C高级

一、结构、联合和枚举

1、typedef 命令

`typedef` 命令用来为某个类型起别名。

```
typedef type name;
```

上面代码中，`type` 代表类型名，`name` 代表别名。

```
typedef unsigned char BYTE;
BYTE c = 'z';
```

上面示例中，`typedef` 命令为类型 `unsign char` 起别名 `BYTE`，然后就可以使用 `BYTE` 声明变量。

typedef 可以一次指定多个别名。

```
typedef int antelope, bagel, mushroom;
```

上面示例中，一次性为 `int` 类型起了三个别名。

typedef 可以为指针起别名。

```
typedef int* intptr;

int a = 10;
intptr x = &a;
```

上面示例中，`intptr` 是 `int*` 的别名。不过，使用的时候要小心，这样不容易看出来，变量 `x` 是一个指针类型。

typedef 也可以用来为数组类型起别名。

```
typedef int five_ints[5];

five_ints x = {11, 22, 33, 44, 55};
```

上面示例中，`five_ints` 是一个数组类型，包含 5 个整数的

typedef 为函数起别名的写法如下。

```
typedef signed char (*fp)(void);
```

上面示例中，类型别名 `fp` 是一个指针，代表函数 `signed char (*)(void)`。

`typedef` 为类型起别名的好处，主要有下面几点。

- 更好的代码可读性。
- 为 struct、union、enum 等命令定义的复杂数据结构创建别名，从而便于引用。

```
struct treenode {
  // ...
};
typedef struct treenode* Tree;
```

上面示例中，`Tree` 为 `struct treenode*` 的别名。

typedef 也可以与 struct 定义数据类型的命令写在一起。

```
typedef struct animal {
  char* name;
  int leg_count, speed;
} animal;
```

上面示例中，自定义数据类型时，同时使用 `typedef` 命令，为 `struct animal` 起了一个别名 `animal` 。

这种情况下，C 语言允许省略 struct 命令后面的类型名。

```
typedef struct {
  char *name;
  int leg_count, speed;
} animal;
```

上面示例相当于为一个匿名的数据类型起了别名 `animal` 。

- typedef 方便以后为变量改类型。

```
typedef float app_float;

app_float f1, f2, f3;
```

上面示例中，变量 `f1` 、 `f2` 、 `f3` 的类型都是 `float` 。如果以后需要为它们改类型，只需要修改 `typedef` 语句即可。

```
typedef long double app_float;
```

上面命令将变量 `f1` 、 `f2` 、 `f3` 的类型都改为 `long double` 。

- 可移植性

某一个值在不同计算机上的类型，可能是不一样的。

```
int i = 100000;
```

上面代码在 32 位整数的计算机没有问题，但是在 16 位整数的计算机就会出错。

C 语言的解决办法，就是提供了类型别名，在不同计算机上会解释成不同类型，比如 `int32_t` 。

```
int32_t i = 100000;
```

上面示例将变量 `i` 声明成 `int32_t` 类型，保证它在不同计算机上都是 32 位宽度，移植代码时就不会出错。

这一类的类型别名都是用 typedef 定义的。下面是类似的例子。

```
typedef long int ptrdiff_t;
typedef unsigned long int size_t;
typedef int wchar_t;
```

这些整数类型别名都放在头文件 `stdint.h`，不同架构的计算机只需修改这个头文件即可，而无需修改代码。

因此，`typedef` 有助于提高代码的可移植性，使其能适配不同架构的计算机。

- 简化类型声明

C 语言有些类型声明相当复杂，比如下面这个。

```
char (*(*x(void))[5])(void);
```

typedef 可以简化复杂的类型声明，使其更容易理解。首先，最外面一层起一个类型别名。

```
typedef char (*Func)(void);
Func (*x(void))[5];
```

这个看起来还是有点复杂，就为里面一层也定义一个别名。

```
typedef char (*Func)(void);
typedef Func Arr[5];
Arr* x(void);
```

上面代码就比较容易解读了。

- `x` 是一个函数，返回一个指向 Arr 类型的指针。
- `Arr` 是一个数组，有 5 个成员，每个成员是 `Func` 类型。
- `Func` 是一个函数指针，指向一个无参数、返回字符值的函数。

2、struct 结构

1）结构变量

声明格式：

```
struct {
  成员 1;
  成员 2;
  ...
} name1,name2;
-----------------------------------
struct{
  int name;
  char sex;
  int id;
} person1,person2;
```

结构的成员在内存中是按照声明的顺序存储的。每一个结构代表一种新的作用域，任何声明在作用域中的名字不会与其他作用域中的名字冲突。

初始化：

struct 也可以在声明的同时进行初始化

```
struct{
    int num;
    char sex;
    int id;
}p1={1,'f',101},
p2={2,'m',102};
```

也可以指定初始化,通过点运算加上成员名

```
struct{
    int num;
    char sex;
    int id;
}p1={.num=1,.sex='f',.id=101},
p2={2,'m',102};
```

2）结构类型

> 想要大量的定义结构数据，需要通过定义结构类型。

结构标记声明：

结构标记用于标识某种特定结构的名字。

```
struct person{
  int num;
  char sex;
  int id;
};//p1,p2;
struct person p1,p2;
```

创建了标记就可以用来声明变量。

结构标记声明可以和变量一起

结构类型的定义：

可以通过 typedef 命令来定义结构类型

```
typedef struct{
  int num;
  char sex;
  int id;
}person;
```

经过定义后，就可以像内置类型一样来声明变量

```
person p1,p2;
```

3）结构指针

如果将 struct 变量传入函数，函数内部得到的是一个原始值的副本。

```
#include <stdio.h>

struct turtle {
  char* name;
  char* species;
  int age;
};

void happy(struct turtle t) {
  t.age = t.age + 1;
}

int main() {
  struct turtle myTurtle = {"MyTurtle", "sea turtle", 99};
  happy(myTurtle);
  printf("Age is %i\n", myTurtle.age); // 输出 99
  return 0;
}
```

上面示例中，函数 `happy()` 传入的是一个 struct 变量 `myTurtle` ，函数内部有一个自增操作。但是，执行完 `happy()` 以后，函数外部的 `age` 属性值根本没变。原因就是函数内部得到的是 struct 变量的副本，改变副本影响不到函数外部的原始数据。

通常情况下，开发者希望传入函数的是同一份数据，函数内部修改数据以后，会反映在函数外部。而且，传入的是同一份数据，也有利于提高程序性能。**这时就需要将 struct 变量的指针传入函数，通过指针来修改 struct 属性，就可以影响到函数外部。**

struct 指针传入函数的写法如下。

```
void happy(struct turtle* t) {
}
happy(&myTurtle);
```

上面代码中， `t` 是 struct 结构的指针，调用函数时传入的是指针。struct 类型跟数组不一样，**类型标识符本身并不是指针，所以传入时，指针必须写成** `&myTurtle` 。

函数内部也必须使用 `(*t).age` **的写法，从指针拿到 struct 结构本身。**

```
void happy(struct turtle* t) {
  (*t).age = (*t).age + 1;
}
```

上面示例中， `(*t).age` 不能写成 `*t.age` ，因为点运算符 `.` 的优先级高于 `*` 。 `*t.age` 这种写法会将 `t.age` 看成一个指针，然后取它对应的值，会出现无法预料的结果。

现在，重新编译执行上面的整个示例， `happy()` 内部对 struct 结构的操作，就会反映到函数外部。

`(*t).age` 这样的写法很麻烦。C 语言就引入了一个新的箭头运算符（`->`），可以从 struct 指针上直接获取属性，大大增强了代码的可读性。

```
void happy(struct turtle* t) {
  t->age = t->age + 1;
}
```

对于 struct 变量名，使用点运算符（`.`）获取属性；对于 struct 变量指针，使用箭头运算符（`->`）获取属性。以变量 `myStruct` 为例，假设 `ptr` 是它的指针，那么下面三种写法是同一回事。

```
// ptr == &myStruct
myStruct.prop == (*ptr).prop == ptr->prop
```

4）结构操作

读取：

对于数组的操作是读取下标，根据位置选择元素，然而结构中成员的操作是根据名字获取的

```
printf("%5d%5c%5d",p1.num,p1.sex,p1.id);
printf("%5d%5c%5d",p2.num,p2.sex,p2.id);
```

复制：

struct 变量可以使用赋值运算符（`=`），复制给另一个变量，这时会生成一个全新的副本。系统会分配一块新的内存空间，大小与原来的变量相同，把每个属性都复制过去，即原样生成了一份数据。这一点跟数组的复制不一样。

```
struct cat { char name[30]; short age; } a, b;
strcpy(a.name, "Hula");
a.age = 3;
b = a;
b.name[0] = 'M';
printf("%s\n", a.name); // Hula
printf("%s\n", b.name); // Mula
```

上面示例中，变量 `b` 是变量 `a` 的副本，两个变量的值是各自独立的，修改掉 `b.name` 不影响 `a.name`。

上面这个示例是有前提的，就是 struct 结构的属性必须定义成字符数组，才能复制数据。如果稍作修改，属性定义成字符指针，结果就不一样。

```
struct cat { char* name; short age; } a, b;

a.name = "Hula";
a.age = 3;

b = a;
```

上面示例中，`name` 属性变成了一个字符指针，这时 `a` 赋值给 `b`，导致 `b.name` 也是同样的字符指针，指向同一个地址，也就是说两个属性共享同一个地址。因为这时，struct 结构内部保存的是一个指针，而不是上一个例子的数组，这时复制的就不是字符串本身，而是它的指针。并且，这个时候也没法修改字符串，因为字符指针指向的字符串是不能修改的。

赋值运算符（`=`）可以将 struct 结构每个属性的值，一模一样复制一份，拷贝给另一个 struct 变量。这一点跟数组完全不同，使用赋值运算符复制数组，不会复制数据，只会共享地址。

