## 关键字const

修饰变量时，表示该变量在赋初值之后不可修改；

如：const int limit = 10;

上面的代码等同于 int const limit = 10;

const 修饰它左侧的类型，左侧如果不存在就是右侧的类型。

如下面的两个定义是相同的，都表示指向只读整数类型的只读指针：

int const \* const p;

const int \* const p;

## 指针相容

指针相容，指的是指针A的类型和指针B的类型是相容的，那么可以A = B，A的类型和B是相同的，且A的类型的限定符包含B的类型的限定符；

如下面是相容的（compatible）：

char \*cp;

const char \*ccp;

ccp = cp;

cp和ccp都是相同类型的指针，指向char，ccp包含了const限定符和cp的限定符（无限定符）；

但下面的不相容（incompatible）：

char \*\*cpp;

const char \*\* ccpp;

ccpp = cpp;

cpp和ccpp都不含有限定符，const修饰的是char，两者是不同的类型，不相容。

## 类型转换

自动类型转换：当执行算数运算时，操作数的类型如果不同，就会发生转换，数据类型一般朝着浮点精度更高，长度更长的方向转换，整形数如果转换为signed，不会丢失信息，就转换为signed，否则转换为unsigned。

ANSI标准采用值保留原则（value preserving），K&R采用的是无符号保留原则（unsigned preserving），所以下面代码-1分别被解释为负数和正数。

如下面代码：

#include <stdio.h>

void main()

{

if( -1 < (unsigned char)1)

{

printf("-1 is less than (unsigned char)1: ANSI semantics.\n");

}

else

{

printf("-1 is NOT less than (unsigned char)1: K&R semantics.\n");

}

}

在使用位域或者掩码时才使用无符号数，一般情况下使用有符号数，或者强制转换所有变量为相同的类型，避免将负数解释为超大的数。

## NUL和NULL

NULL是一个宏，它在几个标准头文件中定义，0是一个整型常量，'\0'是一个字符常量，而NUL是一个字符常量的名字。这几个术语都不可互换。

1、NULL用于表示什么也不指向，也就是空指针((void \*)0)

2、0可以被用于任何地方，它是表示各种类型零值的符号并且编译器会挑出它

3、'\0'应该只被用于结束字符串

4、NUL没有被定义于C和C++，它不应该被使用除非你自己定义它，像：#define nul '\0

NULL是在<stddef.h>头文件中专门为空指针定义的一个宏。NUL是ASCII字符集中第一个字符的名称，它对应于一个零值。C语言中没有NUL这样的预定义宏。注意：在ASCII字符集中，数字0对应于十进制值48，不要把数字0和'\0'(NUL)的值混同起来。

NULL可以被定义为(void \*)0，而NUL可以被定义为'\0'。NULL和NUL都可以被简单地定义为0，这时它们是等价的，可以互换使用，但这是一种不可取的方式。为了使程序读起来更清晰，维护起来更容易，你在程序中应该明确地将NULL定义为指针类型，而将NUL定义为字符类型。

对指针进行解引用操作可以获得它的值。从定义来看，NULL指针并未指向任何东西。因此，对一个NULL指针进行解引用操作是非法的。在对指针进行解引用操作之前，必须确保它并非NULL指针。

例如：

#include <stdio.h>

#include <stdlib.h>  
    #define NUL '\0'

int \*ptr = NULL;  
    char ch = NUL;  
\r和\n都是一个字符，分别表示回车和换行，在ASC码里值分别为0x0D、0x0A，windows中文本一行的结尾都是有两个字符\r\n，但是在unix、linux里面只有一个字符\n，Mac里面则是只有字符\r，但是你是用printf("\r");或printf("\n");大部分程序（某些控制台程序除外）都是没有太大区别的。  
至于字符为什么写成\加上0、r和n等，是因为在C语言中表示字符或字符串时遇到\（转义字符）则不会把下一个字符识别为普通字符，比如说'\n'，表示ASC码0x0A，而不是值为0x6E的字母n。

## 字符串连接

相邻的字符串字面值会自动连接为一个字面值，有两种方式，如下：

printf("\

A favorite children's book,\

is 'muffy Gets It: the hilarious tale of a cat,\

a boy, and his machine gun'");

printf("A second favorite children's book is "

"'Thoms the tank engine and Naughty Enginedriver who"

" tied down Thoms's boiler safety value'");

## 运算顺序

.的优先级高于\*；\*p.f表示\*(p.f)，可以用p->f来表达(\*p).f

[]的优先级高于\*；int\*ap[]表示int\*(a[])，即数组元素是int指针。

函数()的优先级高于\*；int\*fp()表示int\*(fp())，表示函数返回值是int指针。

==和！=高于位操作和赋值；(val&mask!=0)表示val&(mask!=0)。c=getchar()!=EOF表示c=(getchar()!=EOF)

算数运算高于位移；msb<<4+lsb表示msb<<(4+lsb))

逗号运算符优先级最低；i=1,2表示(i=1),2，i的值是1，2被丢弃；

x = f() + g() \* h(); f,g,h三个函数的调用顺序并无确定顺序，确定的只有\*优先于+。函数参数调用中的计算顺序是不确定的。

所有的赋值符具有右结合性。位操作符具有左结合性。a = b = c，表示a = (b = c)。逻辑运算符是右结合性，以便更快得出结果，省去后续的计算。

谨记：算数运算符加入()来明确表明优先级，在运算表达式中不依赖于函数调用顺序；

## 关于结构

1、变量的定义和声明应该分开：

2、应该始终在结构定义中加入标签，在定义变量时减少重复编码。

//类型定义和变量声明分开

struct veg

{

int weight, price\_per\_1b;

};

struct veg onion, radish, turnip;

3、结构体在作为函数参数时是拷贝传值。

4、在结构体中定义指向自身的指针来实现链表、树等动态数据结构。

## 关于联合

1. 定义一组不可能同时存在的数据类型的组合；
2. 将同一数据解释成不同形式；

## 关于枚举

1. 缺省值从0开始，逐个增加1；
2. 枚举值类型是整型，且只能作为右值；
3. 枚举类型的变量可以作为左值，且可以被赋值为枚举值之外的值；

## 声明规则

1. 从它的最左边标识符开始读取，然后按照优先级顺序依次读取；
2. 优先级由高到低依次是：
   1. 声明中被括号括起来的那部分
   2. 后缀操作符：()表示这是一个函数，[]表示这是一个数组
   3. 前缀操作符：\*表示指向…的指针，函数的返回值类型，数组的元素类型；
3. 如果const、volatile关键字后面紧跟着类型说明符（如int，long等），那么它作用于类型说明符。在其他情况下，作用于它左边的紧邻的指针\*；

如 char\*const\*(\*next)();表示next是一个指针，指向一个函数，函数的返回值是一个指针，指针的类型是指向char的const指针。

再如char\*(\*c[10])(int \*\*p);表示c是一个数组，数组的类型是指针，指向一个函数，函数的参数是int的指针的指针，返回值是指向char的指针。

## 关于typedef

表示为原来的类型声明一个新的名字，并没有产生一个新的类型。

void ( \* signal **( int sig, void ( \* func ) ( int ) )** ) ( int ).

signal是一个函数，参数之一是一个整数，之二是一个函数指针，参数是证明，无返回值。Signal的返回值是一个指针，指向一个函数，函数的参数是int，无返回值。

typedef void(\*ptr\_to\_func)(int);表示ptr\_to\_func是一个函数指针，函数的参数是int，无返回值。

ptr\_to\_func signal(int sig, ptr\_to\_func);

1. 不要在一个typedef 中放入几个声明器：typedef int \*ptr, (\*fun)(), arr[5];
2. 不要把typedef嵌到声明中的中间部分。unsigned const long typedef int volatile \*kumquat;
3. typedef使用在数组，结构，指针及函数等组合类型、可移植类型；
4. typedef可以用于类型转换

## 关于可变参数宏

C99中规定宏可以像函数一样带有可变参数，比如

#define LOG(format, ...) fprintf(stdout, format, \_\_VA\_ARGS\_\_)

其中，...表示参数可变，\_\_VA\_ARGS\_\_在预处理中为实际的参数集所替换

GCC中同时支持如下的形式

#define LOG(format, args...) fprintf(stdout, format, args)

