# 如何更好地使用C

为什么可以写出规范的C代码?那是建立在你趟过很多浑水、掉过很多坑的经验积累，能按照规范来写C程序意味着你对操作系统和体系结构对C的底层实现有了深刻了解，对于任何奇怪的Bug你都可以给出合理的解释并修复它。

## 控制IO格式

### 制表符

指标符’\t’，顾名思义将内容以表格的形式排布。其宽度不确定，2~8都有可能，依据内容而定。

示例1：

|  |
| --- |
| printf("A\tB\tC\n");  printf("1\t2\t3\n");  输出： |

示例2：

|  |
| --- |
| printf("Hello\tWolrd\t123\n");  printf("HELLO\tWORLD\tC\n");  输出： |

### 退格符

退格符’\b’主要用于修改流，退格符会将流指针往前回退一个字符。每有一个字符加入流，该字符会被填充到当前流指针的位置并将流指针向后移动一个字符。

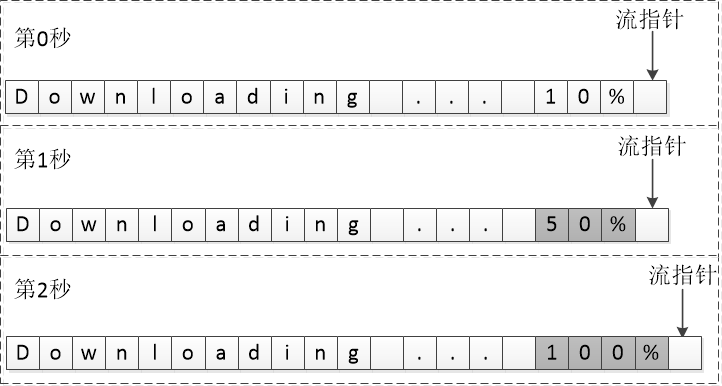
示例1：

|  |
| --- |
| printf("Downloading ... 10%%");  fflush(stdout);  sleep(1);  printf("\b\b\b50%%");  fflush(stdout);  sleep(1);  printf("\b\b\b100%%\n");  输出：  （第0秒）  （第1秒）  （第2秒，结束） |

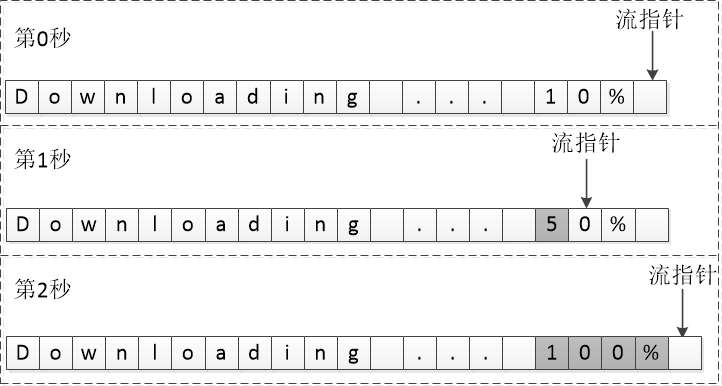
示例2：

|  |
| --- |
| printf("Downloading ... 10%%");  fflush(stdout);  sleep(1);  printf("\b\b\b5");  fflush(stdout);  sleep(1);  printf("\b100%%\n");  输出：  （第0秒）  （第1秒）  （第2秒，结束） |

两个示例的执行效果是一样的，但过程不同，示例也说明两点：1) 刷新缓冲区不影响退格符移动流指针；2) 退格符仅对当前行奏效，无法修改上一行的内容。示例1的执行过程如下图所示：



示例2的执行过程如下图所示：



### 格式化IO参数

C使用’%’作为格式化IO变量的参数，所有基本类型的格式化表示方法见下表：

|  |  |  |
| --- | --- | --- |
| 类型 | 格式化表示 | 常量示例 |
| char | %c | ‘a’ |
| int | %d(十进制), %o(八进制),%x(十六进制) | 15 |
| float | %f | 15.0f,15.0F |
| double | %lf,%f | 15.0 |

组合类型表示方法见下表：

|  |  |  |
| --- | --- | --- |
| 类型 | 格式化表示 | 常量示例 |
| long int/long | %ld(十进制),%lo(八进制),%lx(十六进制) | 15l, 15L |
| long long int | %Ld(十进制),%Lo(八进制),%Lx(十六进制) | 15L |
| short int/short | %hd(十进制),%ho(八进制),%hx(十六进制) | 15 |
| unsigned short | %hu,%u | 15u |
| unsigned int | %u | 15u |
| unsigned long | %lu | 15ul |
| unsigned long long | %Lu | 15uL |
| unsigned char | %c | ‘c’ |
| long double | %Lf | 15.0L |

注意格式化表示必须和类型匹配，因为printf/scanf/fprintf/fscanf等函数可传递可变长度参数，每个参数的地址偏移和其类型相关，而类型长度大小是根据格式化表示’%’判断的，具体见可变参数[printf的实现](#_可变参数)。

示例1：

|  |
| --- |
| printf("%3d\t%5.2f\t%4lu\n", 12, 1.5, 1234ul);  输出： |

示例2：

|  |
| --- |
| printf("%-3d\t%5.2f\t%4lu\n", 12, 1.5, 1234ul);  输出： |

示例3：

|  |
| --- |
| printf("%3x\t%.2Lf\t%4lu\n", 12, 1.5L, 1234ul);  输出： |

示例4：

|  |
| --- |
| printf("%\*.\*s\t%.4s\n", 10,3,"hello","hello");  输出： |

三个示例说明可以格式化数据显示宽度，%nld,%nf,%nlf,%nlu中的n表示数据宽度，这里加上n之后数据会右对齐，如果n是负数，则会左对齐。还可以使用%n.mf,%n.mLF表示，n仍旧是数据宽度，m表示数据精度保留m位。