注意，这种赋值要求两个变量是同一个类型，不同类型的 struct 变量无法互相赋值。

另外，C 语言没有提供比较两个自定义数据结构是否相等的方法，无法用比较运算符（比如 `==` 和 `!=`）比较两个数据结构是否相等或不等。

5）复合字面量和匿名结构

复合字面量：

结构也能定义字面量，可以创建一个实时的结构数据，不需要存储在变量当中，生成的结构可以像参数一样传递，可以作为返回值，也可以赋值给变量。

```
//传递参数
print_struct((struct name){成员值 1,成员值 2});
//赋值给变量
print_struct=(struct name){成员值 1,成员值 2};
```

匿名结构：

如果一个结构或联合包含了以下成员：

- 没有名称
- 没有声明为结构类型，只有成员而没有标记

那么这个成员就是一个匿名结构

```
struct t {int i;struct{char c;float f}};
union u {int i;struct{char c;float f}};
```

访问匿名结构的成员：如果匿名结构 S 是结构或联合 X 的成员，那么 S 的成员就默认为 X 的成员，对于多层嵌套的关系可以递归的应用关系。

6）嵌套数组和结构

嵌套结构：

struct 结构的成员可以是另一个 struct 结构。

```
struct species {
  char* name;
  int kinds;
};
struct fish {
  char* name;
```

```
  int age;
  struct species breed;
};
```

上面示例中，`fish` 的属性 `breed` 是另一个 struct 结构 `species` 。

赋值的时候有多种写法。

```
// 写法一
struct fish shark = {"shark", 9, {"Selachimorpha", 500}};
// 写法二
struct species myBreed = {"Selachimorpha", 500};
struct fish shark = {"shark", 9, myBreed};
// 写法三
struct fish shark = {
  .name="shark",
  .age=9,
  .breed={"Selachimorpha", 500}
};
// 写法四
struct fish shark = {
  .name="shark",
  .age=9,
  .breed.name="Selachimorpha",
  .breed.kinds=500
};
printf("Shark's species is %s", shark.breed.name);
```

上面示例展示了嵌套 Struct 结构的四种赋值写法。另外，引用 `breed` 属性的内部属性，要使用两次点运算符（`shark.breed.name`）。

下面是另一个嵌套 struct 的例子。

```
struct name {
  char first[50];
  char last[50];
};
struct student {
  struct name name;
  short age;
  char sex;
} student1;
strcpy(student1.name.first, "Harry");
strcpy(student1.name.last, "Potter");
// or
struct name myname = {"Harry", "Potter"};
student1.name = myname;
```

上面示例中，自定义类型 `student` 的 `name` 属性是另一个自定义类型，如果要引用后者的属性，就必须使用两个 `.` 运算符，比如 `student1.name.first` 。另外，对字符数组

属性赋值，要使用 `strcpy()` 函数，不能直接赋值，因为直接改掉字符数组名的地址会报错。

struct 结构内部不仅可以引用其他结构，还可以自我引用，即结构内部引用当前结构。比如，链表结构的节点就可以写成下面这样。

```
struct node {
  int data;
  struct node* next;
};
```

上面示例中，`node` 结构的 `next` 属性，就是指向另一个 `node` 实例的指针。下面，使用这个结构自定义一个数据链表。

```
struct node {
  int data;
  struct node* next;
};

struct node* head;

// 生成一个三个节点的列表 (11)->(22)->(33)
head = malloc(sizeof(struct node));

head->data = 11;
head->next = malloc(sizeof(struct node));

head->next->data = 22;
head->next->next = malloc(sizeof(struct node));

head->next->next->data = 33;
head->next->next->next = NULL;

// 遍历这个列表
for (struct node *cur = head; cur != NULL; cur = cur->next) {
  printf("%d\n", cur->data);
}
```

上面示例是链表结构的最简单实现，通过 `for` 循环可以对其进行遍历。

结构数组：

数组和结构常见的组合就是其元素为结构的数组。这类数组可以用作简单的数据库。

```
struct name array[]={{},{},{}};
```

结构数组的初始化：

初始化结构数组和初始化多维数组的方法非常相似，每个结构都拥有自己的带有花括号的初始化器，数组的初始化器简单地在结构初始化器的外围加上另一对花括号。

7）位字段

struct 还可以用来定义二进制位组成的数据结构，称为“位字段”（bit field），这对于操作底层的二进制数据非常有用。

```
struct {
  unsigned int ab:1;
  unsigned int cd:1;
  unsigned int ef:1;
  unsigned int gh:1;
} synth;

synth.ab = 0;
synth.cd = 1;
```

上面示例中，每个属性后面的 `:1` ，表示指定这些属性只占用一个二进制位，所以这个数据结构一共是 4 个二进制位。

注意，定义二进制位时，结构内部的各个属性只能是整数类型。

实际存储的时候，C 语言会按照 `int` 类型占用的字节数，存储一个位字段结构。如果有剩余的二进制位，可以使用未命名属性，填满那些位。也可以使用宽度为 0 的属性，表示占满当前字节剩余的二进制位，迫使下一个属性存储在下一个字节。

```
struct {
  unsigned int field1 : 1;
  unsigned int        : 2;
  unsigned int field2 : 1;
  unsigned int        : 0;
  unsigned int field3 : 1;
} stuff;
```

上面示例中， `stuff.field1` 与 `stuff.field2` 之间，有一个宽度为两个二进制位的未命名属性。 `stuff.field3` 将存储在下一个字节。

8）弹性数组成员

很多时候，不能事先确定数组到底有多少个成员。如果声明数组的时候，事先给出一个很大的成员数，就会很浪费空间。C 语言提供了一个解决方法，叫做弹性数组成员（flexible array member）。

如果不能事先确定数组成员的数量时，可以定义一个 struct 结构。

```
struct vstring {
  int len;
  char chars[];
};
```

上面示例中， `struct vstring` 结构有两个属性。 `len` 属性用来记录数组 `chars` 的长度， `chars` 属性是一个数组，但是没有给出成员数量。

`chars` 数组到底有多少个成员，可以在为 `vstring` 分配内存时确定。

```
struct vstring* str = malloc(sizeof(struct vstring) + n * sizeof(char));
```

```
str->len = n;
```

上面示例中，假定 `chars` 数组的成员数量是 `n`，只有在运行时才能知道 `n` 到底是多少。然后，就为 `struct vstring` 分配它需要的内存：它本身占用的内存长度，再加上 `n` 个数组成员占用的内存长度。最后，`len` 属性记录一下 `n` 是多少。

这样就可以让数组 `chars` 有 `n` 个成员，不用事先确定，可以跟运行时的需要保持一致。

弹性数组成员有一些专门的规则。首先，弹性成员的数组，必须是 struct 结构的最后一个属性。另外，除了弹性数组成员，struct 结构必须至少还有一个其他属性。

3、union 联合

union：

有时需要一种数据结构，不同的场合表示不同的数据类型。

C 语言提供了 Union 结构，用来自定义可以灵活变更的数据结构。它内部包含各种属性，但是所有属性共用一块内存，导致这些属性都是对同一个二进制数据的解读，其中往往只有一个属性的解读是有意义的。并且，后面写入的属性会覆盖前面的属性，这意味着同一块内存，可以先供某一个属性使用，然后再供另一个属性使用。这样做的最大好处是节省内存空间。

```
union quantity {
  short count;
  float weight;
  float volume;
};
```

上面示例中，`union` 命令定义了一个包含三个属性的数据类型 `quantity`。虽然包含三个属性，但是只能写入一个值，三个属性都是对这个值的不同解读。最后赋值的属性，往往就是可以取到有意义的值的那个属性。

使用时，声明一个该类型的变量。

```
// 写法一
union quantity q;
q.count = 4;

// 写法二
union quantity q = {.count=4};

// 写法三
union quantity q = {4};
```

上面代码展示了为 Union 结构赋值的三种写法。最后一种写法不指定属性名，就会赋值给第一个属性。

执行完上面的代码以后，`q.count` 可以取到值，另外两个属性取不到值。

```
printf("count is %i\n", q.count); // count is 4
```

```
printf("weight is %f\n", q.weight); // 未定义行为
```

如果要让 `q.weight` 属性可以取到值，就要先为它赋值。

```
q.weight = 0.5;
printf("weight is %f\n", q.weight); // weight is 0.5
```

一旦为其他属性赋值，原先可以取到值的 `q.count` 属性就跟着改变，使用它可能就没有意义了。除了这一点，Union 结构的其他用法与 Struct 结构，基本上是一致的。

Union 结构也支持指针运算符 `->` 。

```
union quantity {
  short count;
  float weight;
  float volume;
};

union quantity q;
q.count = 4;

union quantity* ptr;
ptr = &q;

printf("%d\n", ptr->count); // 4
```

上面示例中， `ptr` 是 `q` 的指针，那么 `ptr->count` 等同于 `q.count` 。

Union 结构指针与它的属性有关，当前正在按照哪个属性解读数据，它的指针就是对应的数据类型。

```
union foo {
  int a;
  float b;
} x;

int* foo_int_p = (int *)&x;
float* foo_float_p = (float *)&x;

x.a = 12;
printf("%d\n", x.a);           // 12
printf("%d\n", *foo_int_p);    // 12

x.b = 3.141592;
printf("%f\n", x.b);           // 3.141592
printf("%f\n", *foo_float_p);  // 3.141592
```

上面示例中， `&x` 是 foo 结构的指针，它的数据类型完全由当前赋值的属性决定。

typedef 命令可以为 Union 数据类型起别名。

```
typedef union {
  short count;
```

```
  float weight;
  float volume;
} quantity;
```

上面示例中，`union` 命令定义了一个包含三个属性的数据类型，`typedef` 命令为它起别名为 `quantity` 。

Union 结构的好处，主要是节省空间。它将一段内存空间，重用于不同类型的数据。定义了三个属性，但同一时间只用到一个，使用 Union 结构就可以节省另外两个属性的空间。Union 结构占用的内存长度，等于它内部最长属性的长度。

用联合构造混合数据结构：

联合主要的作用是节省空间。还有一个重要的作用就是创建含有不同类型混合数据的数据结构。比如创建一个多类型元素的数组。

```
typedef union{
  int i;
  float f;
}number;
number arr[100];
```

这样数组就可以创建一个混合数组，这个数组的实际长度是 200，(int 类型+float 类型子成员)