其用法和上面的基本一致，只是参数符号有变化

有一点需要注意，上述的宏定义不能省略可变参数，尽管你可以传递一个空参数，这里有必要提到"##"连接符号的用法。

"##"的作用是对token进行连接，在上例中，format、\_\_VA\_ARGS\_\_、args即是token，

如果token为空，那么不进行连接，所以允许省略可变参数(\_\_VA\_ARGS\_\_和args)，对上述变参宏做如下修改

#define LOG(format, ...) fprintf(stdout, format, ##\_\_VA\_ARGS\_\_)

#define LOG(format, args...) fprintf(stdout, format, ##args)

gcc用##解决该问题： 如果可变参数列表为空，就会将紧挨着")"的“,”去掉。

## 关于宏

当然宏定义非常重要的，它可以帮助我们防止出错，提高代码的可移植性和可读性等。

下面列举一些成熟软件中常用得宏定义

1，防止一个头文件被重复包含  
#ifndef COMDEF\_H  
#define COMDEF\_H

//头文件内容 …  
#endif

2，重新定义一些类型，防止由于各种平台和编译器的不同，而产生的类型字节数差异，方便移植。  
typedef  unsigned long int  uint32;      /\* Unsigned 32 bit value \*/

3，得到指定地址上的一个字节或字  
#define  MEM\_B( x )  ( \*( (byte \*) (x) ) )  
#define  MEM\_W( x )  ( \*( (word \*) (x) ) )

4，求最大值和最小值  
#define  MAX( x, y )  ( ((x) > (y)) ? (x) : (y) )  
#define  MIN( x, y )  ( ((x) < (y)) ? (x) : (y) )

5，得到一个field在结构体(struct)中的偏移量  
#define FPOS( type, field )   ( (dword) &(( type \*) 0)-> field )

6,得到一个结构体中field所占用的字节数  
#define FSIZ( type, field ) sizeof( ((type \*) 0)->field )

7，按照LSB格式把两个字节转化为一个word  
#define  FLIPW( ray ) ( (((word) (ray)[0]) \* 256) + (ray)[1] )

8，按照LSB格式把一个word转化为两个字节  
#define  FLOPW( ray, val )   
(ray)[0] = ((val) / 256);   
(ray)[1] = ((val) & 0xFF)

9，得到一个变量的地址（word宽度）  
#define  B\_PTR( var )  ( (byte \*) (void \*) &(var) )  
#define  W\_PTR( var )  ( (word \*) (void \*) &(var) )

10，得到一个字的高位和低位字节  
#define  WORD\_LO(xxx)  ((byte) ((word)(var) & 255))  
#define  WORD\_HI(xxx)  ((byte) ((word)(var) >> 8))

11，返回一个比X大的最接近的8的倍数  
#define RND8( x )       ((((x) + 7) / 8 ) \* 8 )

12，将一个字母转换为大写  
#define  UPCASE( c ) ( ((c) >= ’a' && (c) <= ’z') ? ((c) - 0×20) : (c) )

13，判断字符是不是10进值的数字  
#define  DECCHK( c ) ((c) >= ’0′ && (c) <= ’9′)

14，判断字符是不是16进值的数字  
#define  HEXCHK( c ) ( ((c) >= ’0′ && (c) <= ’9′) ||  
((c) >= ’A' && (c) <= ’F') ||  
((c) >= ’a' && (c) <= ’f') )

15，防止溢出的一个方法  
#define  INC\_SAT( val )  (val = ((val)+1 > (val)) ? (val)+1 : (val))

16，返回数组元素的个数  
#define  ARR\_SIZE( a )  ( sizeof( (a) ) / sizeof( (a[0]) ) )

17，对于IO空间映射在存储空间的结构，输入输出处理  
#define inp(port)         (\*((volatile byte \*) (port)))  
#define inpw(port)        (\*((volatile word \*) (port)))  
#define inpdw(port)       (\*((volatile dword \*)(port)))

#define outp(port, val)   (\*((volatile byte \*) (port)) = ((byte) (val)))  
#define outpw(port, val)  (\*((volatile word \*) (port)) = ((word) (val)))  
#define outpdw(port, val) (\*((volatile dword \*) (port)) = ((dword) (val)))

18，使用一些宏跟踪调试  
ANSI标准说明了五个预定义的宏名。它们是：  
\_\_LINE\_\_  
\_\_FILE\_\_  
\_\_DATE\_\_  
\_\_TIME\_\_  
\_\_STDC\_\_

如果编译不是标准的，则可能仅支持以上宏名中的几个，或根本不支持。记住编译程序 也许还提供其它预定义的宏名。  
是行连接符，会将下一行和前一行连接成为一行，即将物理上的两行连接成逻辑上的一行  
\_\_FILE\_\_ 是内置宏 代表源文件的文件名  
\_\_LINE\_\_ 是内置宏，代表该行代码的所在行号  
\_\_DATE\_\_宏指令含有形式为月/日/年的串，表示源文件被翻译到代码时的日期。  
源代码翻译到目标代码的时间作为串包含在\_\_TIME\_\_ 中。串形式为时：分：秒。  
如果实现是标准的，则宏\_\_STDC\_\_含有十进制常量1。如果它含有任何其它数，则实现是非标准的。

可以定义宏，例如:  
当定义了\_DEBUG，输出数据信息和所在文件所在行

#ifdef \_DEBUG  
#define DEBUGMSG(msg,date) printf(msg);printf(“%d%d%d”,date,\_LINE\_,\_FILE\_)  
#else  
#define DEBUGMSG(msg,date)  
#endif

19，宏定义防止使用是错误  
用小括号包含。  
例如：#define ADD(a,b) （a+b）

用do{}while(0)语句包含多语句防止错误

例如：#difne DO(a,b) a+b;  
a++;  
应用时：if(….)  
DO(a,b); //产生错误  
else  
解决方法: #difne DO(a,b) do{a+b;  
a++;}while(0)

为什么需要do{…}while(0)形式?

总结了以下几个原因：

1)，空的宏定义避免warning:

#define foo() do{}while(0)

2)，存在一个独立的block，可以用来进行变量定义，进行比较复杂的实现。

3)，如果出现在判断语句过后的宏，这样可以保证作为一个整体来是实现：

#define foo(x)

action1();

action2();

在以下情况下：

if(NULL == pPointer)

    foo();

就会出现action2必然被执行的情况，而这显然不是程序设计的目的。

4)，以上的第3种情况用单独的{}也可以实现，但是为什么一定要一个do{}while(0)呢，看以下代码：

#define switch(x,y) {int tmp; tmp=x;x=y;y=tmp;}

if(x>y)

   switch(x,y);

else        //error, parse error before else

   otheraction();

在把宏引入代码中，会多出一个分号，从而会报错。

使用do{….}while(0) 把它包裹起来，成为一个独立的语法单元，从而不会与上下文发生混淆。同时因为绝大多数的编译器都能够识别do{…}while(0)这种无用的循环并进行优化，所以使用这种方法也不会导致程序的性能降低。

为什么很多linux内核中宏#defines用do { … } while(0)？

有很多原因：

（Dave Miller的说法）：

编译器对于空语句会给出告警，这是为什么#define FOO do{ }while(0);

给定一个基本块（局部可视域），定义很多局部变量；

（Ben Collins的说法）：

在条件代码中，允许定义复杂的宏。可以想像有很多行宏，如下代码

#define FOO(x)   
printf(“arg is %sn”, x);   
do\_something\_useful(x);  
现在，想像下面的应用:  
if (blah == 2)  
FOO(blah);  
展开后代码为:  
if (blah == 2)  
printf(“arg is %sn”, blah);  
do\_something\_useful(blah);;  
就像你看到的，if仅仅包含了printf（），而do\_something\_useful()调用是无条件调用。因此，如果用do { … } while(0)，结果是：  
if (blah == 2)  
do {  
printf(“arg is %sn”, blah);  
do\_something\_useful(blah);  
} while (0);  
这才是所期望的结果。  
（Per Persson的说法）：  
像 Miller and Collins指出的那样，需要一个块语句包含多个代码行和声明局部变量。但是，本质如下面例子代码：  
#define exch(x,y) { int tmp; tmp=x; x=y; y=tmp; }  
上面代码在有些时候却不能有效工作，下面代码是一个有两个分支的if语句：  
if (x > y)  
exch(x,y);               // Branch 1  
else  
do\_something();      // Branch 2  
展开后代码如下：  
if (x > y)