### 其他转义字符

|  |  |
| --- | --- |
| 转义字符 | 含义 |
| \a | 响铃 |
| \f | 换页 |
| \\ | 反斜杠 |
| \? | 问号 |
| \’ | 单引号 |
| \” | 双引号 |
| \ooo | 八进制，例如’\142’是字符’b’ |
| \xhh | 十六进制，例如’\x62’是字符’b’ |
| \r | 回车符，移动流指针到行开始处 |
| \v | 纵向制表符 |

使用’\’ 延续字符串：

|  |
| --- |
| printf("123\  -%\  d\n", 456);  输出：    printf("123\  -%\  d\n");  输出： |

字符串延续的另一种方式：

|  |
| --- |
| char \*s = "123" "-";  printf("%s""456\n", s);  输出： |

使用’\’延续代码：

|  |
| --- |
| int a, b\  =5; // 等价于int a, b=5; |

|  |
| --- |
| #define PRINT(a, n) \  int i; \  for (i=0;i<n;++i){ \  printf("%d\n", a[i]); \  }  int a[] = {1,2,3};  PRINT(a, sizeof(a)/sizeof(a[0]))  输出： |

### 常用的IO函数

|  |  |  |
| --- | --- | --- |
| 函数原型 | 含义 | 返回 |
| int printf(const char \*format, ...) | 将参数按format格式化为字符串，打印到stdout | 成功：打印到stdout的字符个数（不含’\0’）  失败：负数 |
| int scanf(const char \*format, ...) | 从stdin读取字符串，按format中的格式对剩余参数赋值 | 成功：成功匹配并赋值的输入项个数  文件末尾：EOF |
| int fprintf(FILE \*stream, const char \*format, ...) | 和printf相同，但将输出定向到stream指向的文件 | 和printf相同 |
| int fscanf(FILE \*stream, const char \*format, ...) | 和scanf相同，但将输入定向为stream指向的文件 | 和scanf相同 |
| int sprintf(char \*str, const char \*format, ...) | 和printf相同，但将输出定向到str指向的字符串 | 和printf相同 |
| int sscanf(const char \*str, const char \*format, ...) | 和scanf相同，但将输入定向为str指向的字符串 | 和scanf相同 |
| int getchar(void) | 从stdin读取一个字符 | 成功：从stdin读取的字符对应的ASCLL码  文件末尾：EOF |
| int putchar(int c) | 写一个字符c到stdout | 成功：写入stdout的字符对应的ASCLL码  失败：负数 |
| int getc(FILE \*stream) | 从stream指向的文件读取一个字符 | 成功：读取的字符对应的ASCLL码  文件末尾：EOF |
| int putc(int c, FILE \*stream) | 向stream指向的文件写入一个字符c | 成功：写入stream指向文件的字符对应的ASCLL码  失败：负数 |
| char \*fgets(char \*line, int size, FILE \*stream) | 从stream指向的文件读取一行（包括换行符）到line，最多可读取size-1个字符，以’\0’结尾 | 成功：返回line  文件末尾：NULL |
| int fputs(const char \*s, FILE \*stream) | 将s写入stream指向的文件 | 成功：非负整数  失败：负数 |
| char \*gets(char\* line) | 从stdin读取一行到line，line会自动过滤’\n’ | 成功：返回line  文件末尾：NULL |
| int puts(const char\* line) | 写入line到stdout，自动添加’\n’ | 成功：非负整数  失败：负数 |

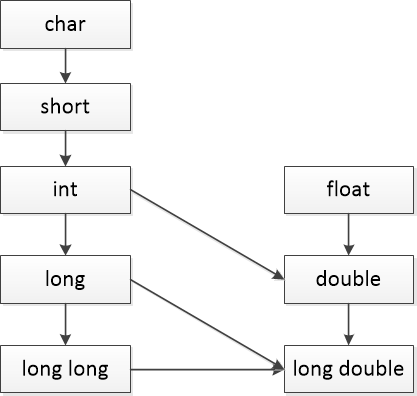
## 赋值与计算

### 类型转换

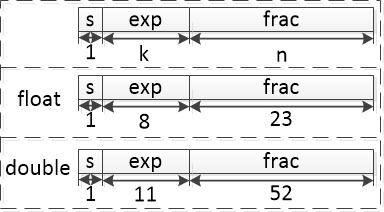
C语言允许从窄字节的操作数自动转换成款字节操作数精度无损转换，这里给出x86\_64（Ubuntu\_x64）机器上各数据类型的宽度：

|  |  |
| --- | --- |
| 类型 | 宽度（字节） |
| char | 1 |
| short | 2 |
| int | 4 |
| long | 8 |
| long long | 8 |
| float | 4 |
| double | 8 |
| long double | 16 |

下图所示的隐式类型转换都不会产生精度损失，可以直接进行隐式的自动类型转换。同时跨类型的无损转换也是允许的，例如从char直接到float/double/int/long也是允许的。这里需要注意，从int到float、从long/long long到float/double的转换编译器都不会给出警告，但实际上这是精度有损的。为了解决从整形到浮点型的无损转换，上表给出了比long long更长字节的long double，而从int到double、从long/long long到long double就是精度无损的。



下面谈谈为什么int不能无损转换成float。浮点数在计算机中的表示方式如下图所示，浮点数的计算方法如式1所示。



式1

基于归一化表示方式，分别计算符号位s、指数位exp和分数位frac的值。

1. 符号位s仅有一位，代表浮点数是正数还是负数。1代表负，0代表正。
2. 指数位exp用于计算式1中的E，E的计算方法见式2。要求exp≠00…0并且exp≠11…1