4、enum 枚举

如果一种数据类型的取值只有少数几种可能，并且每种取值都有自己的含义，为了提高代码的可读性，可以将它们定义为 Enum 类型，中文名为枚举。

```
enum colors {RED, GREEN, BLUE};

printf("%d\n", RED); // 0
printf("%d\n", GREEN);  // 1
printf("%d\n", BLUE);  // 2
```

上面示例中，假定程序里面需要三种颜色，就可以使用 `enum` 命令，把这三种颜色定义成一种枚举类型 `colors` ，它只有三种取值可能 `RED` 、 `GREEN` 、 `BLUE` 。这时，这三个名字自动成为整数常量，编译器默认将它们的值设为数字 `0` 、 `1` 、 `2` 。相比之下， `RED` 要比 `0` 的可读性好了许多。

注意，Enum 内部的常量名，遵守标识符的命名规范，但是通常都使用大写。

使用时，可以将变量声明为 Enum 类型。

```
enum colors color;
```

上面代码将变量 `color` 声明为 `enum colors` 类型。这个变量的值就是常量 `RED` 、 `GREEN` 、 `BLUE` 之中的一个。

```
color = BLUE;
printf("%i\n", color); // 2
```

上面代码将变量 `color` 的值设为 `BLUE`，这里 `BLUE` 就是一个常量，值等于 `2`。

typedef 命令可以为 Enum 类型起别名。

```
typedef enum {
  SHEEP,
  WHEAT,
  WOOD,
  BRICK,
  ORE
} RESOURCE;

RESOURCE r;
```

上面示例中，`RESOURCE` 是 Enum 类型的别名。声明变量时，使用这个别名即可。

还有一种不常见的写法，就是声明 Enum 类型时，在同一行里面为变量赋值。

```
enum {
  SHEEP,
  WHEAT,
  WOOD,
  BRICK,
  ORE
} r = BRICK, s = WOOD;
```

上面示例中，`r` 的值是 `3`，`s` 的值是 `2`。

由于 Enum 的属性会自动声明为常量，所以有时候使用 Enum 的目的，不是为了自定义一种数据类型，而是为了声明一组常量。这时就可以使用下面这种写法，比较简单。

```
enum { ONE, TWO };

printf("%d %d", ONE, TWO);  // 0 1
```

上面示例中，`enum` 是一个关键字，后面跟着一个代码块，常量就在代码内声明。`ONE` 和 `TWO` 就是两个 Enum 常量。

常量之间使用逗号分隔。最后一个常量后面的尾逗号，可以省略，也可以保留。

```
enum { ONE, TWO, };
```

由于 Enum 会自动编号，因此可以不必为常量赋值。C 语言会自动从 0 开始递增，为常量赋值。但是，C 语言也允许为 ENUM 常量指定值，不过只能指定为整数，不能是其他类型。因此，任何可以使用整数的场合，都可以使用 Enum 常量。

```
enum { ONE = 1, TWO = 2 };

printf("%d %d", ONE, TWO);  // 1 2
```

Enum 常量可以是不连续的值。

```
enum { X = 2, Y = 18, Z = -2 };
```

Enum 常量也可以是同一个值。

```
enum { X = 2, Y = 2, Z = 2 };
```

如果一组常量之中，有些指定了值，有些没有指定。那么，没有指定值的常量会从上一个指定了值的常量，开始自动递增赋值。

```
enum {
  A,    // 0
  B,    // 1
  C = 4,  // 4
  D,    // 5
  E,    // 6
  F = 3,   // 3
  G,    // 4
  H     // 5
};
```

Enum 的作用域与变量相同。如果是在顶层声明，那么在整个文件内都有效；如果是在代码块内部声明，则只对该代码块有效。如果与使用 `int` 声明的常量相比，Enum 的好处是更清晰地表示代码意图。

二、高级指针–内存管理

C 语言的内存管理，分成两部分。一部分是系统管理的，另一部分是用户手动管理的。
系统管理的内存，主要是函数内部的变量（局部变量）。这部分变量在函数运行时进入内存，函数运行结束后自动从内存卸载。这些变量存放的区域称为”栈“（stack），”栈“所在的内存是系统自动管理的。
用户手动管理的内存，主要是程序运行的整个过程中都存在的变量（全局变量），这些变量需要用户手动从内存释放。如果使用后忘记释放，它就一直占用内存，直到程序退出，这种情况称为”内存泄漏“（memory leak）。这些变量所在的内存称为”堆“（heap），”堆“所在的内存是用户手动管理的。

1. void 空指针

每一块内存都有地址，通过指针变量可以获取指定地址的内存块。指针变量必须有类型，否则编译器无法知道，如何解读内存块保存的二进制数据。但是，向系统请求内存的时候，有时不确定会有什么样的数据写入内存，需要先获得内存块，稍后再确定写入的数据类型。

为了满足这种需求，**C 语言提供了一种不定类型的指针，叫做 void 指针。它只有内存块的地址信息，没有类型信息，等到使用该块内存的时候，再向编译器补充说明，里面的数据类型是什么。**

另一方面，void 指针等同于无类型指针，可以指向任意类型的数据，但是不能解读数据。void 指针与其他所有类型指针之间是互相转换关系，任一类型的指针都可以转为 void 指针，而 void 指针也可以转为任一类型的指针。

```
int x = 10;

void* p = &x; // 整数指针转为 void 指针
int* q = p; // void 指针转为整数指针
```

上面示例演示了，整数指针和 void 指针如何互相转换。`&x` 是一个整数指针，`p` 是 void 指针，赋值时 `&x` 的地址会自动解释为 void 类型。同样的，`p` 再赋值给整数指针 `q` 时，`p` 的地址会自动解释为整数指针。

注意，由于不知道 void 指针指向什么类型的值，所以不能用 `*` 运算符取出它指向的值。

```
char a = 'X';
void* p = &a;

printf("%c\n", *p); // 报错
```

上面示例中，`p` 是一个 void 指针，所以这时无法用 `*p` 取出指针指向的值。

void 指针的重要之处在于，很多内存相关函数的返回值就是 void 指针，只给出内存块的地址信息，所以放在最前面进行介绍。

2. 内存分配函数

malloc 函数：

`malloc()` 函数用于分配内存，该函数向系统要求一段内存，系统就在“堆”里面分配一段连续的内存块给它。它的原型定义在头文件 `stdlib.h` 。

```
void* malloc(size_t size)
```

它接受一个非负整数作为参数，表示所要分配的内存字节数，返回一个 void 指针，指向分配好的内存块。这是非常合理的，因为 `malloc()` 函数不知道，将要存储在该块内存的数据是什么类型，所以只能返回一个无类型的 void 指针。

可以使用 `malloc()` 为任意类型的数据分配内存，常见的做法是先使用 `sizeof()` 函数，算出某种数据类型所需的字节长度，然后再将这个长度传给 `malloc()` 。

```
int* p = malloc(sizeof(int));
*p = 12;
printf("%d\n", *p); // 12
```

上面示例中，先为整数类型分配一段内存，然后将整数 `12` 放入这段内存里面。这个例子其实不需要使用 `malloc()` ，因为 C 语言会自动为整数（本例是 `12` ）提供内存。

有时候为了增加代码的可读性，可以对 `malloc()` 返回的指针进行一次强制类型转换。

```
int* p = (int*) malloc(sizeof(int));
```

上面代码将 `malloc()` 返回的 void 指针，强制转换成了整数指针。

`malloc()` 分配内存有可能分配失败，这时返回常量 NULL。Null 的值为 0，是一个无法读写的内存地址，可以理解成一个不指向任何地方的指针。它在包括 `stdlib.h` 等多个

头文件里面都有定义，所以只要可以使用 `malloc()`，就可以使用 `NULL`。由于存在分配失败的可能，所以最好在使用 `malloc()` 之后检查一下，是否分配成功。

```
int* p = malloc(sizeof(int));
if (p == NULL) {
  // 内存分配失败
}
// or
if (!p) {
  //...
}
```

上面示例中，通过判断返回的指针 `p` 是否为 `NULL`，确定 `malloc()` 是否分配成功。

`malloc()` 最常用的场合，就是为数组和自定义数据结构分配内存。

```
int* p = (int*) malloc(sizeof(int) * 10);

for (int i = 0; i < 10; i++)
  p[i] = i * 5;
```

上面示例中，`p` 是一个整数指针，指向一段可以放置 10 个整数的内存，所以可以用作数组。

`malloc()` 用来创建数组，有一个好处，就是它可以创建动态数组，即根据成员数量的不同，而创建长度不同的数组。

```
int* p = (int*) malloc(n * sizeof(int));
```

上面示例中，`malloc()` 可以根据变量 `n` 的不同，动态为数组分配不同的大小。

注意，`malloc()` **不会对所分配的内存进行初始化，里面还保存着原来的值。如果没有初始化，就使用这段内存，可能从里面读到以前的值。程序员要自己负责初始化**，比如，字符串初始化可以使用 `strcpy()` 函数。

```
char* p = malloc(4);
strcpy(p, "abc");

// or
p = "abc";
```

上面示例中，字符指针 `p` 指向一段 4 个字节的内存，`strcpy()` 将字符串“abc”拷贝放入这段内存，完成了这段内存的初始化。

free 函数：

`free()` 用于释放 `malloc()` 函数分配的内存，将这块内存还给系统以便重新使用，否则这个内存块会一直占用到程序运行结束。该函数的原型定义在头文件 `stdlib.h` 里面。

```
void free(void* block)
```

上面代码中，`free()` 的参数是 `malloc()` 返回的内存地址。下面就是用法实例。

```
int* p = (int*) malloc(sizeof(int));

*p = 12;
free(p);
```

注意，分配的内存块一旦释放，就不应该再次操作已经释放的地址，也不应该再次使用 `free()` 对该地址释放第二次。

一个很常见的错误是，在函数内部分配了内存，但是函数调用结束时，没有使用 `free()` 释放内存。

```
void gobble(double arr[], int n) {
  double* temp = (double*) malloc(n * sizeof(double));
  // ...
}
```