{                // Single-branch if-statement!!!  
int tmp;            // The one and only branch consists  
tmp = x;           // of the block.  
x = y;  
y = tmp;  
}  
;                // empty statement  
else                             // ERROR!!! “parse error before else”  
do\_something();  
问题是分号（；）出现在块后面。解决这个问题可以用do{}while（0）：  
if (x > y)  
do {  
int tmp;  
tmp = x;  
x = y;  
y = tmp;  
} while(0);  
else  
do\_something();  
（ Bart Trojanowski的说法）：  
Gcc加入了语句解释，它提供了一个替代do-while-0块的方法。对于上面的解决方法如下，并且更加符合常理  
#define FOO(arg) ({   
typeof(arg) lcl;    
lcl = bar(arg);     
lcl;                    
})  
这是一个奇怪的循环，它根本就只会运行一次，为什么不去掉外面的do{..}while结构呢？我曾一度在心里把它叫做“怪圈”。原来这也是非常巧妙的技巧。在工程中可能经常会引起麻烦，而上面的定义能够保证这些麻烦不会出现。下面是解释：  
假设有这样一个宏定义  
#define macro(condition)   
if(condition) dosomething()  
现在在程序中这样使用这个宏：  
if(temp)  
macro(i);  
else  
doanotherthing();  
一切看起来很正常，但是仔细想想。这个宏会展开成：  
if(temp)  
if(condition) dosomething();  
else  
doanotherthing();  
这时的else不是与第一个if语句匹配，而是错误的与第二个if语句进行了匹配，编译通过了，但是运行的结果一定是错误的。  
为了避免这个错误，我们使用do{….}while(0) 把它包裹起来，成为一个独立的语法单元，从而不会与上下文发生混淆。同时因为绝大多数的编译器都能够识别do{…}while(0)这种无用的循环并进行优化，所以使用这种方法也不会导致程序的性能降低。

另一个讲解  
这是为了含多条语句的宏的通用性  
因为默认规则是宏定义最后是不能加分号的，分号是在引用的时候加上的  
比如定义了一个宏fw(a,b)，那么在c文件里一定是这样引用  
fw(a,b);  
如果不用do…while，那么fw就得定义成:  
#define fw(a,b) {read((a));write((b));}  
那这样fw(a,b);展开后就成了:  
{read(a);write(b);};  
最后就多了个分号，这是语法错误  
而定义成do…while的话，展开后就是:  
do{read(a);write(b);}while(0);    完全正确  
所以要写一个包含多条语句的宏的话，不用do…while是不可能的

宏中#和##的用法

一、一般用法  
我们使用#把宏参数变为一个字符串,用##把两个宏参数贴合在一起.  
用法:  
＃include<cstdio>  
＃include<climits>  
using namespace std;

#define STR(s)     #s  
#define CONS(a,b)  int(a##e##b)

int main()

{

printf(STR(vck));               // 输出字符串vck  
printf(%dn, CONS(2,3));  // 2e3 输出:2000  
return 0;  
}

二、当宏参数是另一个宏的时候  
需要注意的是凡宏定义里有用’#'或’##’的地方宏参数是不会再展开.

1, 非’#'和’##’的情况  
#define TOW      (2)  
#define MUL(a,b) (a\*b)

printf(%d\*%d=%dn, TOW, TOW, MUL(TOW,TOW));  
这行的宏会被展开为：  
printf(%d\*%d=%dn, (2), (2), ((2)\*(2)));  
MUL里的参数TOW会被展开为(2).

2, 当有’#'或’##’的时候  
#define A          (2)  
#define STR(s)     #s  
#define CONS(a,b)  int(a##e##b)

printf(“int max: %sn”,  STR(INT\_MAX));    // INT\_MAX ＃include<climits>  
这行会被展开为：  
printf(“int max: %sn”, #INT\_MAX);

printf(%sn, CONS(A, A));               // compile error  
这一行则是：  
printf(%sn, int(AeA));

INT\_MAX和A都不会再被展开, 然而解决这个问题的方法很简单. 加多一层中间转换宏.  
加这层宏的用意是把所有宏的参数在这层里全部展开, 那么在转换宏里的那一个宏(\_STR)就能得到正确的宏参数.

#define A           (2)  
#define \_STR(s)     #s  
#define STR(s)      \_STR(s)                 // 转换宏  
#define \_CONS(a,b)  int(a##e##b)  
#define CONS(a,b)   \_CONS(a,b)       // 转换宏

printf(int max: %sn, STR(INT\_MAX));          // INT\_MAX，int型的最大值，为一个变量 ＃include<climits>  
输出为: int max: 0x7fffffff  
STR(INT\_MAX) –>  \_STR(0x7fffffff) 然后再转换成字符串；

printf(%dn, CONS(A, A));  
输出为：200  
CONS(A, A)  –>  \_CONS((2), (2))  –> int((2)e(2))

三、’#'和’##’的一些应用特例  
1、合并匿名变量名  
#define  \_\_ANONYMOUS1(type, var, line)  type  var##line  
#define  \_ANONYMOUS0(type, line)  \_\_ANONYMOUS1(type, \_anonymous, line)  
#define  ANONYMOUS(type)  \_ANONYMOUS0(type, \_\_LINE\_\_)  
例：ANONYMOUS(static int);  即: static int \_anonymous70;  70表示该行行号；  
第一层：ANONYMOUS(static int);  –>  \_\_ANONYMOUS0(static int, \_\_LINE\_\_);  
第二层：–>  \_\_\_ANONYMOUS1(static int, \_anonymous, 70);  
第三层：–>  static int  \_anonymous70;  
即每次只能解开当前层的宏，所以\_\_LINE\_\_在第二层才能被解开；

2、填充结构  
#define  FILL(a)   {a, #a}

enum IDD{OPEN, CLOSE};  
typedef struct MSG{  
IDD id;  
const char \* msg;  
}MSG;

MSG \_msg[] = {FILL(OPEN), FILL(CLOSE)};  
相当于：  
MSG \_msg[] = {{OPEN, “OPEN”},  
{CLOSE, ”CLOSE“}};

3、记录文件名  
#define  \_GET\_FILE\_NAME(f)   #f  
#define  GET\_FILE\_NAME(f)    \_GET\_FILE\_NAME(f)  
static char  FILE\_NAME[] = GET\_FILE\_NAME(\_\_FILE\_\_);

4、得到一个数值类型所对应的字符串缓冲大小  
#define  \_TYPE\_BUF\_SIZE(type)  sizeof #type  
#define  TYPE\_BUF\_SIZE(type)   \_TYPE\_BUF\_SIZE(type)  
char  buf[TYPE\_BUF\_SIZE(INT\_MAX)];  
–>  char  buf[\_TYPE\_BUF\_SIZE(0x7fffffff)];  
–>  char  buf[sizeof 0x7fffffff];  
这里相当于：  
char  buf[11];

## 关于定义和声明

无论定义是什么样的，程序总是按照声明的形式进行解释执行。比如：

A文件中，定义了数组int a[4];

B文件中声明为extern int \*a; 在B文件中访问a时，那么无论是\*a的方式，还是a[0]的方式，都是会把实际a的值(a存储的值，以&a[0]为地址的四字节长度的地址值)作为地址，再把地址中的值取出来访问。

所以，实际编写代码时，必须保持变量的定义和声明是一致的，实际上，如果将数据封装在文件内部，不对外暴露内部的数据细节，就不会有这样的问题了。

## 链接

### 动态链接

目的是ABI(Application Binary Interface)，介于应用程序和函数库二进制可执行文件所提供的服务之间的接口，该接口保持稳定。

**例子：**

有如下文件，lib目录下是准备好的动态库的源文件，根目录下是使用动态库的源文件：

.