式2

其中，，

float类型：，，

double类型：，，

1. 分数位frac用于计算式1中的M，，，而就是n位的分数部分，计算机默认在分数部分前加附加，M最小可取00…0(1.0)，最大可取11…1(2.0-)。

举例，对于浮点数，，所以，则。，所以最终的二进制表示方式如下图所示：

C:\Users\DELL\Desktop\绘图2.png

从上面例子我们看到，数值的整形部分会映射到float的分数部分，而float的分数部分只有23位，明显会出现精度损失，例如示例1。但double的分数部分有53位，完全可以满足int的无损转换，例如示例2。

示例1：

|  |
| --- |
| int i = 123456789;  float a = i;  printf("%f\n", a);  输出： |

示例2：

|  |
| --- |
| int i = 123456789;  double a = i;  printf("%f\n", a);  输出： |

这里给出各浮点类型的取值范围和精度：

|  |  |  |
| --- | --- | --- |
| 浮点类型 | 有效小数位数 | 范围 |
| float | 6~7 | [-3.4\*10^38,-1.0\*10^-38], [1.0\*10^-38,3.4\*10^38] |
| double | 15~16 | [-1.7\*10^308,-1.0\*10^-308], [1.0\*10^-308,1.7\*10^308] |
| long double | 18~19 | [-1.2\*10^4932,-1.0\*10^-4932], [1.0\*10^-4932,1.2\*10^4932] |

关于隐式类型转换，ANSI C规定表达式中的char都会被转换为int，float转换为double，short、枚举和位都会被转换成int。例如示例3。函数参数也是表达式，所以printf中%d适用于char、short和int，%f适用于float和double；在参数传递时也会存在隐式类型转换，char、short、int会被当做int字节大小压入栈，所以va\_arg(va, int)可用来取一个char、short或int类型参数，va\_arg(va, double)可用来取一个float或double类型参数，使用固定的几种类型会大大简化参数传递过程。

示例3：

|  |
| --- |
| char c1,c2;  c1 = c1 + c2; // c1和c2被提升为int进行相加，然后将结果转换为char |

### 运算符优先级

下表列举了C运算符结合性，从上到下优先级从高变低。

|  |  |
| --- | --- |
| 运算符 | 结合性 |
| () [] -> . | 左结合 |
| ! ~ ++ -- + - \* (type) sizeof & | 右结合 |
| \* / % | 左结合 |
| + - | 左结合 |
| << >> | 左结合 |
| < <= > >= | 左结合 |
| == != | 左结合 |
| & | 左结合 |
| ^ | 左结合 |
| | | 左结合 |
| && | 左结合 |
| || | 左结合 |
| ?: | 左结合 |
| = += -= \*= /= %= &= ^= |= <<= >>= | 右结合 |
| , | 右结合 |

注意：C标准对二元运算符+、-、\*、/、|、&的左右表达式运算次序没有做规定，例如res = exprA op exprB，对exprA和exprB的运算次序和具体编译器实现有关，如果两个表达式运算次序不同则res可能也不一样，所以最好以中间变量过渡，不要让代码执行结果和求值顺序相关。

### 指针与数组的区别

1. 指针是左值，数组名是右值不能使用操作符和被赋值，例如示例1。其他情况下数组和指针可以相互转换，元素访问方法也完全相同。

示例1：

|  |
| --- |
| int a[10];  int \*ap = a;  a++; // 非法  a = ap; // 非法  ap++; // 合法  ap = &a[1]; // 合法 |

1. 大小不同，指针不管指向什么sizeof的结果永远是内存地址编码长度，数组则是数组长度\*元素大小，例如示例2

示例2：

|  |
| --- |
| int\* a = (int\*)malloc(sizeof(int)\*3);  int b[3] = {1,2};  printf("%ld %ld\n", sizeof(a), sizeof(b));  输出： |

1. 数组是定长，指针可以变长。见示例3

示例3：

|  |
| --- |
| int a[3][3], \*ap[3], i;  for(i=0;i<3;++i){  ap[i] = (int\*)malloc(sizeof(int)\*(i+1));  }  /\*  \* a是二维数组，大小为3\*3\*4  \* ap是包含3个int指针的数组，数组0~2号元素的指针分别指向1、2、3个int元素  \* a占用空间多，元素地址连续，ap是变长二维数组，每个数组指针元素指向的int数目可以不同，如图：  a的地址空间：  C:\Users\DELL\Desktop\绘图1.png  ap的地址空间：  C:\Users\DELL\Desktop\绘图1.png  \*/ |

1. 实现方式不同，指针和数组访问数据的具体区别见《C专家编程》4.2、4.3节，这也是指针、数组声明和定义必须保持一致的原因。

指针和数组在以下两种情况下可互换：

1. 函数参数。例如void f(int \*a) ⬄ void f(int a[])，两种情况编译器都是把参数当做指针（数组开始地址）来处理的，调用f时可以传入数组或指针。函数数组参数被编译器当做指针来处理的原因是传递整个数组的效率很低，如果只传递数组的开始地址也可以达到相同的使用效果。下表展示了多维数组作为函数参数被编译器处理后的样子。

|  |  |
| --- | --- |
| 函数参数（形参或实际传递） | 编译器处理后的形式 |
| char s[8][10] | char (\*s)[10] |
| char \*s[10] | char \*\*s |
| char (\*s)[10] | char (\*s)[10] |
| char \*\*s | char \*\*s |

示例4展示了所有合法的三维数组传参方法。

示例4：

|  |
| --- |
| void fun\_a(int v[2][3][5]){}  void fun\_b(int v[][3][5]){}  void fun\_c(int (\*v)[3][5]){}  int main(){  int v[2][3][5];  fun\_a(v);  fun\_b(v);  fun\_c(v);  int (\*p)[3][5] = v;  fun\_a(p);  fun\_b(p);  fun\_c(p);  int (\*q)[2][3][5] = &v;  fun\_a(\*q);  fun\_b(\*q);  fun\_c(\*q);  return 0;  } |