上面示例中，函数 `gobble()` 内部分配了内存，但是没有写 `free(temp)` 。这会造成函数运行结束后，占用的内存块依然保留，如果多次调用 `gobble()` ，就会留下多个内存块。并且，由于指针 `temp` 已经消失了，也无法访问这些内存块，再次使用。

calloc 函数：

`calloc()` 函数的作用与 `malloc()` 相似，也是分配内存块。该函数的原型定义在头文件 `stdlib.h` 。

两者的区别主要有两点：

（1） `calloc()` 接受两个参数，第一个参数是某种数据类型的值的数量，第二个是该数据类型的单位字节长度。

```
void* calloc(size_t n, size_t size);
```

`calloc()` *的返回值也是一个 void 指针。分配失败时，返回 NULL。*

（2） `calloc()` 会将所分配的内存全部初始化为 `0` 。 `malloc()` 不会对内存进行初始化，如果想要初始化为 `0` ，还要额外调用 `memset()` 函数。

```
int* p = calloc(10, sizeof(int));

// 等同于
int* p = malloc(sizeof(int) * 10);
memset(p, 0, sizeof(int) * 10);
```

上面示例中， `calloc()` 相当于 `malloc() + memset()` 。

`calloc()` 分配的内存块，也要使用 `free()` 释放。

realoc 函数：

`realloc()` 函数用于修改已经分配的内存块的大小，可以放大也可以缩小，返回一个指向新的内存块的指针。如果分配不成功，返回 NULL。该函数的原型定义在头文件 `stdlib.h` 。

```
void* realloc(void* block, size_t size)
```

它接受两个参数。

- `block`：已经分配好的内存块指针（由 `malloc()` 或 `calloc()` 或 `realloc()` 产生）。
- `size`：该内存块的新大小，单位为字节。

`realloc()` 可能返回一个全新的地址（数据也会自动复制过去），也可能返回跟原来一样的地址。`realloc()` 优先在原有内存块上进行缩减，尽量不移动数据，所以通常是返回原先的地址。如果新内存块小于原来的大小，则丢弃超出的部分；如果大于原来的大小，则不对新增的部分进行初始化（程序员可以自动调用 `memset()`）。

下面是一个例子，`b` 是数组指针，`realloc()` 动态调整它的大小。

```
int* b;

b = malloc(sizeof(int) * 10);
b = realloc(b, sizeof(int) * 2000);
```

上面示例中，指针 `b` 原来指向 10 个成员的整数数组，使用 `realloc()` 调整为 2000 个成员的数组。这就是手动分配数组内存的好处，可以在运行时随时调整数组的长度。

`realloc()` 的第一个参数可以是 NULL，这时就相当于新建一个指针。

```
char* p = realloc(NULL, 3490);
// 等同于
char* p = malloc(3490);
```

如果 `realloc()` 的第二个参数是 `0`，就会释放掉内存块。

由于有分配失败的可能，所以调用 `realloc()` 以后，最好检查一下它的返回值是否为 NULL。分配失败时，原有内存块中的数据不会发生改变。

```
float* new_p = realloc(p, sizeof(*p * 40));

if (new_p == NULL) {
  printf("Error reallocing\n");
  return 1;
}
```

注意，`realloc()` 不会对内存块进行初始化。

3. 动态分配字符串

动态内存分配对字符串操作非常有作用，字符串储存在字符数组中，而且很可能难以预测这些数组需要的长度。通过动态分配字符串，可以延迟程序运行时再确定长度。

用 malloc 给字符串分配内存是容易的。char 类型的值刚好需要一个字节的内存，为 n 个字符的字符串分配内存：

```
p=malloc(n+1);
```

这里的 p 是 char*类型的变量（实际参数是 n+1 是给空字符留出了空间），在执行赋值操作时 malloc 函数返回的通用指针转换为 char*类型，而不需要强制类型转换。当然也可以对返回值执行强制类型转换

```
p=(char*)malloc(n+1);
```

使用 malloc 函数分配的内存不会进行清零或初始化，可以调用 strcpy()函数解决初始化问题。

动态存储分配可以编写指向“未存在”字符串的指针的函数，也就是函数在调用之前字符串并不存在。

4. 动态分配数组

可以使用 malloc 函数为数组分配存储空间，需要用到 sizeof 运算符测量每一元素需要的空间，再乘以元素的个数。

```
p=malloc(n*sizeof(int));
```

虽然 malloc 函数的效率比较高，但是在某种情况下 calloc 函数更加适合，calloc 函数会在分配了内存之后就把所有位设置为 0 进行初始化。

5. 释放存储

free 函数：

`free()` 用于释放 `malloc()` 函数分配的内存，将这块内存还给系统以便重新使用，否则这个内存块会一直占用到程序运行结束。该函数的原型定义在头文件 `stdlib.h` 里面。

```
void free(void* block)
```

上面代码中，`free()` 的参数是 `malloc()` 返回的内存地址。下面就是用法实例。

```
int* p = (int*) malloc(sizeof(int));

*p = 12;
free(p);
```

注意，分配的内存块一旦释放，就不应该再次操作已经释放的地址，也不应该再次使用 `free()` 对该地址释放第二次。

悬空指针：

调用 free 函数，会造成一个问题，调用函数会释放内存块，但不会改变指针本身，指针本身还是存在的，但是不指向任何存储空间，这样就造成了悬空指针

```
char *p=malloc(n+1);
free(p);
strcpy(p,"abc");//err
```

释放内存后再修改指向的内存是错误的，因为程序不再对改内存区域有任何控制权。

6. restrict 说明符--受限指针

声明指针变量时，可以使用 `restrict` 说明符，告诉编译器，该块内存区域只有当前指针一种访问方式，其他指针不能读写该块内存。这种指针称为“受限指针”（restrict pointer）。

```
int* restrict p;
p = malloc(sizeof(int));
```

上面示例中，声明指针变量 `p` 时，加入了 `restrict` 说明符，使得 `p` 变成了受限指针。后面，当 `p` 指向 `malloc()` 函数返回的一块内存区域，就味着，该区域只有通过 `p` 来访问，不存在其他访问方式。

```
int* restrict p;
p = malloc(sizeof(int));

int* q = p;
*q = 0; // 未定义行为
```

上面示例中，另一个指针 `q` 与受限指针 `p` 指向同一块内存，现在该内存有 `p` 和 `q` 两种访问方式。这就违反了对编译器的承诺，后面通过 `*q` 对该内存区域赋值，会导致未定义行为。

7. 内存操作函数

1、memcpy()函数

`memcpy()` 用于将一块内存拷贝到另一块内存。该函数的原型定义在头文件 `string.h` 。

```
void* memcpy(
  void* restrict dest,
  void* restrict source,
  size_t n
);
```

上面代码中， `dest` 是目标地址， `source` 是源地址，第三个参数 `n` 是要拷贝的字节数 `n` 。如果要拷贝 10 个 double 类型的数组成员， `n` 就等于 `10 * sizeof(double)` ，而不是 `10` 。该函数会将从 `source` 开始的 `n` 个字节，拷贝到 `dest` 。

`dest` 和 `source` 都是 void 指针，表示这里不限制指针类型，各种类型的内存数据都可以拷贝。两者都有 restrict 关键字，表示这两个内存块不应该有互相重叠的区域。

`memcpy()` 的返回值是第一个参数，即目标地址的指针。

因为 `memcpy()` 只是将一段内存的值，复制到另一段内存，所以不需要知道内存里面的数据是什么类型。下面是复制字符串的例子。

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char s[] = "Goats!";
  char t[100];
```

```
  memcpy(t, s, sizeof(s));  // 拷贝 7 个字节，包括终止符

  printf("%s\n", t);  // "Goats!"

  return 0;
}
```

上面示例中，字符串 `s` 所在的内存，被拷贝到字符数组 `t` 所在的内存。

`memcpy()` 可以取代 `strcpy()` 进行字符串拷贝，而且是更好的方法，不仅更安全，速度也更快，它不检查字符串尾部的 `\0` 字符。

```
char* s = "hello world";

size_t len = strlen(s) + 1;
char *c = malloc(len);

if (c) {
  // strcpy() 的写法
  strcpy(c, s);

  // memcpy() 的写法
  memcpy(c, s, len);
}
```

上面示例中，两种写法的效果完全一样，但是 `memcpy()` 的写法要好于 `strcpy()` 。

使用 void 指针，也可以自定义一个复制内存的函数。

```
void* my_memcpy(void* dest, void* src, int byte_count) {
  char* s = src;
  char* d = dest;

  while (byte_count--) {
    *d++ = *s++;
  }

  return dest;

}
```

上面示例中，不管传入的 `dest` 和 `src` 是什么类型的指针，将它们重新定义成一字节的 Char 指针，这样就可以逐字节进行复制。 `*d++ = *s++` 语句相当于先执行 `*d = *s` （源字节的值复制给目标字节），然后各自移动到下一个字节。最后，返回复制后的 `dest` 指针，便于后续使用。

2、memmove()函数

`memmove()` 函数用于将一段内存数据复制到另一段内存。它跟 `memcpy()` 的主要区别是，它允许目标区域与源区域有重叠。如果发生重叠，源区域的内容会被更改；如果没

有重叠，它与 `memcpy()` 行为相同。

该函数的原型定义在头文件 `string.h` 。

```
void* memmove(
  void* dest,
  void* source,
  size_t n
);
```

上面代码中， `dest` 是目标地址， `source` 是源地址， `n` 是要移动的字节数。

`dest` 和 `source` 都是 void 指针，表示可以移动任何类型的内存数据，两个内存区域可以有重叠。

`memmove()` 返回值是第一个参数，即目标地址的指针。

```
int a[100];
// ...

memmove(&a[0], &a[1], 99 * sizeof(int));
```

上面示例中，从数组成员 `a[1]` 开始的 99 个成员，都向前移动一个位置。

下面是另一个例子。

```
char x[] = "Home Sweet Home";

// 输出 Sweet Home Home
printf("%s\n", (char *) memmove(x, &x[5], 10));
```

上面示例中，从字符串 `x` 的 5 号位置开始的 10 个字节，就是“Sweet Home”，`memmove()` 将其前移到 0 号位置，所以 `x` 就变成了“Sweet Home Home”。

3、memcmp()函数

`memcmp()` 函数用来比较两个内存区域。它的原型定义在 `string.h` 。