├── lib

│   ├── libfruit.h

│   └── tomato.c

└── test.c

编译过程：

1、进入lib目录下，使用如下命令进行编译动态库：gcc tomato.c -fPIC -shared -o libfruit.so

-shared该选项指定生成动态连接库（让连接器生成T类型的导出符号表，有时候也生成弱连接W类型的导出符号），不用该标志外部程序无法连接。相当于一个可执行文件

-fPIC：表示编译为位置独立的代码，不用此选项的话编译后的代码是位置相关的所以动态载入时是通过代码拷贝的方式来满足不同进程的需要，而不能达到真正代码段共享的目的。

2、进入根目录下，使用如下命令进行编译：gcc test.c -L./lib -Wl,-rpath ./lib -lfruit -I./lib -o test

-L.：表示要连接的库在当前目录中

-lfruit：编译器查找动态连接库时有隐含的命名规则，即在给出的名字前面加上lib，后面加上.so来确定库的名称

-I：表示包含头文件的路径

-Wl,-rpath，指定编译好的程序在运行时动态库的目录。这种方法会将动态库路径写入到elf文件中去。-Wl, 表示后面的参数将传给 link 程序 ld （因为 gcc 可能会自动调用 ld ）。这里通过 gcc 的参数 "-Wl,-rpath" 指定。当指定多个动态库搜索路径时，路径之间用冒号 " ： " 分隔。注：也可以通过export LD\_LIBRARY\_PATH，但是会影响到其他程序的运行。当然如果有root权限的话，也可以修改/etc/ld.so.conf文件，将要添加的动态库搜索路径写入该文件中，然后调用/sbin/ldconfig来达到同样的目的。

3、通过命令ldd test来查看是否可以列出动态链接的库，如果可以看到libfruit.so则说明编译没有问题。

### 静态链接

**例子：**

有如下文件，lib目录下是准备好的动态库的源文件，根目录下是使用动态库的源文件：

.

├── lib

│   ├── libfruit.h

│   └── tomato.c

└── test.c

编译过程：

1. 进入lib目录下，使用如下命令进行编译目标文件：gcc -c tomato.c -o tomato.o
2. 继续使用如下命令编译生成静态库：ar rc libfruit.a tomato.o

该命令将hello.o添加到静态库文件libhello.a，ar命令就是用来创建、修改库的，也可以从库中提出单个模块，参数r表示在库中插入或者替换模块，c表示创建一个库，关于ar命令的详细使用规则可以参考文章http://blog.csdn.net/xuhongning/article/details/6365200。这一步将在libtest/lib目录中生成一个libhello.a文件。

3、进入根目录下，使用如下命令进行编译：gcc test.c -I./lib -L./lib -lfruit -static -o test1

-L.：表示要连接的库在当前目录中

-lfruit：编译器查找静态连接库时有隐含的命名规则，即在给出的名字前面加上lib，后面加上.a来确定库的名称

-I：表示包含头文件的路径

注：如果在-L的目录下有同名的动态库，那么会优先进行动态链接。此时，执行程序会找不到库，因为没有指定运行目录。

### 链接顺序

1. 在编译命令中，各个静态库的顺序很重要，所以建议把所有编译的源文件放在前面，而-l命令放在最后，例子：

gcc -lm main.c

/tmp/ccVKYhEq.o: In function `main':

main.c:(.text+0x31): undefined reference to `sin'

collect2: error: ld returned 1 exit status

1. 链接程序在链接时一般是优先链接动态库的，即在库的目录下有两个同名的库文件libxxx.a和libxxx.so，肯定链接的是libxxx.so，除非用-static参数指定链接静态库。
2. 静态库链接时搜索路径顺序：
   1. ld会去找GCC命令中的参数-L
   2. 再找gcc的环境变量LIBRARY\_PATH
   3. 再找内定目录 /lib /usr/lib /usr/local/lib 这是当初compile gcc时写在程序内的
3. 动态链接时、执行时搜索路径顺序:
   1. 编译目标代码时指定的动态库搜索路径；
   2. 环境变量LD\_LIBRARY\_PATH指定的动态库搜索路径；
   3. 配置文件/etc/ld.so.conf中指定的动态库搜索路径；
   4. 默认的动态库搜索路径/lib；
   5. 默认的动态库搜索路径/usr/lib。
4. 当对动态库与静态库混合连接的时候，使用-static会导致所有的库都使用静态连接的方式。这时需要作用-Wl的方式：

gcc test.c -L./lib -I./lib -Wl,-Bstatic -lfruit **-Wl,-Bdynamic -Wl,-rpath ./lib** –lmeat，粗体部分可以写成**-Wl,-Bdynamic,-rpath ./lib，具体原因见下面的参数说明**

另外还要注意系统的运行库使用动态连接的方式，所以当动态库在静态库前面连接时，必须在命令行最后使用动态连接的命令才能正常连接，如：

gcc test.c -L./lib -I./lib -Wl,-Bdynamic,-rpath ./lib -lmeat -Wl,-Bstatic -lfruit -Wl,-Bdynamic

最后的-Wl,-Bdynamic表示将缺省库链接模式恢复成动态链接。否则就会找不到库：

/usr/bin/ld: cannot find -lgcc\_s

/usr/bin/ld: cannot find -lgcc\_s

collect2: error: ld returned 1 exit status

**参数说明：**

其中用到的两个选项：-Wl,-Bstatic和-Wl,-Bdynamic。这两个选项是gcc的特殊选项，它会将选项的参数传递给链接器，作为链接器的选项。比如-Wl,-Bstatic告诉链接器使用-Bstatic选项，该选项是告诉链接器，对接下来的-l选项使用静态链接；-Wl,-Bdynamic就是告诉链接器对接下来的-l选项使用动态链接。下面是man gcc对-Wl,option的描述，

-Wl,option Pass option as an option to the linker. If option contains commas, it is split into multiple options at the commas. You can use this syntax to pass an argument to the option. For example, -Wl,-Map,output.map passes -Map output.map to the linker. When using the GNU linker, you can also get the same effect with -Wl,-Map=output.map.

下面是man ld分别对-Bstatic和-Bdynamic的描述，

-Bdynamic -dy -call\_shared Link against dynamic libraries. You may use this option multiple times on the command line: it affects library searching for -l options which follow it. -Bstatic -dn -non\_shared -static Do not link against shared libraries. You may use this option multiple times on the command line: it affects library searching for -l options which follow it. This option also implies --unresolved-symbols=report-all. This option can be used with -shared. Doing so means that a shared library is being created but that all of the library's external references must be resolved by pulling in entries from static libraries

### 关于interpositioning

也称为interposing，即通过编写与库函数同名的函数来取代库函数的行为。所以，不要让程序中的任何符号成为全局的，除非有意的把它们作为程序的接口之一。

## 关于a.out

a.out是assembler output（汇编程序输出）的缩写。文件以7F开头，紧跟在后面的第二个到第四个字节为“ELF”。ELF原意为“Extensible Linker Format”，可扩展连接格式。现在代表“Excutable and Linking Format”，可执行文件和连接格式。

Segment，段表示一个二进制文件相关的内容块，里面保存了和某种特定类型（如符号表条目）相关的所有信息。

Section是ELF文件中的最小组织单位，一个段一般包含几个section。

例子程序：

#include <stdlib.h>

char pear[40];

static double peach;

int mango = 13;

static long melon = 2001;

void main()

{

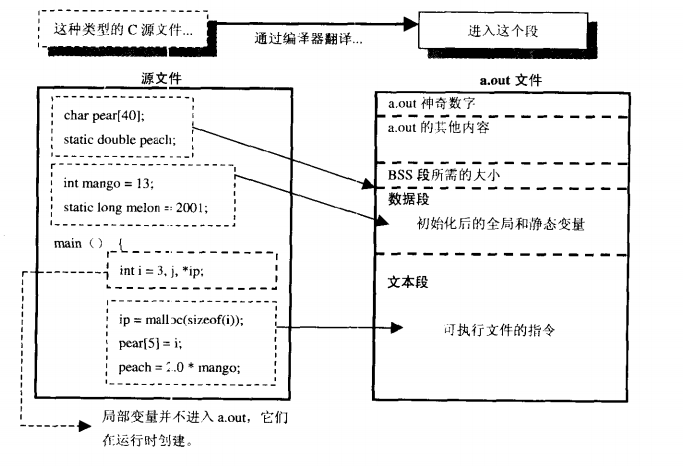
int i = 3, j, \*ip;

ip = malloc(sizeof(i));

pear[5] = i;

peach = 2.0 \* mango;

}



Bss段：“Block Stated by Symbol”由符号开始的块，它是旧式IBM704汇编程序的一个伪指令，Unix就用这个名字，沿用至今。BSS段只保存没有值的全局变量，所以它并不需要保存这些变量的映像。运行时所需要的BSS段的大小记录在目标文件中，但BSS段（不像其他段）并不占据目标文件的任何空间。