1. 元素访问。例如a[i] ⬄ \*(a+i)，不论a原来是数组还是指针，都支持两种访问方式。

指针的有效**算数运算**包括：

1. 同类型指针之间的赋值运算
2. 指针同整数之间的加减法运算
3. 指向相同数组中元素的两个指针之间的减法和比较运算
4. 指针赋值为0或与0之间比较

函数可以返回指针，但指针绝不能指向函数的局部变量。见示例5。

示例5：

|  |
| --- |
| int\* f(int\* a)  {  int i = 1;  return &i; // 非法，局部变量i离开f后会自动回收，地址不再有效  static int is;  return &is; // 合法，可以返回指向static变量的指针  return a; // 合法，f参数传入的指针指向的内容并没有被回收  } |

### 结构体

赋值方法，见示例1。

示例1：

|  |
| --- |
| typedef struct {  const char\* s;  int count;  } Word;  Word word = {"00", 0};  Word content[] = {{"12", 2}, {"34", 1}};  Word content2[] = {"12", 2, "34", 1}; // 与content等价 |

大小计算方法：

1. 按照struct内部成员变量（函数不算）声明次序开始，第一个变量的地址从0开始。
2. 每个元素E的地址偏移量为Addr\_E=N\*sizeof(min(E, pack))，N是整数要求Addr\_E大于等于上一个元素的地址偏移量加上上一个元素大小，即Addr\_Last+sizeof(Last)。E是当前元素类型的大小，pack是系统默认的内存对齐字节数可以通过#pragma pack(n)设置。
3. 最终的struct大小S为S=max(Addr\_End, N\*min(pack, MAX\_ELE\_TYPE))，其中Addr\_End是最后一个元素的结束地址，N是整数要求计算出的S大于等于Addr\_End，pack是系统默认的内存对齐字节数，MAX\_ELE\_TYPE是结构体中当前类型最大的元素类型的大小。
4. 如果存在嵌套结构体，把嵌套的结构体整体看成一个元素，该元素的偏移量为结构体min(MAX\_INNER\_ELE\_TYPE, pack)，其中MAX\_INNER\_ELE\_TYPE为嵌套结构体中类型最大的元素类型大小，pack是系统默认的内存对齐字节数，该元素的就是嵌套结构体的大小。

示例2：

|  |
| --- |
| struct S{  int i;  char c;  double b;  char v[12];  };  输出： |

示例3：

|  |
| --- |
| #pragma pack(4)  struct S{  int i;  double b;  };  输出： |

示例4：

|  |
| --- |
| #pragma pack(2)  struct S{  int i;  char c;  double b;  char v[12];  };  输出： |

示例5：

|  |
| --- |
| struct S{  int i;  struct In{  int i;  double d;  };  double d;  };  输出： |

示例6：

|  |
| --- |
| struct S{  int i;  struct In{  int i;  double d;  } InnerEle;  double d;  };  输出： |

示例7：

|  |
| --- |
| #pragma pack(2)  struct S{  int i;  struct In{  int i;  double d;  } InnerEle;  double d;  };  输出： |

示例8：

|  |
| --- |
| struct S{  unsigned int bit1 :1; // unsigned int类型的1个bit位  unsigned int bit2 :2;  };  输出： |

### 联合体

大小计算方法：

1. union大小S为S=N\*min(pack, MAX\_ELE\_TYPE), S≥MAX\_ELE，其中pack是系统默认的内存对齐字节数，MAX\_ELE\_TYPE是联合体中当前类型最大的元素类型的大小，MAX\_ELE是联合体中当前最大的元素大小（数组也是元素），N是整数要求计算出的S大于等于MAX\_ELE。

判定小端和大端：

|  |
| --- |
| union {  char c;  int i;  } u;  u.i = 1;  printf(u.c == 1 ? "Little Endian\n":"Big Endian\n");  输出： |

示例1：

|  |
| --- |
| union S{  char v[10];  double d;  };  输出： |

示例2：

|  |
| --- |
| #pragma pack(4)  union S{  char v[10];  double d;  };  输出： |

## 函数

### 可变参数

printf的实现：

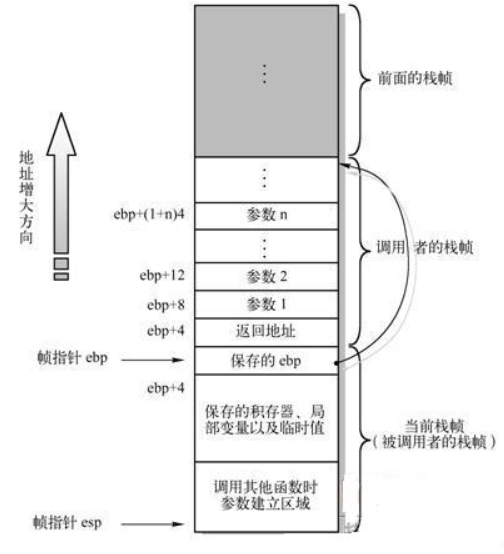
|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <stdarg.h>  void myPrintf(const char\* num, ...){  va\_list va; // 参数列表，用于存放接收的可变参数  va\_start(va, num); // 将第一个可变参数的首地址赋给va  while(\*num!='\0'){  if(\*num=='%'){  ++num;  if(\*num=='c'){  char c = va\_arg(va, int); // 获取一个可变参数的内容，并设置其类型为int，并将可变参数列表的地址偏移指向下一个可变参数  printf("%c", c);  ++num;  }else if(\*num=='s'){  char\* s = va\_arg(va, char\*);  printf("%s", s);  ++num;  }else if(\*num=='d'){  int i = va\_arg(va, int);  printf("%d", i);  ++num;  }else{  break;  }  }else{  printf("%c", \*num);  ++num;  }  }  va\_end(va); // 清空可变参数列表，释放空间  }  int main(){  myPrintf("I receive %s,%d,%c,123\n", "string", 1, 'q');  return 0;  }  输出： |