```
int memcmp(
  const void* s1,
  const void* s2,
  size_t n
);
```

它接受三个参数，前两个参数是用来比较的指针，第三个参数指定比较的字节数。

它的返回值是一个整数。两块内存区域的每个字节以字符形式解读，按照字典顺序进行比较，如果两者相同，返回 `0` ；如果 `s1` 大于 `s2` ，返回大于 0 的整数；如果 `s1` 小于 `s2` ，返回小于 0 的整数。

```
char* s1 = "abc";
char* s2 = "acd";
int r = memcmp(s1, s2, 3); // 小于 0
```

上面示例比较 `s1` 和 `s2` 的前三个字节，由于 `s1` 小于 `s2`，所以 `r` 是一个小于 0 的整数，一般为-1。

下面是另一个例子。

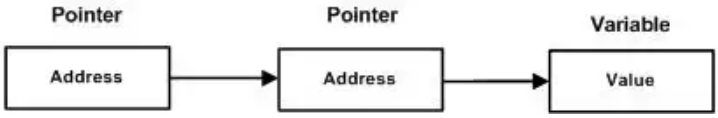
```
char s1[] = {'b', 'i', 'g', '\0', 'c', 'a', 'r'};
char s2[] = {'b', 'i', 'g', '\0', 'c', 'a', 't'};

if (memcmp(s1, s2, 3) == 0) // true
if (memcmp(s1, s2, 4) == 0) // true
if (memcmp(s1, s2, 7) == 0) // false
```

上面示例展示了，`memcmp()` 可以比较内部带有字符串终止符 `\0` 的内存区域。

8、指向指针的指针

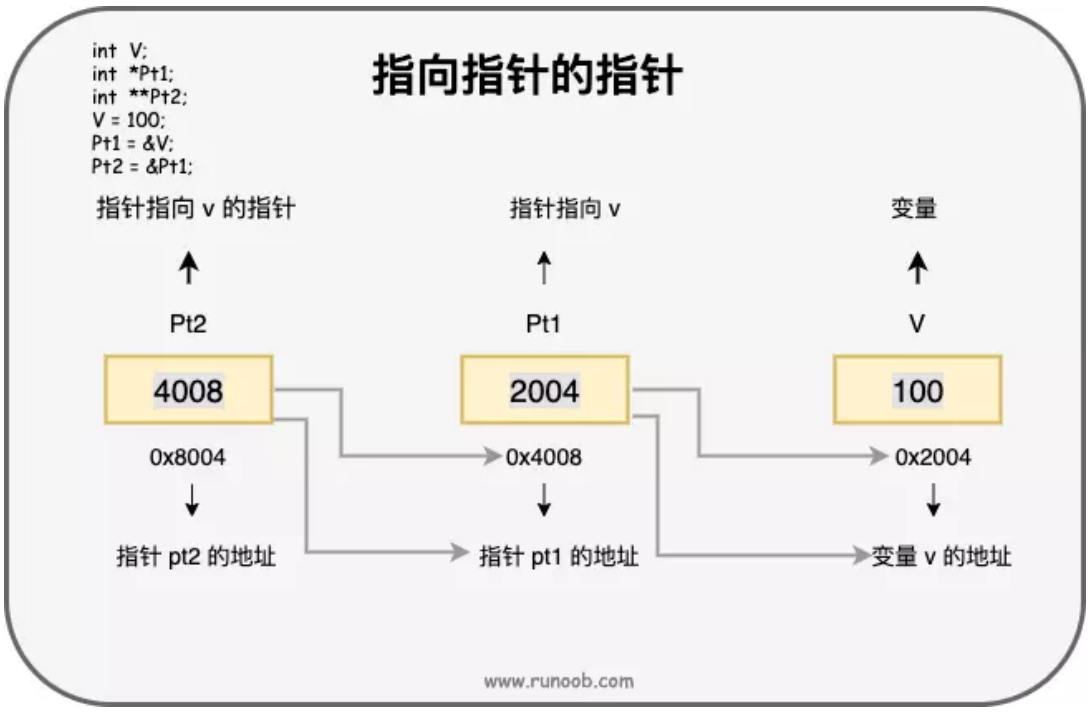
指向指针的指针是一种多级间接寻址的形式，或者说是一个指针链。通常，一个指针包含一个变量的地址。当我们定义一个指向指针的指针时，第一个指针包含了第二个指针的地址，第二个指针指向包含实际值的位置。



一个指向指针的指针变量必须如下声明，即在变量名前放置两个星号。例如，下面声明了一个指向 int 类型指针的指针：

```
int **var;
```

当一个目标值被一个指针间接指向到另一个指针时，访问这个值需要使用两个星号运算符，如下面实例所示：




```
#include <stdio.h>
int main () {
    int  V;
```

```
    int  *Pt1;
    int  **Pt2;
    V = 100;
    /* 获取 V 的地址 */
    Pt1 = &V;
    /* 使用运算符 & 获取 Pt1 的地址 */
    Pt2 = &Pt1;
    /* 使用 pptr 获取值 */
    printf("var = %d\n", V );
    printf("Pt1 = %p\n", Pt1 );
    printf("*Pt1 = %d\n", *Pt1 );
    printf("Pt2 = %p\n", Pt2 );
    printf("**Pt2 = %d\n", **Pt2);
return 0; }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
var = 100
Pt1 = 0x7ffee2d5e8d8
*Pt1 = 100
Pt2 = 0x7ffee2d5e8d0
**Pt2 = 100
```

9、指向函数的指针

函数指针是指向函数的指针变量。

通常我们说的指针变量是指向一个整型、字符型或数组等变量，而函数指针是指向函数。

函数指针可以像一般函数一样，用于调用函数、传递参数。

函数指针变量的声明：

```
typedef int (*fun_ptr)(int,int); // 声明一个指向同样参数、返回值的函数指针类型
```

以下实例声明了函数指针变量 p，指向函数 max：

```
#include <stdio.h>
int max(int x, int y) { return x > y ? x : y; }
int main(void) {
  /* p 是函数指针 */
  int (* p)(int, int) = & max;
  // &可以省略
  int a, b, c, d;
  printf("请输入三个数字:");
  scanf("%d %d %d", & a, & b, & c);
  /* 与直接调用函数等价，d = max(max(a, b), c) */
  d = p(p(a, b), c);
  printf("最大的数字是: %d\n", d);
return 0; }
```

编译执行，输出结果如下：

```
请输入三个数字:1 2 3
最大的数字是: 3
```

回调函数

函数指针作为某个函数的参数

函数指针变量可以作为某个函数的参数来使用的，回调函数就是一个通过函数指针调用的函数。

简单讲：回调函数是由别人的函数执行时调用你实现的函数。

> 你到一个商店买东西，刚好你要的东西没有货，于是你在店员那里留下了你的电话，过了几天店里有货了，店员就打了你的电话，然后你接到电话后就到店里去取了货。在这个例子里，你的电话号码就叫回调函数，你把电话留给店员就叫登记回调函数，店里后来有货了叫做触发了回调关联的事件，店员给你打电话叫做调用回调函数，你到店里去取货叫做响应回调事件。

实例中 populate_array() 函数定义了三个参数，其中第三个参数是函数的指针，通过该函数来设置数组的值。

实例中我们定义了回调函数 getNextRandomValue()，它返回一个随机值，它作为一个函数指针传递给 populate_array() 函数。

populate_array() 将调用 10 次回调函数，并将回调函数的返回值赋值给数组。

```c
#include <stdlib.h>
#include <stdio.h>
void populate_array(int *array, size_t arraySize, int (*getNextValue)(void
)) {
    for (size_t i=0; i<arraySize; i++) array[i] = getNextValue();
}
// 获取随机值
int getNextRandomValue(void) { return rand(); }
int main(void) {
int myarray[10];
/* getNextRandomValue 不能加括号，否则无法编译，因为加上括号之后相当于传入此参数时传入了 int ，而不是函数指针*/
populate_array(myarray, 10, getNextRandomValue);
for(int i = 0; i < 10; i++) {
    printf("%d ", myarray[i]); } printf("\n"); return 0;
}
```

编译执行，输出结果如下：

```
16807 282475249 1622650073 984943658 1144108930 470211272 101027544
1457850878 1458777923 2007237709
```

三、预处理和多文件项目

（一）预处理

1、预处理器

C 语言编译器在编译程序之前，会先使用预处理器（preprocessor）处理代码。

预处理器首先会清理代码，进行删除注释、多行语句合成一个逻辑行等工作。然后，执行 `#` 开头的预处理指令。本章介绍 C 语言的预处理指令。

预处理指令可以出现在程序的任何地方，但是习惯上，往往放在代码的开头部分。

每个预处理指令都以 `#` 开头，放在一行的行首，指令前面可以有空白字符（比如空格或制表符）。`#` 和指令的其余部分之间也可以有空格，但是为了兼容老的编译器，一般不留空格。

所有预处理指令都是一行的，除非在行尾使用反斜杠，将其折行。指令结尾处不需要分号。

2、宏定义

1）#define

`#define` 是最常见的预处理指令，用来将指定的词替换成另一个词。它的参数分成两个部分，第一个参数就是要被替换的部分，其余参数是替换后的内容。每条替换规则，称为一个宏（macro）。

```
#define MAX 100
```

上面示例中，`#define` 指定将源码里面的 `MAX`，全部替换成 `100`。`MAX` 就称为一个宏。

宏的名称不允许有空格，而且必须遵守 C 语言的变量命名规则，只能使用字母、数字与下划线（`_`），且首字符不能是数字。

宏是原样替换，指定什么内容，就一模一样替换成什么内容。

```
#define HELLO "Hello, world"

// 相当于 printf("%s", "Hello, world");
printf("%s", HELLO);
```

上面示例中，宏 `HELLO` 会被原样替换成 `"Hello, world"`。

`#define` 指令可以出现在源码文件的任何地方，从指令出现的地方到文件末尾都有效。习惯上，会将 `#define` 放在源码文件的头部。它的主要好处是，会使得程序的可读性更好，也更容易修改。

`#define` 指令从 `#` 开始，一直到换行符为止。如果整条指令过长，可以在折行处使用反斜杠，延续到下一行。