### 查看a.out文件的命令：

#### size

使用size命令可以查看可执行文件中的三个段的大小：

**$ size a.out**

text data bss dec hex filename

1473 312 72 1857 741 a.out

#### nm

Nm命令中，b表示私有（static），B表示全局的。

**$ nm -S a.out**

00002010 B \_\_bss\_start

00002020 00000001 b completed.7217

w \_\_cxa\_finalize@@GLIBC\_2.1.3

00002000 D \_\_data\_start

00002000 W data\_start

00000430 t deregister\_tm\_clones

000004c0 t \_\_do\_global\_dtors\_aux

00001edc t \_\_do\_global\_dtors\_aux\_fini\_array\_entry

00002004 D \_\_dso\_handle

00001ee0 d \_DYNAMIC

00002010 D \_edata

00002068 B \_end

000005e4 T \_fini

000005f8 00000004 R \_fp\_hw

00000510 t frame\_dummy

00001ed8 t \_\_frame\_dummy\_init\_array\_entry

00000718 r \_\_FRAME\_END\_\_

00001fd8 d \_GLOBAL\_OFFSET\_TABLE\_

w \_\_gmon\_start\_\_

00000600 r \_\_GNU\_EH\_FRAME\_HDR

0000037c T \_init

00001edc t \_\_init\_array\_end

00001ed8 t \_\_init\_array\_start

000005fc 00000004 R \_IO\_stdin\_used

w \_ITM\_deregisterTMCloneTable

w \_ITM\_registerTMCloneTable

000005e0 00000002 T \_\_libc\_csu\_fini

00000580 0000005d T \_\_libc\_csu\_init

U \_\_libc\_start\_main@@GLIBC\_2.0

0000051d 00000061 T main

U malloc@@GLIBC\_2.0

00002008 00000004 D mango

0000200c 00000004 d melon

00002028 00000008 b peach

00002040 00000028 B pear

00000470 t register\_tm\_clones

000003e0 T \_start

00002010 D \_\_TMC\_END\_\_

00000420 00000004 T \_\_x86.get\_pc\_thunk.bx

00000519 T \_\_x86.get\_pc\_thunk.dx

#### readelf

用readelf -s 或 objdump -t 查看符号表  
用readelf -S 或 objdump -h 查看段表

**$ readelf -s a.out**

Symbol table '.dynsym' contains 8 entries:

Num: Value Size Type Bind Vis Ndx Name

0: 00000000 0 NOTYPE LOCAL DEFAULT UND

1: 00000000 0 NOTYPE WEAK DEFAULT UND \_ITM\_deregisterTMCloneTab

2: 00000000 0 FUNC WEAK DEFAULT UND \_\_cxa\_finalize@GLIBC\_2.1.3 (2)

3: 00000000 0 FUNC GLOBAL DEFAULT UND malloc@GLIBC\_2.0 (3)

4: 00000000 0 NOTYPE WEAK DEFAULT UND \_\_gmon\_start\_\_

5: 00000000 0 FUNC GLOBAL DEFAULT UND \_\_libc\_start\_main@GLIBC\_2.0 (3)

6: 00000000 0 NOTYPE WEAK DEFAULT UND \_ITM\_registerTMCloneTable

7: 000005fc 4 OBJECT GLOBAL DEFAULT 16 \_IO\_stdin\_used

Symbol table '.symtab' contains 70 entries:

Num: Value Size Type Bind Vis Ndx Name

0: 00000000 0 NOTYPE LOCAL DEFAULT UND

1: 00000154 0 SECTION LOCAL DEFAULT 1

2: 00000168 0 SECTION LOCAL DEFAULT 2

3: 00000188 0 SECTION LOCAL DEFAULT 3

4: 000001ac 0 SECTION LOCAL DEFAULT 4

5: 000001cc 0 SECTION LOCAL DEFAULT 5

6: 0000024c 0 SECTION LOCAL DEFAULT 6

7: 000002ea 0 SECTION LOCAL DEFAULT 7

8: 000002fc 0 SECTION LOCAL DEFAULT 8

9: 0000032c 0 SECTION LOCAL DEFAULT 9

10: 0000036c 0 SECTION LOCAL DEFAULT 10

11: 0000037c 0 SECTION LOCAL DEFAULT 11

12: 000003a0 0 SECTION LOCAL DEFAULT 12

13: 000003d0 0 SECTION LOCAL DEFAULT 13

14: 000003e0 0 SECTION LOCAL DEFAULT 14

15: 000005e4 0 SECTION LOCAL DEFAULT 15

16: 000005f8 0 SECTION LOCAL DEFAULT 16

17: 00000600 0 SECTION LOCAL DEFAULT 17

18: 00000634 0 SECTION LOCAL DEFAULT 18

19: 00001ed8 0 SECTION LOCAL DEFAULT 19

20: 00001edc 0 SECTION LOCAL DEFAULT 20

21: 00001ee0 0 SECTION LOCAL DEFAULT 21

22: 00001fd8 0 SECTION LOCAL DEFAULT 22

23: 00002000 0 SECTION LOCAL DEFAULT 23

24: 00002020 0 SECTION LOCAL DEFAULT 24

25: 00000000 0 SECTION LOCAL DEFAULT 25

26: 00000000 0 FILE LOCAL DEFAULT ABS crtstuff.c

27: 00000430 0 FUNC LOCAL DEFAULT 14 deregister\_tm\_clones

28: 00000470 0 FUNC LOCAL DEFAULT 14 register\_tm\_clones

29: 000004c0 0 FUNC LOCAL DEFAULT 14 \_\_do\_global\_dtors\_aux

30: 00002020 1 OBJECT LOCAL DEFAULT 24 completed.7217

31: 00001edc 0 OBJECT LOCAL DEFAULT 20 \_\_do\_global\_dtors\_aux\_fin

32: 00000510 0 FUNC LOCAL DEFAULT 14 frame\_dummy

33: 00001ed8 0 OBJECT LOCAL DEFAULT 19 \_\_frame\_dummy\_init\_array\_

34: 00000000 0 FILE LOCAL DEFAULT ABS test.c

35: 00002028 8 OBJECT LOCAL DEFAULT 24 peach

36: 0000200c 4 OBJECT LOCAL DEFAULT 23 melon

37: 00000000 0 FILE LOCAL DEFAULT ABS crtstuff.c

38: 00000718 0 OBJECT LOCAL DEFAULT 18 \_\_FRAME\_END\_\_

39: 00000000 0 FILE LOCAL DEFAULT ABS

40: 00001edc 0 NOTYPE LOCAL DEFAULT 19 \_\_init\_array\_end

41: 00001ee0 0 OBJECT LOCAL DEFAULT 21 \_DYNAMIC

42: 00001ed8 0 NOTYPE LOCAL DEFAULT 19 \_\_init\_array\_start

43: 00000600 0 NOTYPE LOCAL DEFAULT 17 \_\_GNU\_EH\_FRAME\_HDR

44: 00001fd8 0 OBJECT LOCAL DEFAULT 22 \_GLOBAL\_OFFSET\_TABLE\_

45: 000005e0 2 FUNC GLOBAL DEFAULT 14 \_\_libc\_csu\_fini

46: 00002040 40 OBJECT GLOBAL DEFAULT 24 pear

47: 00000000 0 NOTYPE WEAK DEFAULT UND \_ITM\_deregisterTMCloneTab

48: 00000420 4 FUNC GLOBAL HIDDEN 14 \_\_x86.get\_pc\_thunk.bx

49: 00002000 0 NOTYPE WEAK DEFAULT 23 data\_start

50: 00002010 0 NOTYPE GLOBAL DEFAULT 23 \_edata

51: 000005e4 0 FUNC GLOBAL DEFAULT 15 \_fini

52: 00000519 0 FUNC GLOBAL HIDDEN 14 \_\_x86.get\_pc\_thunk.dx

53: 00000000 0 FUNC WEAK DEFAULT UND \_\_cxa\_finalize@@GLIBC\_2.1

54: 00000000 0 FUNC GLOBAL DEFAULT UND malloc@@GLIBC\_2.0

55: 00002000 0 NOTYPE GLOBAL DEFAULT 23 \_\_data\_start

56: 00000000 0 NOTYPE WEAK DEFAULT UND \_\_gmon\_start\_\_

57: 00002004 0 OBJECT GLOBAL HIDDEN 23 \_\_dso\_handle

58: 000005fc 4 OBJECT GLOBAL DEFAULT 16 \_IO\_stdin\_used

59: 00002008 4 OBJECT GLOBAL DEFAULT 23 mango

60: 00000000 0 FUNC GLOBAL DEFAULT UND \_\_libc\_start\_main@@GLIBC\_

61: 00000580 93 FUNC GLOBAL DEFAULT 14 \_\_libc\_csu\_init

62: 00002068 0 NOTYPE GLOBAL DEFAULT 24 \_end

63: 000003e0 0 FUNC GLOBAL DEFAULT 14 \_start

64: 000005f8 4 OBJECT GLOBAL DEFAULT 16 \_fp\_hw

65: 00002010 0 NOTYPE GLOBAL DEFAULT 24 \_\_bss\_start

66: 0000051d 97 FUNC GLOBAL DEFAULT 14 main

67: 00002010 0 OBJECT GLOBAL HIDDEN 23 \_\_TMC\_END\_\_

68: 00000000 0 NOTYPE WEAK DEFAULT UND \_ITM\_registerTMCloneTable

69: 0000037c 0 FUNC GLOBAL DEFAULT 11 \_init

**$ readelf -S a.out**

There are 29 section headers, starting at offset 0x17dc:

Section Headers:

[Nr] Name Type Addr Off Size ES Flg Lk Inf Al

[ 0] NULL 00000000 000000 000000 00 0 0 0

[ 1] .interp PROGBITS 00000154 000154 000013 00 A 0 0 1

[ 2] .note.ABI-tag NOTE 00000168 000168 000020 00 A 0 0 4

[ 3] .note.gnu.build-i NOTE 00000188 000188 000024 00 A 0 0 4

[ 4] .gnu.hash GNU\_HASH 000001ac 0001ac 000020 04 A 5 0 4

[ 5] .dynsym DYNSYM 000001cc 0001cc 000080 10 A 6 1 4

[ 6] .dynstr STRTAB 0000024c 00024c 00009d 00 A 0 0 1

[ 7] .gnu.version VERSYM 000002ea 0002ea 000010 02 A 5 0 2

[ 8] .gnu.version\_r VERNEED 000002fc 0002fc 000030 00 A 6 1 4

[ 9] .rel.dyn REL 0000032c 00032c 000040 08 A 5 0 4

[10] .rel.plt REL 0000036c 00036c 000010 08 AI 5 22 4

[11] .init PROGBITS 0000037c 00037c 000023 00 AX 0 0 4

[12] .plt PROGBITS 000003a0 0003a0 000030 04 AX 0 0 16

[13] .plt.got PROGBITS 000003d0 0003d0 000010 08 AX 0 0 8

[14] .text PROGBITS 000003e0 0003e0 000202 00 AX 0 0 16

[15] .fini PROGBITS 000005e4 0005e4 000014 00 AX 0 0 4

[16] .rodata PROGBITS 000005f8 0005f8 000008 00 A 0 0 4

[17] .eh\_frame\_hdr PROGBITS 00000600 000600 000034 00 A 0 0 4

[18] .eh\_frame PROGBITS 00000634 000634 0000e8 00 A 0 0 4

[19] .init\_array INIT\_ARRAY 00001ed8 000ed8 000004 04 WA 0 0 4

[20] .fini\_array FINI\_ARRAY 00001edc 000edc 000004 04 WA 0 0 4

[21] .dynamic DYNAMIC 00001ee0 000ee0 0000f8 08 WA 6 0 4

[22] .got PROGBITS 00001fd8 000fd8 000028 04 WA 0 0 4

[23] .data PROGBITS 00002000 001000 000010 00 WA 0 0 4

[24] .bss NOBITS 00002020 001010 000048 00 WA 0 0 32

[25] .comment PROGBITS 00000000 001010 000023 01 MS 0 0 1

[26] .symtab SYMTAB 00000000 001034 000460 10 27 45 4

[27] .strtab STRTAB 00000000 001494 00024a 00 0 0 1

[28] .shstrtab STRTAB 00000000 0016de 0000fc 00 0 0 1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), I (info),

L (link order), O (extra OS processing required), G (group), T (TLS),

C (compressed), x (unknown), o (OS specific), E (exclude),

p (processor specific)

### 局部变量对各段的影响

先把前面例子中的a.out文件信息和size信息打印出来：

-rwxrwxr-x 1 jason jason 7268 Nov 3 14:04 a.out\*

**$ size a.out**

text data bss dec hex filename

1473 312 72 1857 741 a.out

按照下面表格对代码进行修改，并观察：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 修改 | a.out大小 | 变化 | bss | 变化 | data | 变化 | text | 变化 | sum | 变化 |
| 无 | 7268 |  | 72 |  | 312 |  | 1473 |  | 1857 |  |
| int a[1000000]; | 7352 | 84 | 72 | 0 | 316 | 4 | 1676 | 203 | 2064 | 207 |
| int a[1000000] = {0}; | 7384 | 32 | 72 | 0 | 320 | 4 | 1757 | 81 | 2149 | 85 |
| int a[1000000] = {0}; int b[1000000]; | 7384 | 0 | 72 | 0 | 320 | 0 | 1757 | 0 | 2149 | 0 |
| int a[1000000] = {0}; int b[1000000] = {0}; | 7384 | 0 | 72 | 0 | 320 | 0 | 1773 | 16 | 2165 | 16 |
| int a[1000000] = {0}; int b[1000000] = {0}; int c[1000000]; | 7384 | 0 | 72 | 0 | 320 | 0 | 1773 | 0 | 2165 | 0 |
| int a[1000000] = {0}; int b[1000000] = {0}; int c[1000001] = {0}; | 7384 | 0 | 72 | 0 | 320 | 0 | 1805 | 32 | 2197 | 32 |
| int a[1000000] = {0}; int b[1000000] = {0}; int c[1000001] = {0}; static int x; | 7408 | 24 | 72 | 0 | 320 | 0 | 1805 | 0 | 2197 | 0 |
| int a[1000000] = {0}; int b[1000000] = {0}; int c[1000001] = {0}; static int x,y; | 7432 | 24 | 72 | 0 | 320 | 0 | 1805 | 0 | 2197 | 0 |
| int a[1000000] = {0}; int b[1000000] = {0}; int c[1000001] = {0}; static int x,y,z=0; | 7452 | 20 | 72 | 0 | 320 | 0 | 1805 | 0 | 2197 | 0 |
| int a[1000000] = {0}; int b[1000000] = {0}; int c[1000001] = {0}; static int x,y,z=1; | 7456 | 4 | 72 | 0 | 324 | 4 | 1805 | 0 | 2201 | 4 |

结论：

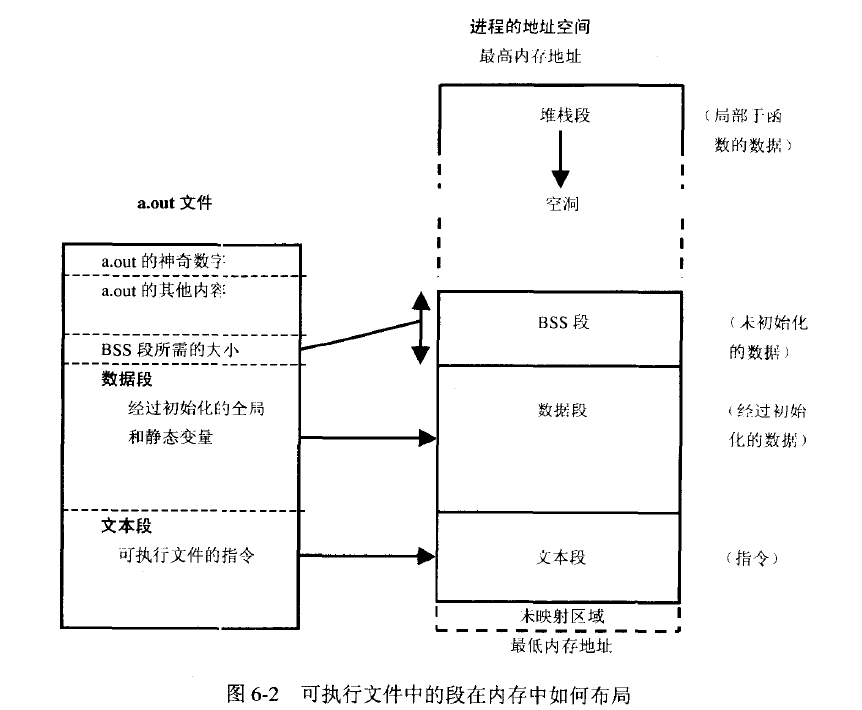
1. bss段只记录无初值全局变量，局部变量不影响bss，且bss段不占用a.out空间，只记录大小。
2. 数据段只记录有初值全局变量，局部变量不影响data段，data段占用a.out空间。
3. 局部变量的变化会影响text段，因为属于运行时的数据。
4. text段是最容易受优化措施影响的段。
5. a.out文件的大小受调试状态下编译的影响(gcc -g)，但是段不受影响。