### 函数栈帧

机器用栈传递参数、存储返回信息。为函数调用分配的栈叫栈帧，由两个寄存器esp（栈顶）和ebp（栈底）圈定。

函数调用发生：

|  |
| --- |
| 当前栈帧  Push args // 参数压入数据栈  Push PC // 压入返回地址（返回后继续执行PC指向的instruction）  Push ebp // 保存原栈底  Mov ebp esp // 令当前栈底为esp，开始进入新的栈帧  … // 新的栈帧  Pop ebp // 修改ebp为原栈底  Pop PC // 设置PC为跳转前的PC位置  Pop args // 参数出栈 |



参数从右到左入栈，所以printf(“%d %d”,\*p,\*(++p));打印的两个变量一样，注意返回地址也要入栈。ebp下面将调用的函数的局部变量压栈。

示例1：

|  |
| --- |
| void pass(void){  int v;  int \*temp = &v;  while(\*temp != 123){ // 从此栈帧向高地址寻找值为123的地址，由于内存对齐所以每次移动(int)4字节是合理的，从可变参数中char被当做int的处理就可以看出内存每次至少以4字节偏移。  ++temp;  }  \*temp = 456;  }  int main(){  int a = 123;  pass();  printf("%d\n", a);  return 0;  }  输出： |

### 函数指针

使用示例：

|  |
| --- |
| **函数定义：**  void f(int){  …  }  **声明：**  typedef void (\*F)(int); // F是一个类型，代表参数为int返回值为void的函数指针类型  F fp; // fp是一个变量，代表一个参数为int返回值为void的函数指针  void (\*ffp)(int); // ffp是一个变量，代表一个参数为int返回值为void的函数指针  **赋值：**  fp = f;  F tmpfp = f;  ffp = f;  void (\*tmpffp)(int) = f;  **调用：**  fp(1);  (\*fp)(1);  ffp(2);  (\*ffp)(2);  tmpfp(3);  (\*tmpfp)(3);  tmpffp(4);  (\*tmpffp)(4); |

## 工程环境

### 生成库

库的本质是.c源文件经过编译后生成的.o文件集合，并将.o文件压缩打包后就是库，所以库是和平台相关的。

|  |  |  |
| --- | --- | --- |
|  | 生成指令 | 特点 |
| 静态库 | ar -cr libA.a a.c | 最终可执行文件体积大，执行效率高，执行不依赖于库。如果多个程序含有同一个静态库，内存会将静态库加载多次，造成体积膨胀 |
| 动态库 | gcc -shared -fPIC -o liba.so a.c | 最终可执行文件体积小，动态链接（边执行边引用），执行效率低。执行需要依赖动态库，可以在程序运行时在线更新动态库文件。多个程序链接同一个动态库，动态库在内存仅被加载一次 |

### 可执行文件生成过程

**预处理：**代码文本替换

1. 删除#define，展开宏定义，包括宏定义的全局变量、代码（常见于头文件防止被多次包含的#ifndef A\_H\_ #define A\_H\_）、函数。
2. 处理条件编译指令#if，#ifdef，#elif，#else，#endif
3. 处理#include，递归包含头文件
4. 删除注释//，/\*\*/，将注释替换成空格

示例：

|  |
| --- |
| int a/\*\*/b = 1;  宏替换后：  int a b = 1; |

1. 添加行号和文件名标识，定义\_\_LINE\_\_和\_\_FILE\_\_
2. 保留#pragma

**编译：**从高级语言到汇编代码（目标计算机体系结构相关），gcc -S a.c -o a.s

1. 词法分析

编译器对符号的解析策略是贪心的（大嘴法），如果编译器的输入流截止至某个字符之前都已经被分解为一个个符号，那么下一个符号将包括从该字符之后可能组成一个符号的最长字符串。

示例：

|  |
| --- |
| a---b等价于 a-- - b等价于(a--) - b  a - --b等价于a - (--b) |

1. 语法分析

多余的分号会被编译器翻译为空语句，或者编译优化直接去掉。

函数调用不加括号直接调用合法，意味着获取函数地址，例如：

|  |
| --- |
| void f(){}  f; // 合法，计算函数f的地址 |

1. 语义分析
2. 优化

**汇编：**从汇编语言到目标机器指令（二进制，目标CPU相关），as a.s -o a.o

**链接：**目标文件+库 => 可执行文件(ld \*.o -L… -o main)

链接器的输入是一组目标模块和库文件，输出是一个载入模块。对每个目标模块中的每个外部对象，链接器检查载入模块看是否已经有同名的外部对象，如果没有，链接器将外部对象添加到载入模块；如果有，链接器开始处理命名冲突。例如，一个包含printf函数的目标模块，有一个对printf函数的引用，可以推测出该引用指向位于某个库文件中的外部对象，链接器生成载入模块时会记录这个外部对象的引用。当链接器读入一个目标模块时，它必须解析这个目标模块定义的所有外部对象的引用，并标记这些外部对象不再是未定义的。

链接过程会将实现和调用对应起来，包括全局变量和函数。所以变量和函数不可重复定义，不然链接的时候会出现歧义（到底是链接哪个实现）。函数不一定必须在.h文件声明，也可以在局部作用域声明，编译器会自动链接函数到对应的实现（编译器会为每个变量和函数维护符号表），例如：

|  |
| --- |
| main.c:  int main(){  double my\_atoi(char\* s);  }  realize.c:  double my\_atoi(char\*){  //…  }  // 虽然main.c没有包含声明头文件，但编译器仍可以找到my\_atoi的实现 |

### 预编译

**NDEBUG：**开关debug模式

示例1：

|  |
| --- |
| #ifndef NDEBUG  printf("Debug mode\n”);  #endif  gcc -DNDEBUG –o aa a.c // 关闭debug模式，assert语句不会执行  gcc -o aa a.c // 开启debug模式，assert语句都会执行 |