```
#define OW "C programming language is invented \
in 1970s."
```

上面示例中，第一行结尾的反斜杠将 `#define` 指令拆成两行。

`#define` 允许多重替换，即一个宏可以包含另一个宏。

```
#define TWO 2
#define FOUR TWO*TWO
```

上面示例中， `FOUR` 会被替换成 `2*2` 。

注意，如果宏出现在字符串里面（即出现在双引号中），或者是其他标识符的一部分，就会失效，并不会发生替换。

```
#define TWO 2

// 输出 TWO
printf("TWO\n");

// 输出 22
const TWOs = 22;
printf("%d\n", TWOs);
```

上面示例中，双引号里面的 `TWO` ，以及标识符 `TWOs` ，都不会被替换。

同名的宏可以重复定义，只要定义是相同的，就没有问题。如果定义不同，就会报错。

```
// 正确
#define FOO hello
#define FOO hello

// 报错
#define BAR hello
#define BAR world
```

上面示例中，宏 `FOO` 没有变化，所以可以重复定义，宏 `BAR` 发生了变化，就报错了。

2）带参数的宏

宏的强大之处在于，它的名称后面可以使用括号，指定接受一个或多个参数。

```
#define SQUARE(X) X*X
```

上面示例中，宏 `SQUARE` 可以接受一个参数 `X` ，替换成 `X*X` 。

注意，宏的名称与左边圆括号之间，不能有空格。

这个宏的用法如下。

```
// 替换成 z = 2*2;
z = SQUARE(2);
```

这种写法很像函数，但又不是函数，而是完全原样的替换，会跟函数有不一样的行为。

```
#define SQUARE(X) X*X

// 输出19
printf("%d\n", SQUARE(3 + 4));
```

上面示例中，`SQUARE(3 + 4)` 如果是函数，输出的应该是49（`7*7`）；宏是原样替换，所以替换成 `3 + 4*3 + 4`，最后输出19。

可以看到，原样替换可能导致意料之外的行为。解决办法就是在定义宏的时候，尽量多使用圆括号，这样可以避免很多意外。

```
#define SQUARE(X) ((X) * (X))
```

上面示例中，`SQUARE(X)` 替换后的形式，有两层圆括号，就可以避免很多错误的发生。

宏的参数也可以是空的。

```
#define getchar() getc(stdin)
```

上面示例中，宏 `getchar()` 的参数就是空的。这种情况其实可以省略圆括号，但是加上了，会让它看上去更像函数。

一般来说，带参数的宏都是一行的。下面是两个例子。

```
#define MAX(x, y) ((x)>(y)?(x):(y))
#define IS_EVEN(n) ((n)%2==0)
```

如果宏的长度过长，可以使用反斜杠（`\`）折行，将宏写成多行。

```
#define PRINT_NUMS_TO_PRODUCT(a, b) { \
  int product = (a) * (b); \
  for (int i = 0; i < product; i++) { \
    printf("%d\n", i); \
  } \
}
```

上面示例中，替换文本放在大括号里面，这是为了创造一个块作用域，避免宏内部的变量污染外部。

带参数的宏也可以嵌套，一个宏里面包含另一个宏。

```
#define QUADP(a, b, c)
((-(b) + sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
#define QUADM(a, b, c)
((-(b) - sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
#define QUAD(a, b, c) QUADP(a, b, c), QUADM(a, b, c)
```

上面示例是一元二次方程组求解的宏，由于存在正负两个解，所以宏 `QUAD` 先替换成另外两个宏 `QUADP` 和 `QUADM`，后者再各自替换成一个解。

那么，什么时候使用带参数的宏，什么时候使用函数呢?

一般来说，应该首先使用函数，它的功能更强、更容易理解。宏有时候会产生意想不到的替换结果，而且往往只能写成一行，除非对换行符进行转义，但是可读性就变得很差。

宏的优点是相对简单，本质上是字符串替换，不涉及数据类型，不像函数必须定义数据类型。而且，宏将每一处都替换成实际的代码，省掉了函数调用的开销，所以性能会好一些。

3）#、##运算符

由于宏不涉及数据类型，所以替换以后可能为各种类型的值。如果希望替换后的值为字符串，可以在替换文本的参数前面加上 `#` 。

```
#define STR(x) #x

// 等同于 printf("%s\n", "3.14159");
printf("%s\n", STR(3.14159));
```

上面示例中， `STR(3.14159)` 会被替换成 `3.14159` 。如果 `x` 前面没有 `#` ，这会被解释成一个浮点数，有了 `#` 以后，就会被转换成字符串。

下面是另一个例子。

```
#define XNAME(n) "x"#n

// 输出 x4
printf("%s\n", XNAME(4));
```

上面示例中， `#n` 指定参数输出为字符串，再跟前面的字符串结合，最终输出为 `"x4"` 。如果不加 `#` ，这里实现起来就很麻烦了。

如果替换后的文本里面，参数需要跟其他标识符连在一起，组成一个新的标识符，可以使用 `##` 运算符。它起到粘合作用，将参数“嵌入”一个标识符之中。

```
#define MK_ID(n) i##n
```

上面示例中， `n` 是宏 `MK_ID` 的参数，这个参数需要跟标识符 `i` 粘合在一起，这时 `i` 和 `n` 之间就要使用 `##` 运算符。下面是这个宏的用法示例。

```
int MK_ID(1), MK_ID(2), MK_ID(3);
// 替换成
int i1, i2, i3;
```

上面示例中，替换后的文本 `i1` 、 `i2` 、 `i3` 是三个标识符，参数 `n` 是标识符的一部分。从这个例子可以看到， `##` 运算符的一个主要用途是批量生成变量名和标识符。

4）不定参数的宏

宏的参数还可以是不定数量的（即不确定有多少个参数）， `...` 表示剩余的参数。

```
#define X(a, b, ...) (10*(a) + 20*(b)), __VA_ARGS__
```

上面示例中， `X(a, b, ...)` 表示 `X()` 至少有两个参数，多余的参数使用 `...` 表示。在替换文本中， `__VA_ARGS__` 代表多余的参数（每个参数之间使用逗号分隔）。下面是用法示例。

```
X(5, 4, 3.14, "Hi!", 12)
// 替换成
(10*(5) + 20*(4)), 3.14, "Hi!", 12
```

注意，`...`只能替代宏的尾部参数，不能写成下面这样。

```
// 报错
#define WRONG(X, ..., Y) #X #__VA_ARGS__ #Y
```

上面示例中，`...`替代中间部分的参数，这是不允许的，会报错。

`__VA_ARGS__`前面加上一个`#`号，可以让输出变成一个字符串。

```
#define X(...) #__VA_ARGS__

printf("%s\n", X(1,2,3));  // Prints "1, 2, 3"
```

5）预定义宏

C 语言提供一些预定义的宏，可以直接使用。

- `__DATE__`：编译日期，格式为“Mmm dd yyyy”的字符串（比如 Nov 23 2021）。
- `__TIME__`：编译时间，格式为“hh:mm:ss”。
- `__FILE__`：当前文件名。
- `__LINE__`：当前行号。
- `__func__`：当前正在执行的函数名。该预定义宏必须在函数作用域使用。
- `__STDC__`：如果被设为1，表示当前编译器遵循 C 标准。
- `__STDC_HOSTED__`：如果被设为1，表示当前编译器可以提供完整的标准库；否则被设为0（嵌入式系统的标准库常常是不完整的）。
- `__STDC_VERSION__`：编译所使用的 C 语言版本，是一个格式为`yyyymmL`的长整数，C99 版本为“199901L”，C11 版本为“201112L”，C17 版本为“201710L”。

下面示例打印这些预定义宏的值。

```
#include <stdio.h>

int main(void) {
  printf("This function: %s\n", __func__);
  printf("This file: %s\n", __FILE__);
  printf("This line: %d\n", __LINE__);
  printf("Compiled on: %s %s\n", __DATE__, __TIME__);
  printf("C Version: %ld\n", __STDC_VERSION__);
}

/* 输出如下

This function: main
This file: test.c
This line: 7
Compiled on: Mar 29 2021 19:19:37
C Version: 201710

*/
```

3、文件包含

`#include` 指令用于编译时将其他源码文件，加载进入当前文件。它有两种形式。

```
// 形式一
#include <foo.h> // 加载系统提供的文件

// 形式二
#include "foo.h" // 加载用户提供的文件
```

形式一，文件名写在尖括号里面，表示该文件是系统提供的，通常是标准库的库文件，不需要写路径。因为编译器会到系统指定的安装目录里面，去寻找这些文件。

形式二，文件名写在双引号里面，表示该文件由用户提供，具体的路径取决于编译器的设置，可能是当前目录，也可能是项目的工作目录。如果所要包含的文件在其他位置，就需要指定路径，下面是一个例子。

```
#include "/usr/local/lib/foo.h"
```

GCC 编译器的 `-I` 参数，也可以用来指定 `include` 命令中用户文件的加载路径。

```
$ gcc -Iinclude/ -o code code.c
```

上面命令中，`-Iinclude/` 指定从当前目录的 `include` 子目录里面，加载用户自己的文件。

`#include` 最常见的用途，就是用来加载包含函数原型的头文件（后缀名为 `.h`）（多文件项目）

4、条件编译

1）#defined运算符

`defined` 是一个预处理运算符，如果它的参数是一个定义过的宏，就会返回1，否则返回0。

使用这种语法，可以完成多重判断。

```
#if defined FOO
  x = 2;
#elif defined BAR
  x = 3;
#endif
```

这个运算符的一个应用，就是对于不同架构的系统，加载不同的头文件。

```
#if defined IBMPC
  #include "ibmpc.h"
#elif defined MAC
  #include "mac.h"
#else
  #include "general.h"
#endif
```

上面示例中，不同架构的系统需要定义对应的宏。代码根据不同的宏，加载对应的头文件。

2）#if...#endif

`#if...#endif` 指令用于预处理器的条件判断，满足条件时，内部的行会被编译，否则就被编译器忽略。