## 运行时数据

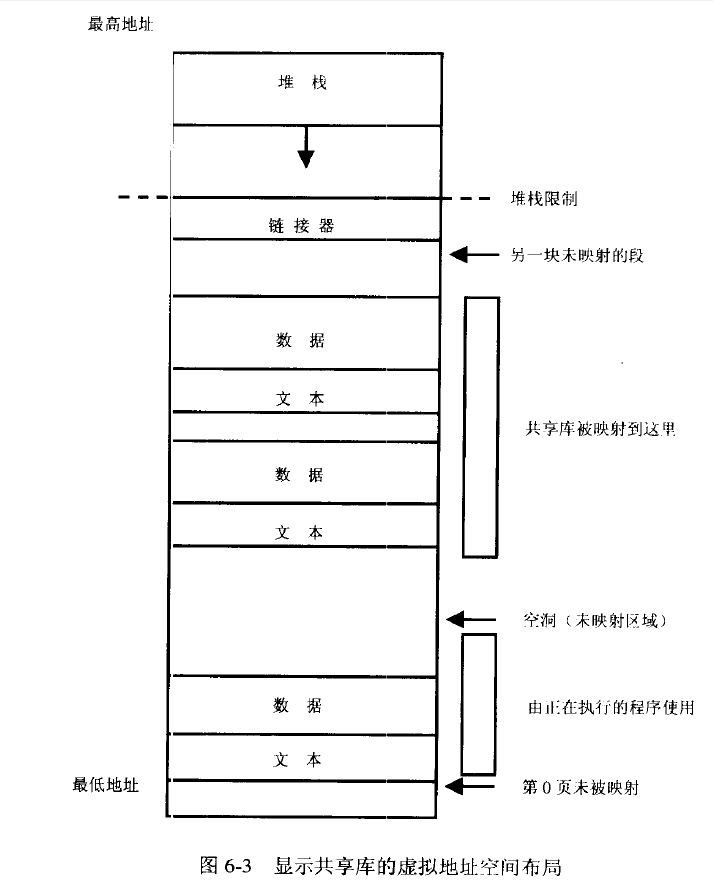
### 操作系统对a.out的操作

段在正在执行的程序中是一块内存区域，每个区域都有特定的目的。载入器只是取文件中每个段的映像，并直接将它们放入内存中。

文本段包含了程序的指令。数据段包含经过初始化的全局和静态变量以及它的值。BSS段的大小可以从文件中得到，链接器得到这个大小的内存块紧跟在数据段之后，该内存块的地址空间全部清零。数据段和BSS段统称为数据区。还需要堆栈段保存局部变量、临时变量、传递到函数的参数等。还需要堆空间用于动态申请的内存。虚拟地址空间的最低部分未被映射，任何对它的引用都是非法的。典型的，它是地址零开始的几K字节，它用于捕捉用于空指针和小整型值的指针引用内存的情况。



当考虑共享库的时，进程的地址空间如下：



### C语言运行时系统对a.out的操作

运行时数据结构有好几种：堆栈，活动记录（activation record），数据，堆等。

#### 堆栈段

堆栈段包含一种单一的数据结构——堆栈，它是一块动态内存区域，实现了一种后进先出的结构。函数可以通过参数或者全局指针访问它所调用的局部变量。运行时系统维护一个指针（常位于寄存器中），通常成为sp(stack pointer)，用于提示栈当前的顶部位置。

堆栈段有三个用途;

1. 堆栈为函数内部声明的局部变量提供存储空间，这些变量亦被称为自动变量。
2. 进行函数调用时，堆栈存储与此有关的一些维护信息。这些信息被称为堆栈结构（stack frame），或被称为过程活动记录（procedure activation record），它包括函数调用的地址（即所调用的函数结束后跳回的地方）、任何不适合装入寄存器的参数以及一些寄存器值得保存。
3. 堆栈也可被称为暂时存储区。有时需要一些临时存储区，比如计算一个很长的算术表达式时，可以把部分结果压入到栈中，当需要时再把它取出来。

在绝大部分的处理器中，堆栈是向下增长的，也即是朝着低地址的方向生长。如下面程序例子：

#include <stdio.h>

void f2();

int j;

int k = 10;

void f1()

{

int i[100000];

printf("The stack f1 is near %p.\n", &i);

f2();

}

void f2()

{

int i[100000];

printf("The stack f2 is near %p.\n", &i);

}

void main()

{

int i;

printf("The stack top is near %p.\n", &i);

f1();

printf("The bss addr is near %p.\n", &j);

printf("The data addr is near %p.\n", &k);

printf("The text addr is near %p.\n", &f1);

}

The stack top is near 0x7ffd90a6c9e4.

The stack f1 is near 0x7ffd90a0af40.

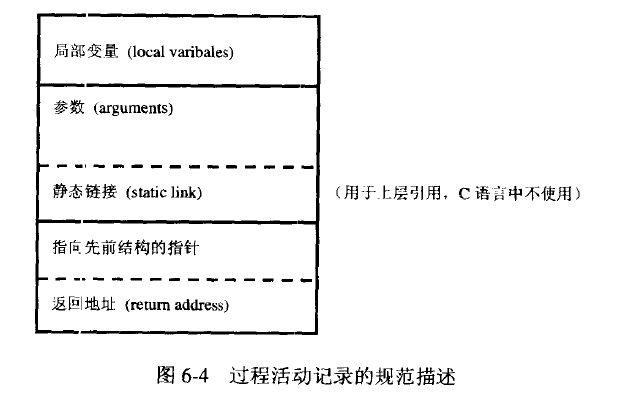
The stack f2 is near 0x7ffd909a94a0.

The bss addr is near 0x601048.

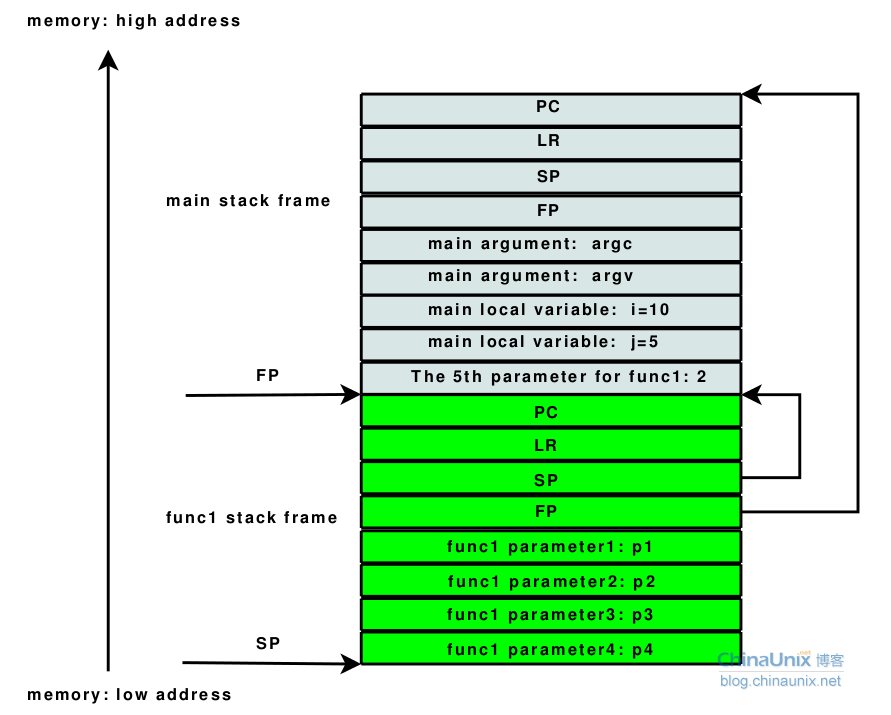
The data addr is near 0x601040.