**pragma：**要求编译器执行某些特殊操作

|  |
| --- |
| #pragma pack(4) // 设置内存对齐字节数为4 |

**error：**编译器终止编译，打印出错消息

|  |
| --- |
| #error I am wrong  输出： |

**include：**#include "XXX"，会优先从当前项目中用户自定义的文件查找XXX，如果没有再查找环境或库中的XXX；#include <XXX>，会优先从环境或库里查找XXX,如果没有再从用户自定义的文件查找XXX。

**define：**宏定义函数举例：

|  |
| --- |
| #define dprint(expr) printf(#expr " = %g\n", expr)  dprint(x/y)会被替换为：  printf("x/y" " = %g\n", x/y) // 参数名以#作为前缀则结果被扩展为由实际参数替换的带引号的字符串  #define paste(X, Y) X ## Y  int paste(a,b) = 1;会被替换为：  int ab = 1; // ##用于连接变量名称 |

**undef：**取消名字的宏定义

**defined：**判断名字有没有被定义过，示例1、2等价

示例1：

|  |
| --- |
| #ifndef A\_H  #define A\_H  …  #endif |

示例2：

|  |
| --- |
| #if !defined(A\_H) && defined(B\_H)  #define A\_H  …  #endif |

**预定义宏：**

|  |  |
| --- | --- |
| \_\_LINE\_\_ | 被编译文件的行号 |
| \_\_FILE\_\_ | 被编译文件的名字 |
| \_\_DATE\_\_ | 编译日期 |
| \_\_TIME\_\_ | 编译时间 |

### 编译选项和链接选项

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 编译选项 | 链接选项 | 说明 | 示例 |
| -L | √ | × | 指定库目录，里面存放程序需要链接的库，默认有/usr/lib和/lib | -L./lib |
| -I | √ | × | 指定头文件目录，里面存放需要包含的头文件，默认有/usr/include/ | -I./include |
| -l | √ | × | 指定程序要链接的库，这些库必须在-L指定的目录下面 | -la，libA.a |
| -Wl,-rpath | × | √ | -Wl表示后面的选项在链接过程生效。-rpath指定程序深层链接的库目录，特指-l连接的库还依赖于其他库，而且这些依赖库的目录还未指定。-rpath默认包括/usr/lib和/lib，使用ldd liba.so查看动态库liba.so依赖的库 | -Wl,-rpath=./lib，这句话解决了liba.so依赖于libb.so的深层库链接问题，即使-L=./lib，libb.so在./lib中 |
| -Wl,-rpath-link | × | √ | 和-rpath的区别在于，rpath不但指定链接时动态库的目录，还指定了运行时动态库的目录，rpath-link不指定运行时库的查找目录，所以使用-rpath-link需要将动态库拷贝至/usr/lib，使用-rpath需要将动态库拷贝至/usr/lib或-rpath指定的目录 | -Wl,-rpath-link=./lib |

示例1：

|  |
| --- |
| gcc -shared -fPIC -o liba.so a.c  gcc -o main main.c -L. -la -Wl,-rpath=. // 运行./main OK，此时运行时动态库要从/usr/lib、/lib和.目录下查找  gcc -o main main.c -L. -la -Wl,-rpath-link=. // 运行./main报错，此时运行时库目录仅从/usr/lib和/lib目录下查找，除非将liba.so放到/usr/lib下  // 为了运行时更新和减小可执行文件与内存体积，尽可能使用动态库，具体原因详见《C专家编程》5.2，库链接的方法见5.3 |

### 调试

**逻辑调试：**gdb-https://blog.csdn.net/tzshlyt/article/details/53668885

**编程规范：**splint-https://blog.csdn.net/xiaowang1379214245/article/details/82688802

**内存查漏：**valgrind-https://www.cnblogs.com/AndyStudy/p/6409287.html

## 作用域

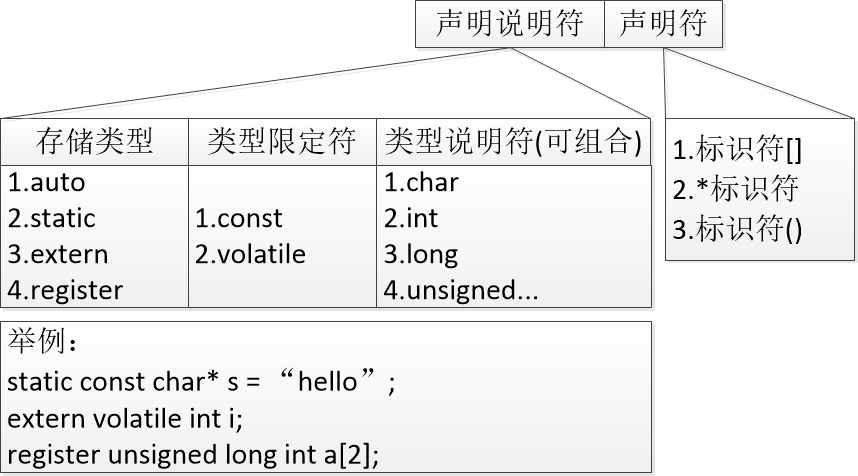
### 变量和函数

extern int a; // 整型变量a的声明，意味着在这个文件中可以使用a，这个变量可能在当前.c文件的其他地方定义或者在其他.c文件中定义。生存周期为定义开始到程序结束。其实int a表示定义，但如果前面加extern，编译器就不会分配空间给它

void f(); // 函数的声明可不加extern，C中的函数都是全局函数

注：声明的变量和函数标识符不要和系统关键字和保留字相同，C允许用户自定义和标准库同名的函数，这种行为叫interpositioning，但要警惕interpositioning，详见《C专家编程》5.4