```
#if 0
  const double pi = 3.1415; // 不会执行
#endif
```

上面示例中，`#if` 后面的 `0` ，表示判断条件不成立。所以，内部的变量定义语句会被编译器忽略。`#if 0` 这种写法常用来当作注释使用，不需要的代码就放在 `#if 0` 里面。

`#if` 后面的判断条件，通常是一个表达式。如果表达式的值不等于 `0` ，就表示判断条件为真，编译内部的语句；如果表达式的值等于0，表示判断条件为伪，则忽略内部的语句。

`#if...#endif` 之间还可以加入 `#else` 指令，用于指定判断条件不成立时，需要编译的语句。

```
#define FOO 1

#if FOO
  printf("defined\n");
#else
  printf("not defined\n");
#endif
```

上面示例中，宏 `FOO` 如果定义过，会被替换成 `1` ，从而输出 `defined` ，否则输出 `not defined` 。

如果有多个判断条件，还可以加入 `#elif` 命令。

```
#if HAPPY_FACTOR == 0
  printf("I'm not happy!\n");
#elif HAPPY_FACTOR == 1
  printf("I'm just regular\n");
#else
  printf("I'm extra happy!\n");
#endif
```

上面示例中，通过 `#elif` 指定了第二重判断。注意， `#elif` 的位置必须在 `#else` 之前。如果多个判断条件皆不满足，则执行 `#else` 的部分。

没有定义过的宏，等同于 `0` 。因此如果 `UNDEFINED` 是一个没有定义过的宏，那么 `#if UNDEFINED` 为伪，而 `#if !UNDEFINED` 为真。

`#if` 的常见应用就是打开（或关闭）调试模式。

```
#define DEBUG 1
```

```
#if DEBUG
printf("value of i : %d\n", i);
printf("value of j : %d\n", j);
#endif
```

上面示例中，通过将 `DEBUG` 设为 `1` ，就打开了调试模式，可以输出调试信息。

GCC 的 `-D` 参数可以在编译时指定宏的值，因此可以很方便地打开调试开关。

```
$ gcc -DDEBUG=1 foo.c
```

上面示例中， `-D` 参数指定宏 `DEBUG` 为 `1` ，相当于在代码中指定 `#define DEBUG 1` 。

3）#ifdef...#endif

`#ifdef...#endif` 指令用于判断某个宏是否定义过。

有时源码文件可能会重复加载某个库，为了避免这种情况，可以在库文件里使用 `#define` 定义一个空的宏。通过这个宏，判断库文件是否被加载了。

```
#define EXTRA_HAPPY
```

上面示例中， `EXTRA_HAPPY` 就是一个空的宏。

然后，源码文件使用 `#ifdef...#endif` 检查这个宏是否定义过。

```
#ifdef EXTRA_HAPPY
  printf("I'm extra happy!\n");
#endif
```

上面示例中， `#ifdef` 检查宏 `EXTRA_HAPPY` 是否定义过。如果已经存在，表示加载过库文件，就会打印一行提示。

`#ifdef` 可以与 `#else` 指令配合使用。

```
#ifdef EXTRA_HAPPY
  printf("I'm extra happy!\n");
#else
  printf("I'm just regular\n");
#endif
```

上面示例中，如果宏 `EXTRA_HAPPY` 没有定义过，就会执行 `#else` 的部分。

`#ifdef...#else...#endif` 可以用来实现条件加载。

```
#ifdef MAVIS
  #include "foo.h"
  #define STABLES 1
#else
  #include "bar.h"
  #define STABLES 2
#endif
```

上面示例中，通过判断宏 `MAVIS` 是否定义过，实现加载不同的头文件。

4）#ifndef...#endif

`#ifndef...#endif` 指令跟 `#ifdef...#endif` 正好相反。它用来判断，如果某个宏没有被定义过，则执行指定的操作。

```
#ifdef EXTRA_HAPPY
  printf("I'm extra happy!\n");
#endif

#ifndef EXTRA_HAPPY
  printf("I'm just regular\n");
#endif
```

上面示例中，针对宏 `EXTRA_HAPPY` 是否被定义过， `#ifdef` 和 `#ifndef` 分别指定了两种情况各自需要编译的代码。

`#ifndef` 常用于防止重复加载。举例来说，为了防止头文件 `myheader.h` 被重复加载，可以把它放在 `#ifndef...#endif` 里面加载。

```
#ifndef MYHEADER_H
  #define MYHEADER_H
  #include "myheader.h"
#endif
```

上面示例中，宏 `MYHEADER_H` 对应文件名 `myheader.h` 的大写。只要 `#ifndef` 发现这个宏没有被定义过，就说明该头文件没有加载过，从而加载内部的代码，并会定义宏 `MYHEADER_H` ，防止被再次加载。

`#ifndef` 等同于 `#if !defined` 。

```
#ifndef FOO
// 等同于
#if !defined FOO
```

5、其他指令

1）#undef

`#undef` 指令用来取消已经使用 `#define` 定义的宏。

```
#define LIMIT 400
#undef LIMIT
```

上面示例的 `undef` 指令取消已经定义的宏 `LIMIT` ，后面就可以重新用 LIMIT 定义一个宏。

有时候想重新定义一个宏，但不确定是否以前定义过，就可以先用 `#undef` 取消，然后再定义。因为同名的宏如果两次定义不一样，会报错，而 `#undef` 的参数如果是不存在的宏，并不会报错。

GCC 的 `-U` 选项可以在命令行取消宏的定义，相当于 `#undef` 。

```
$ gcc -ULIMIT foo.c
```

上面示例中的 `-U` 参数，取消了宏 `LIMIT` ，相当于源文件里面的 `#undef LIMIT` 。

2）#line

`#line` 指令用于覆盖预定义宏 `__LINE__` ，将其改为自定义的行号。后面的行将从 `__LINE__` 的新值开始计数。

```
// 将下一行的行号重置为 300
#line 300
```

上面示例中，紧跟在 `#line 300` 后面一行的行号，将被改成300，其后的行会在300的基础上递增编号。

`#line` 还可以改掉预定义宏 `__FILE__` ，将其改为自定义的文件名。

```
#line 300 "newfilename"
```

上面示例中，下一行的行号重置为 `300` ，文件名重置为 `newfilename` 。

3）#error

`#error` 指令用于让预处理器抛出一个错误，终止编译。

```
#if __STDC_VERSION__ != 201112L
  #error Not C11
#endif
```

上面示例指定，如果编译器不使用 C11 标准，就中止编译。GCC 编译器会像下面这样报错。

```
$ gcc -std=c99 newish.c
newish.c:14:2: error: #error Not C11
```

上面示例中，GCC 使用 C99 标准编译，就报错了。

```
#if INT_MAX < 100000
  #error int type is too small
#endif
```

上面示例中，编译器一旦发现 `INT` 类型的最大值小于 `100,000` ，就会停止编译。

`#error` 指令也可以用在 `#if...#elif...#else` 的部分。

```
#if defined WIN32
  // ...
#elif defined MAC_OS
  // ...
#elif defined LINUX
  // ...
#else
  #error NOT support the operating system
#endif
```

4）#pragma

`#pragma` 指令用来修改编译器属性。

```
// 使用 C99 标准
#pragma c9x on
```

上面示例让编译器以 C99 标准进行编译。

（二）多文件

1）多文件项目

一个软件项目往往包含多个源码文件，编译时需要将这些文件一起编译，生成一个可执行文件。

假定一个项目有两个源码文件 `foo.c` 和 `bar.c`，其中 `foo.c` 是主文件，`bar.c` 是库文件。所谓“主文件”，就是包含了 `main()` 函数的项目入口文件，里面会引用库文件定义的各种函数。

```
// File foo.c
#include <stdio.h>

int main(void) {
  printf("%d\n", add(2, 3));  // 5!
}
```

上面代码中，主文件 `foo.c` 调用了函数 `add()`，这个函数是在库文件 `bar.c` 里面定义的。

```
// File bar.c

int add(int x, int y) {
  return x + y;
}
```

现在，将这两个文件一起编译。

```
$ gcc -o foo foo.c bar.c

# 更省事的写法
$ gcc -o foo *.c
```

上面命令中，gcc 的 `-o` 参数指定生成的二进制可执行文件的文件名，本例是 `foo`。

这个命令运行后，编译器会发出警告，原因是在编译 `foo.c` 的过程中，编译器发现一个不认识的函数 `add()`，`foo.c` 里面没有这个函数的原型或者定义。因此，最好修改一下 `foo.c`，在文件头部加入 `add()` 的原型。

```
// File foo.c
#include <stdio.h>

```

```
int add(int, int);

int main(void) {
  printf("%d\n", add(2, 3));  // 5!
}
```

现在再编译就没有警告了。

你可能马上就会想到，如果有多个文件都使用这个函数 `add()`，那么每个文件都需要加入函数原型。一旦需要修改函数 `add()`（比如改变参数的数量），就会非常麻烦，需要每个文件逐一改动。所以，通常的做法是新建一个专门的头文件 `bar.h`，放置所有在 `bar.c` 里面定义的函数的原型。

```
// File bar.h

int add(int, int);
```

然后使用 `include` 命令，在用到这个函数的源码文件里面加载这个头文件 `bar.h`。

```
// File foo.c

#include <stdio.h>
#include "bar.h"

int main(void) {
  printf("%d\n", add(2, 3));  // 5!
}
```

上面代码中，`#include "bar.h"` 表示加入头文件 `bar.h`。这个文件没有放在尖括号里面，表示它是用户提供的；它没有写路径，就表示与当前源码文件在同一个目录。

然后，最好在 `bar.c` 里面也加载这个头文件，这样可以让编译器验证，函数原型与函数定义是否一致。

```
// File bar.c
#include "bar.h"

int add(int a, int b) {
  return a + b;
}
```

现在重新编译，就可以顺利得到二进制可执行文件。

```
$ gcc -o foo foo.c bar.c
```

2）重复加载

头文件里面还可以加载其他头文件，因此有可能产生重复加载。比如，`a.h` 和 `b.h` 都加载了 `c.h`，然后 `foo.c` 同时加载了 `a.h` 和 `b.h`，这意味着 `foo.c` 会编译两次 `c.h`。

最好避免这种重复加载，虽然多次定义同一个函数原型并不会报错，但是有些语句重复使用会报错，比如多次重复定义同一个 Struct 数据结构。解决重复加载的常见方法是，在头文件里面设置一个专门的宏，加载时一旦发现这个宏存在，就不再继续加载当前文件了。