The text addr is near 0x400596.

C语言提供了跟踪函数调用链，当下一个return语句执行时，控制将返回何处。当每个函数调用时，都会产生一个过程活动记录（或类似的结构），结构的具体细节在不同的编译器中各不相同，这些字段的次序很可能不相同，而且还可能存在一个调用函数前保存寄存器值得区域。运行时系统维护一个指针（常常位于寄存器中），通常称为fp(frame pointer)，用于提示活动堆栈结构。它的值是最靠近堆栈顶部的过程活动记录的地址。



每个函数所使用的栈空间是一个栈帧，所有的栈帧就组成了这个进程完整的栈。而fp就是栈基址寄存器，指向当前函数栈帧的栈底，sp则指向当前函数栈帧的栈顶。通过sp和fp所指出的栈帧可以恢复出母函数的栈帧，以此类推就可以backtrace出所有函数的调用顺序。



上图描述的是ARM的栈帧布局方式，main stack frame为调用函数的栈帧，func1 stack frame为当前函数(被调用者)的栈帧，栈底在高地址，栈向下增长。此图是网上的图，理论上应该是上图的格式，fp、sp、lr和pc这四个寄存器是非常特殊的寄存器，它们记录了当前正在运行的函数一些重要信息，在刚进入一个新的函数开始执行的时候，它们保存的是上个函数的信息，需要将它们入栈保存起来，这很重要！这些并没有定义在ATPCS（ARM System Developer's Guide中定义的函数调用的规则）中，ATPCS规定的是函数调用的时候参数如何传递，以及函数返回值的保存等。上面的这些个人觉得是一种默契，定义函数现场的保存及恢复，这些默契包括ATPCS都是人为的一种约束，目的是为了保证程序运行中不会出错，具体怎样实现应该是不同的编译器不尽相同。

例子：

#include <stdio.h>

void a(int i)

{

if(i > 0)

{

a(--i);

}

else

{

printf("i has reached zero!\n");

}

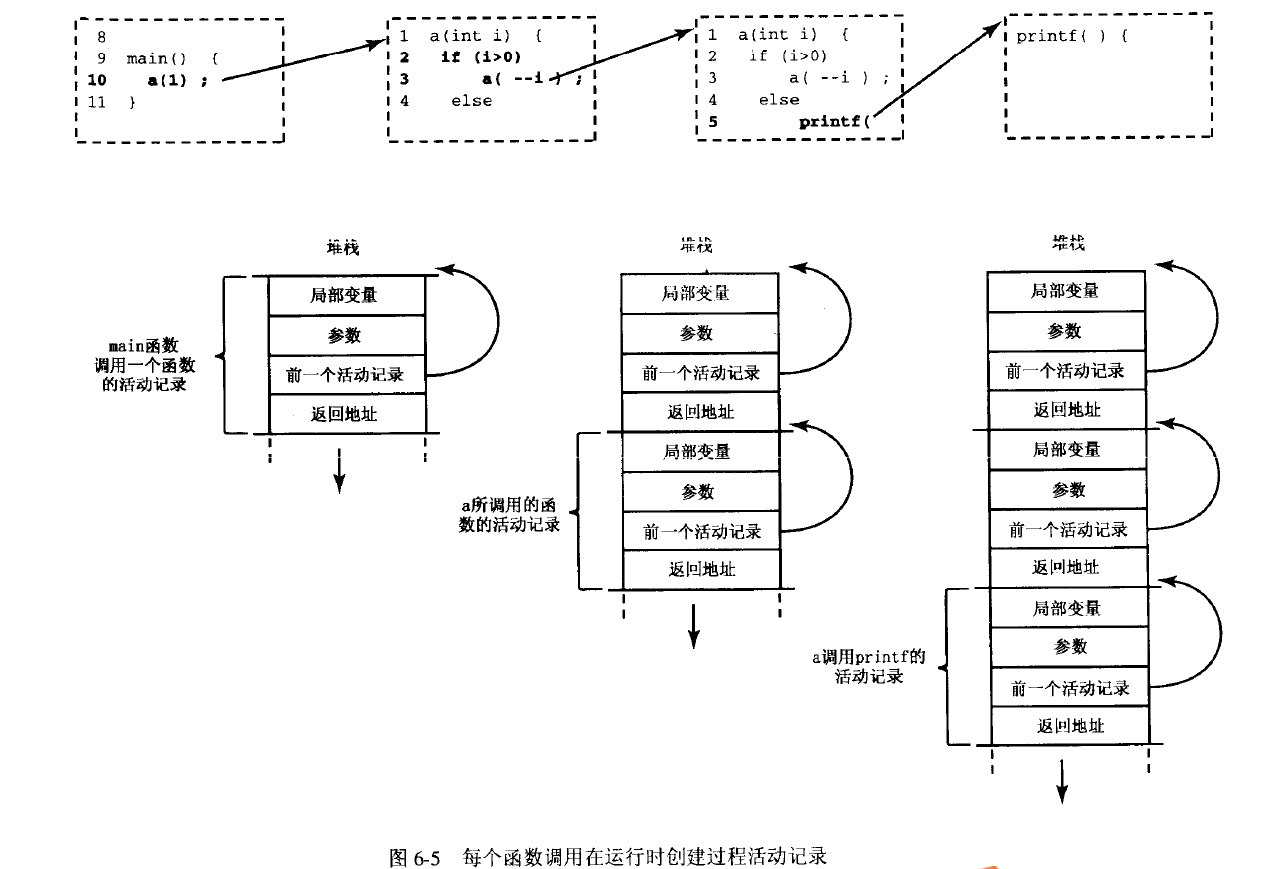
}

void main()

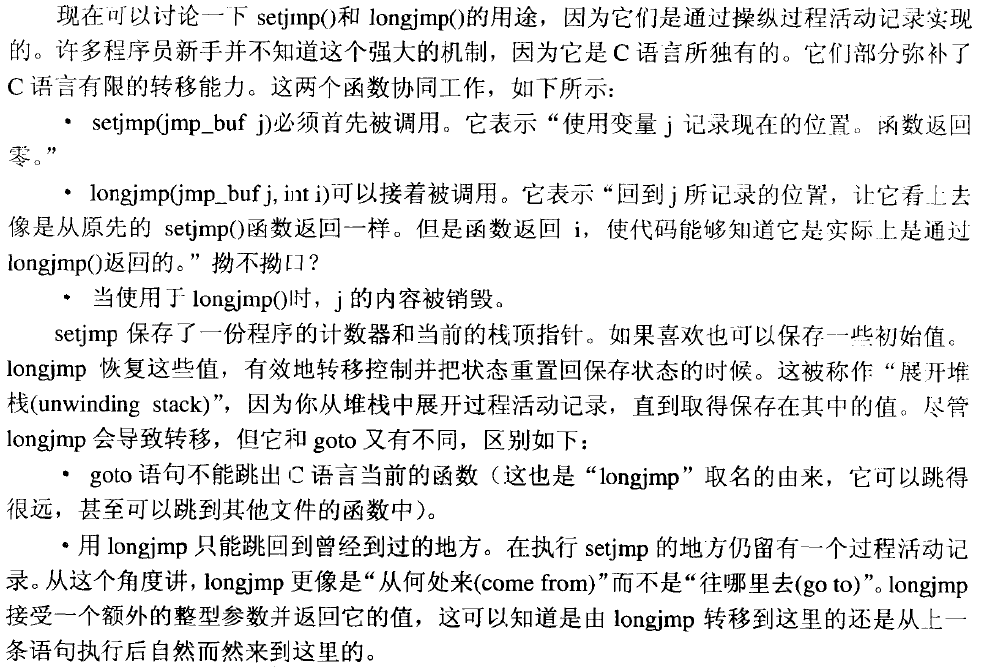
{

a(1);

}



### Setjmp和longjmp



Setjmp和longjmp最大的作用是用来恢复错误，只要还没有从函数中返回，一旦发现一个不可恢复的错误，可以把控制转移到主输入循环，并从那里重新开始。在C++中是catch和throw。