例通过参数n提供了参数个数的信息，就是为了控制循环。标准库函数printf根据格式串中的转换描述的数目确定实际参数的个数。如果这方面信息有误，函数执行中就可能出现严重问题。编译程序无法检查这里的数据一致性问题，需要写程序的人自己负责。在前面章节里，我们一直强调对printf等函数调用时，要注意格式串与其他参数个数之间一致性，其原因就在这里。  
        第三：编译系统无法对变参数函数中由三个圆点代表的那些实际参数做类型检查，因为函数的头部没有给出这些参数的类型信息。因此编译处理中既不会生成必要的类型转换，也不会提供类型错误信息。考虑标准库函数printf，在调用这个函数时，不但实际参数个数可能变化，各参数的类型也可能不同，因此不可能有统一方式来描述它们的类型。对于这种参数，C语言的处理方式就是不做类型检查，要求写程序的人保证函数调用的正确性。  
假设我们写出下面的函数调用：  
k = sum(6, 2.4, 4, 5.72, 6, 2);  
        编译程序不会发现这里参数类型不对，需要做类型转换，所有实参都将直接传给函数。函数里也会按照内部定义的方式把参数都当作整数使用。编译程序也不会发现参数个数与6不符。这一调用的结果完全由编译程序和执行环境决定，得到的结果肯定不会是正确的

# 8.c程序的内存分布：

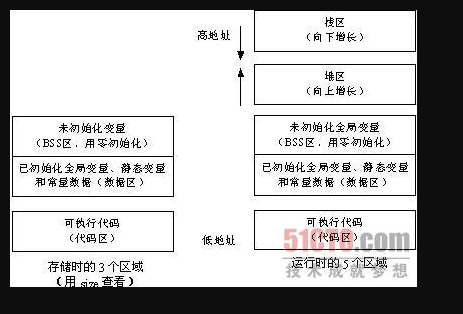
C可执行程序在**存储时**（还没调入内存）分为三个区：

**1.代码区（text segment）**。存放CPU执行的机器指令（machine instructions）。通常，代码区是可共享的（即另外的执行程序可以调用它），因为对于频繁被执行的程序，只需要在内存中有一份代码即可。代码区通常是只读的，使其只读的原因是防止程序意外地修改了它的指令。另外，代码区还规划了局部变量的相关信息。

**2.全局初始化数据区/静态数据区（initialized data segment/data segment）**。该区包含了在程序中明确被初始化的全局变量、静态变量（包括全局静态变量和局部静态变量）和常量数据（如字符串常量）。例如，一个不在任何函数内的声明（全局数据）

**3.未初始化数据区。亦称BSS区（uninitialized data segment）**，存入的是**全局未初始化变量**。BSS这个叫法是根据一个早期的汇编运算符而来，这个汇编运算符标志着一个块的开始。BSS区的数据在程序开始执行之前被内核初始化为0或者空指针（NULL）。

程序在运行时，c编译程序占用的内存分为：**代码区，初始化数据区（静态数据区），未初始化数据区，堆区，栈区。**



1. **代码区：**代码区指令根据程序设计流程依次执行，对于顺序指令，则只会执行一次（每个进程），如果反复，则需要使用跳转指令，如果进行递归，则需要借助栈来实现。代码区的指令中包括操作码和要操作的对象（或对象地址引用）。如果是立即数（即具体的数值，如5），将直接包含在代码中；如果是局部数据，将在栈区分配空间，然后引用该数据地址；如果是BSS区和数据区，在代码中同样将引用该数据地址。
2. **初始化全局数据区：**全局初始化数据区/静态数据区（Data Segment）。只初始化一次。
3. **未初始化全局数据区：**在运行时改变其值。
4. 栈区（stack）。**由编译器自动分配释放（程序编译的时候已经有相应的机器指令来分配释放）**，存放函数的参数值、局部变量的值等。其操作方式类似于数据结构中的栈。每当一个函数被调用，该函数返回地址和一些关于调用的信息，比如某些寄存器的内容，被存储到栈区。然后这个被调用的函数再为它的自动变量和临时变量在栈区上分配空间，这就是C实现函数递归调用的方法。每执行一次递归函数调用，一个新的栈框架就会被使用，这样这个新实例栈里的变量就不会和该函数的另一个实例栈里面的变量混淆。
5. 堆区（heap）。用于动态内存分配。堆在内存中位于bss区和栈区之间。一般由程序员分配和释放，若程序员不释放，程序结束时有可能由OS回收。

**内存分配方式**：在C语言中，对象可以使用静态或动态的方式分配内存空间。

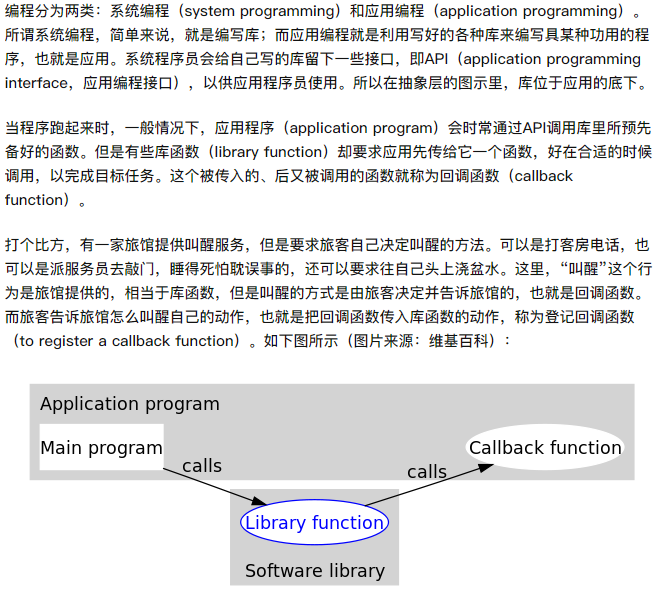
静态分配：编译器在处理程序源代码时分配。动态分配：程序在执行时调用malloc库函数申请分配。静态内存分配是在程序执行之前进行的因而效率比较高，而动态内存分配则可以灵活的处理未知数目的。静态与动态内存分配的主要区别如下：静态对象是有名字的变量，可以直接对其进行操作；动态对象是没有名字的变量，需要通过指针间接地对它进行操作。静态对象的分配与释放由编译器自动处理；动态对象的分配与释放必须由程序员显式地管理，它通过malloc()和free两个函数（C++中为new和delete运算符）来完成。

**堆栈的区别**：栈是向低地址扩展的数据结构，是一块连续的内存区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，当申请的空间超过栈的剩余空间时，将提示溢出。因此，用户能从栈获得的空间较小。

**堆的增长方向是向上的，即向着内存地址增加的方向；栈的增长方向是向下的，即向着内存地址减小的方向。**

**效率不一样：**栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行。堆则是C函数库提供的，它的机制很复杂，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆内存中搜索可用的足够大的空间，如果没有足够大的空间（可能是由于内存碎片太多），就有需要操作系统来重新整理内存空间，这样就有机会分到足够大小的内存，然后返回。显然，堆的效率比栈要低得多。

# 9.回调函数（callback）



回调函数通常和应用处于同一抽象层，

回调函数的优势：

我们可以在应用层注册登记不同的回调函数，在应用层其他函数调用库函数，库函数在运行时可以通过传入不同的回调函数来改变库函数的行为。