### 声明结构



解析复杂声明的准则：

1. 从内向外读声明符。定位标识符，然后从此处的声明开始解释。
2. 根据运算符优先级逐步对标识符做解释。
3. 函数不能返回数组、函数，不能定义函数型的数组。

示例：

|  |
| --- |
| int \*(\*x[10]) (void); // x是一个数组，每个数组元素是一个函数指针，该函数传参void并返回int\*  void (\*(\*x)[])(int p); // x是一个指向数组的指针，每个数组元素是一个函数指针，该函数传参int并返回void  int (\*(\*x)(int \*p))[5]; // x是一个函数指针，该函数传参int\*并返回一个指针，该指针指向包含5个int元素的数组  int (\*(\*x)[5])(int \*p); // x是一个指针，指向一个数组，该数组包含5个函数指针，每个函数传参int\*返回int |

### 关键字

**static：**静态变量/函数，作用域为当前文件，生存周期为声明开始到程序退出，所以下面示例编译正确：

|  |
| --- |
| a.c:  static int i = 2;  int main(){  extern int i;  printf("%d\n", i);  return 0;  }  b.c:  int i = 1;  输出：    // a.c中的i就近原则找到static int i=2，如果改行被注释则会找到b.c中的int i=1,。编译器根据全局变量符号表检测变量和函数的重定义，static int i并不会出现在全局变量符号表中，所以编译正确。 |

static还可用于修饰函数内的局部变量，用于记忆变量。第一次调用时对变量赋值，后续每次函数调用该变量都使用上一次函数退出前的值。

**const：**常量，声明与定义相伴随，作用域为当前文件，除非声明和使用处都使用extern关键字修饰。

|  |  |
| --- | --- |
| 语句 | 含义 |
| const int\* a; | a是指向常量int的指针，其地址指向的内容不能变 |
| int\* const a; | a是指向int的常量指针，其地址不能变 |
| const int\* a[10]; | a是一个数组，数组a包含10个指向int的指针，并且每个指针指向的int都是常量 |
| int\* const a[10]; | a是一个数组，数组a包含10个指向int的指针，并且每个指针都是常量，它的地址不可修改 |
| const int(\*a) [10]; | a是一个指针，指针a指向一个数组，这个数组含有10个int，并且这个数组是常量不可修改 |
| int(\* const a)[10]; | a是一个指针，指针a指向一个数组，这个数组含有10个int，并且a是常量，地址不可修改 |
| typedef int\* Pint;  const Pint a = &i; | a是一个常量指针，不能再指向其他int变量，但可以修改其值 |

**register：**声明变量在程序中使用频繁，建议编译器将变量放到寄存器中。register只能修饰局部变量或函数形参，例如：

|  |
| --- |
| register int x;  void f(register int p); |

**volatile：**取消编译器对变量的访问优化，必须从内存访问变量（而不是寄存器或或高速缓存）。volatile解决的是多线程间共享变量的**可见性**问题,并不能保证变量非原子性操作带来的安全问题。例如多线程的i++,++i（这两个都是非原子操作）,依然还是会存在多线程带来的变量结果不确定问题。

## 内存布局

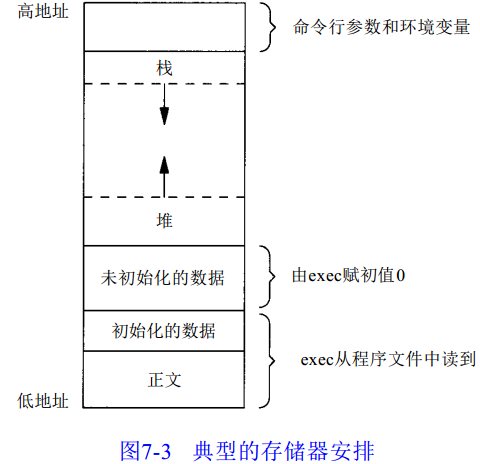
### 程序内存布局

**逻辑上：**指的是程序逻辑上的内存数据区域划分，对于任何程序[Java/C++/C]而言都一样

1. 栈区：系统自动动态分配，例如局部变量（局部const变量也包括在内）和函数调用
2. 堆区：程序员手动动态分配，例如malloc
3. 全局静态区：包括全局变量（初始化和未初始化）、static数据
4. 常量区：宏定义常量、字符串字面值
5. 代码区：程序执行代码

**物理上：**指的是使用C编译器完成后用size myProgram查看可执行程序的实际物理空间由哪几部分构成，每个部分分别对应到逻辑上的哪些数据部分

|  |  |  |
| --- | --- | --- |
| 内存类型 | 物理内存区域 | 对应逻辑分区 |
| 动态区域（运行时动态分配的空数据间） | 栈(系统自动分配释放) | 栈区 |
| 堆(程序员手动分配释放) | 堆区 |
| 静态区域（编译时分配的数据空间） | data段 | 全局静态区的初始化变量+已初始化的static数据(全局和局部都算。而且初始化只包括初始化数值不为0的) |
| bss段 | 全局静态区的未初始化变量+未初始化的static数据(全局和局部都算。而且未初始化还包括初始化数值为0的) |
| text段 | 代码区+常量区 |



### 申请和释放堆区内存

**(void\*) malloc(size\_t size)：**申请size个字节的内存并返回指向该内存的指针，如果申请失败返回NULL；如果size为0，也可以正确返回指针，并且后续可以被free释放。

**(void\*) calloc(size\_t nmemb, size\_t size)：**申请nmemb个大小为size字节的元素对应的内存，并将这块内存初始化为0。如果申请失败返回NULL；如果nmemb \*size为0，也可以正确返回指针，并且后续可以被free释放。

**(void\*) realloc(void \*ptr, size\_t size)：**改变ptr指向的内存大小为size个字节，并返回指向新内存区域的指针，失败返回NULL。如果size比ptr原来指向的内存大，则ptr原来指向的内存内容不变，新增的内存区域不被初始化；如果size比ptr原来指向的内存小，在size范围内的内存内容不会改变，size超出来的内存会被释放；如果ptr是NULL，调用相当于malloc；如果size是0并且ptr不是NULL，调用相当于free。

**void free(void \*ptr)：**释放ptr指向的内存。

## 安全

### 指针操作

**悬垂指针**：指针曾经指向存放对象的一块内存，对象销毁后指针未置NULL，导致指针悬垂，再次使用这个指针会导致（恶意代码，程序奔溃）等UAP（Use after free）错误。

**野指针**：指针申明后未初始化，指针值不确定；指针指向的对象销毁后指针未置NULL；指针指向的地址超过变量范围（数组越界）。

**引用未初始化的内存**：新申请内存数据可能并不是预期的结果。而且可能访问到旧的内存信息，导致隐私泄露。

**重复释放**：导致内存管理器出问题。

### 可重入函数

可重入函数就是可以被中断的函数，可以在函数执行的任何时间点中断它，当调度回函数时可以继续执行。可重入函数必须保证每次输入相同的数据得到的输出一定是相同的。一般，包含以下三种情况的都不是可重入函数：

1. 使用全局变量，例如

示例1：

|  |
| --- |
| int a[10];  void f(){  int temp = 0;  for(int i=0;i<10;i++){  temp += a;  }  }// 使用全局变量a，可能在循环过程中由于调度，其他线程改变了a |

示例2：

|  |
| --- |
| void f(){  static int a = 1;  a++;  int temp = 2;  temp += a;  }// static变量是全局变量，虽然函数作用域外访问不到，但假如f由两个线程同时执行，线程1对a++后将要赋给temp，但赋值之前被线程2又a++一了次，这时是对同一个a的操作。 |

1. 函数内对堆区操作（malloc,free）

|  |
| --- |
| void f(){  int\* a = (int\*)malloc(sizeof(int)\*10);  }// 堆空间属于线程全局数据，因此可能被其他线程修改堆区数据，例如通过内存memset操作 |

1. 函数内调用了IO操作

|  |
| --- |
| void f(){  printf(“123”);  }  void g(){  printf(“456”);  }// 其他这里f和g分别被两个线程同时执行，可能最终的stdout输出124536 |

总结上述不可重入函数，都是由于多线程环境下对公共数据的操作引发的问题，所以可重入函数又叫做线程安全函数。编写线程安全函数应避免公共数据操作，例如栈是每个线程独有的，尽量使用栈数据。如果无法避免使用公共数据，请对临界区（对公共数据操作的代码）加锁。

### 缓冲区溢出攻击

拷贝内容大于目标缓冲区导致数据覆盖未知区域，可能会造成绕过验证或修改返回地址去执行恶意代码。

示例1：

|  |
| --- |
| char s[5];  char pass[15] = "1234";  scanf("%s", s);  if(strncmp(pass, s, 4) == 0){  printf("Success\n");  }else{  printf("Fail\n");  } // 栈向下增长，先声明的s输入覆盖后面变量pass，导致pass内容被改写  输出： |

推荐使用strncpy代替strcpy，最后在字符结尾处改0，例如示例2。

示例2：

|  |
| --- |
| // 安全的示例  char s[4] = "123", d[2];  strncpy(d, s, sizeof(d));  d[sizeof(d)-1] = '\0';  printf("%s\n", d);  输出：    //危险的示例  char s[4] = "123", d[2];  strncpy(d, s, sizeof(d));  //d[sizeof(d)-1] = '\0';  printf("%s\n", d);  输出： |

### 利用格式化字符串函数访问非法内存，导致隐私泄露、程序奔溃或执行恶意代码

格式化字符串函数参数数量和类型一定要匹配，大多数非法输入带有”%s”、”%p”之类的格式化字符串直接传入格式化字符串函数可能会导致意想不到的结果，示例1通过格式化字符串访问非法内存，导致密码泄露。

示例1：

|  |
| --- |
| int main(int argc, char\* argv[]){  if (argc < 2){  return 0;  }  char tmp[] = "password";  char\* secret = (char\*)malloc(sizeof(char)\*10);  strcpy(secret, tmp);  printf(argv[1]);  printf("\n");  free(secret);  return 0;  }  输出： |

### 整数溢出

整数操作在以下情况下一定要检查是否溢出

1. 数组索引
2. 指针运算
3. 对象长度或大小
4. 数组边界
5. 内存分配/内存拷贝长度
6. 循环终止条件

示例1：

|  |
| --- |
| int i, offset;  int pos = i + offset; // 危险，有符号整形相加可能会导致符号反转  解决方法：  int i, offset;  if(i>0 && INT\_MAX – i < offset || i<0 && INT\_MIN – i > offset){  return;  }  int pos = i + offset; |

示例2：

|  |
| --- |
| void mem\_op(char\* str, int len){  char buf[BUFF\_LEN];  if(BUFF\_LEN <= len){  return;  }  memcpy(buf, str, len); // 危险，memcpy的第三个参数size\_t是无符号整数，如果len为负数，则会反转成一个很大的unsigned int  buf[len] = ‘\0’;  } |

示例3：

|  |
| --- |
| unsigned int a = 100;  if(a < -1){ // -1被隐式转换成一个很大的unsigned int  printf("Woo! %u\n", -1);  }  输出： |

### 系统调用

C通过system或popen执行系统命令，但不会对输入命令做安全检查，接受来自用户的外部输入作为cmd可能会导致严重的系统问题。

推荐两种替代方案：

1. 使用POSIX exec系列函数替代system
2. 使用system时，对系统调用部分做硬编码，即把命令写死在程序中（类似cmd = “ls -l > record”; system(cmd)），通过白名单防止非法外部输入。