```
// File bar.h
#ifndef BAR_H
  #define BAR_H
  int add(int, int);
#endif
```

上面示例中，头文件 `bar.h` 使用 `#ifndef` 和 `#endif` 设置了一个条件判断。每当加载这个头文件时，就会执行这个判断，查看有没有设置过宏 `BAR_H` 。如果设置过了，表明这个头文件已经加载过了，就不再重复加载了，反之就先设置一下这个宏，然后加载函数原型。

3）extern说明符

当前文件还可以使用其他文件定义的变量，这时要使用 `extern` 说明符，在当前文件中声明，这个变量是其他文件定义的。

```
extern int myVar;
```

上面示例中， `extern` 说明符告诉编译器，变量 `myvar` 是其他脚本文件声明的，不需要在这里为它分配内存空间。

由于不需要分配内存空间，所以 `extern` 声明数组时，不需要给出数组长度。

```
extern int a[];
```

这种共享变量的声明，可以直接写在源码文件里面，也可以放在头文件中，通过 `#include` 指令加载。

4）static说明符

正常情况下，当前文件内部的全局变量，可以被其他文件使用。有时候，不希望发生这种情况，而是希望某个变量只局限在当前文件内部使用，不要被其他文件引用。

这时可以在声明变量的时候，使用 `static` 关键字，使得该变量变成当前文件的私有变量。

```
static int foo = 3;
```

上面示例中，变量 `foo` 只能在当前文件里面使用，其他文件不能引用。

5）编译策略

多个源码文件的项目，编译时需要所有文件一起编译。哪怕只是修改了一行，也需要从头编译，非常耗费时间。

为了节省时间，通常的做法是将编译拆分成两个步骤。第一步，使用 GCC 的 `-c` 参数，将每个源码文件单独编译为对象文件（object file）。第二步，将所有对象文件链

接在一起，合并生成一个二进制可执行文件。

```
$ gcc -c foo.c # 生成 foo.o
$ gcc -c bar.c # 生成 bar.o

# 更省事的写法
$ gcc -c *.c
```

上面命令为源码文件 `foo.c` 和 `bar.c` ，分别生成对象文件 `foo.o` 和 `bar.o` 。

对象文件不是可执行文件，只是编译过程中的一个阶段性产物，文件名与源码文件相同，但是后缀名变成了 `.o` 。

得到所有的对象文件以后，再次使用 `gcc` 命令，将它们通过链接，合并生成一个可执行文件。

```
$ gcc -o foo foo.o bar.o

# 更省事的写法
$ gcc -o foo *.o
```

以后，修改了哪一个源文件，就将这个文件重新编译成对象文件，其他文件不用重新编译，可以继续使用原来的对象文件，最后再将所有对象文件重新链接一次就可以了。由于链接的耗时大大短于编译，这样做就节省了大量时间。

6）make命令

大型项目的编译，如果全部手动完成，是非常麻烦的，容易出错。一般会使用专门的自动化编译工具，比如 make。

make 是一个命令行工具，使用时会自动在当前目录下搜索配置文件 makefile（也可以写成 Makefile）。该文件定义了所有的编译规则，每个编译规则对应一个编译产物。为了得到这个编译产物，它需要知道两件事。

- 依赖项（生成该编译产物，需要用到哪些文件）
- 生成命令（生成该编译产物的命令）

比如，对象文件 `foo.o` 是一个编译产物，它的依赖项是 `foo.c` ，生成命令是 `gcc -c foo.c` 。对应的编译规则如下：

```
foo.o: foo.c
  gcc -c foo.c
```

上面示例中，编译规则由两行组成。第一行首先是编译产物，冒号后面是它的依赖项，第二行则是生成命令。

注意，第二行的缩进必须使用 Tab 键，如果使用空格键会报错。

完整的配置文件 makefile 由多个编译规则组成，可能是下面的样子。

```
foo: foo.o bar.o
  gcc -o foo foo.o bar.o

foo.o: bar.h foo.c
```

```
  gcc -c foo.c

bar.o: bar.h bar.c
  gcc -c bar.c
```

上面是 makefile 的一个示例文件。它包含三个编译规则，对应三个编译产物（`foo.o`、`bar.o` 和 `foo`），每个编译规则之间使用空行分隔。

有了 makefile，编译时，只要在 make 命令后面指定编译目标（编译产物的名字），就会自动调用对应的编译规则。

```
$ make foo.o

# or
$ make bar.o

# or
$ make foo
```

上面示例中，make 命令会根据不同的命令，生成不同的编译产物。

如果省略了编译目标，`make` 命令会执行第一条编译规则，构建相应的产物。

```
$ make
```

上面示例中，`make` 后面没有编译目标，所以会执行 makefile 的第一条编译规则，本例是 `make foo`。由于用户期望执行 `make` 后得到最终的可执行文件，所以建议总是把最终可执行文件的编译规则，放在 makefile 文件的第一条。makefile 本身对编译规则没有顺序要求。

make 命令的强大之处在于，它不是每次执行命令，都会进行编译，而是会检查是否有必要重新编译。具体方法是，通过检查每个源码文件的时间戳，确定在上次编译之后，哪些文件发生过变动。然后，重新编译那些受到影响的编译产物（即编译产物直接或间接依赖于那些发生变动的源码文件），不受影响的编译产物，就不会重新编译。

举例来说，上次编译之后，修改了 `foo.c`，没有修改 `bar.c` 和 `bar.h`。于是，重新运行 `make foo` 命令时，Make 就会发现 `bar.c` 和 `bar.h` 没有变动过，因此不用重新编译 `bar.o`，只需要重新编译 `foo.o`。有了新的 `foo.o` 以后，再跟 `bar.o` 一起，重新编译成新的可执行文件 `foo`。

Make 这样设计的最大好处，就是自动处理编译过程，只重新编译变动过的文件，因此大大节省了时间。

四、I\O、文件操作

（一）输入输出

1、缓存和字节流

严格地说，输入输出函数并不是直接与外部设备通信，而是通过缓存（buffer）进行间接通信。这个小节介绍缓存是什么。

普通文件一般都保存在磁盘上面，跟 CPU 相比，磁盘读取或写入数据是一个很慢的操作。所以，程序直接读写磁盘是不可行的，可能每执行一行命令，都必须等半天。C 语言的解决方案，就是只要打开一个文件，就在内存里面为这个文件设置一个缓存区。

程序向文件写入数据时，程序先把数据放入缓存，等到缓存满了，再把里面的数据会一次性写入磁盘文件。这时，缓存区就空了，程序再把新的数据放入缓存，重复整个过程。

程序从文件读取数据时，文件先把一部分数据放到缓存里面，然后程序从缓存获取数据，等到缓存空了，磁盘文件再把新的数据放入缓存，重复整个过程。

内存的读写速度比磁盘快得多，缓存的设计减少了读写磁盘的次数，大大提高了程序的执行效率。另外，一次性移动大块数据，要比多次移动小块数据快得多。

这种读写模式，对于程序来说，就有点像水流（stream），不是一次性读取或写入所有数据，而是一个持续不断的过程。先操作一部分数据，等到缓存吞吐完这部分数据，再操作下一部分数据。这个过程就叫做字节流操作。

由于缓存读完就空了，所以字节流读取都是只能读一次，第二次就读不到了。这跟读取文件很不一样。

C 语言的输入输出函数，凡是涉及读写文件，都是属于字节流操作。输入函数从文件获取数据，操作的是输入流；输出函数向文件写入数据，操作的是输出流。

2、printf()

`printf()` 是最常用的输出函数，用于屏幕输出，原型定义在头文件 `stdio.h`，（见C基础）

3、scanf()

1）基本用法

`scanf()` 函数用于读取用户的键盘输入。程序运行到这个语句时，会停下来，等待用户从键盘输入。用户输入数据、按下回车键后，`scanf()` 就会处理用户的输入，将其存入变量。它的原型定义在头文件 `stdio.h`。

`scanf()` 的语法跟 `printf()` 类似。

```
scanf("%d", &i);
```

它的第一个参数是一个格式字符串，里面会放置占位符（与 `printf()` 的占位符基本一致），告诉编译器如何解读用户的输入，需要提取的数据是什么类型。这是因为 C 语言的数据都是有类型的，`scanf()` 必须提前知道用户输入的数据类型，才能处理数据。它的其余参数就是存放用户输入的变量，格式字符串里面有多少个占位符，就有多少个变量。

上面示例中，`scanf()` 的第一个参数 `%d`，表示用户输入的应该是一个整数。`%d` 就是一个占位符，`%` 是占位符的标志，`d` 表示整数。第二个参数 `&i` 表示，将用户从键盘输入的整数存入变量 `i`。

注意，变量前面必须加上 `&` 运算符（指针变量除外），因为 `scanf()` 传递的不是值，而是地址，即将变量 `i` 的地址指向用户输入的值。如果这里的变量是指针变量（比如字符串变量），那就不用加 `&` 运算符。

下面是一次将键盘输入读入多个变量的例子。

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

上面示例中，格式字符串 `%d%d%f%f`，表示用户输入的前两个是整数，后两个是浮点数，比如 `1 -20 3.4 -4.0e3`。这四个值依次放入 `i`、`j`、`x`、`y` 四个变量。

`scanf()` 处理数值占位符时，会自动过滤空白字符，包括空格、制表符、换行符等。所以，用户输入的数据之间，有一个或多个空格不影响 `scanf()` 解读数据。另外，用户使用回车键，将输入分成几行，也不影响解读。

```
1
-20
3.4
-4.0e3
```

上面示例中，用户分成四行输入，得到的结果与一行输入是完全一样的。每次按下回车键以后，`scanf()` 就会开始解读，如果第一行匹配第一个占位符，那么下次按下回车键时，就会从第二个占位符开始解读。

`scanf()` 处理用户输入的原理是，用户的输入先放入缓存，等到按下回车键后，按照占位符对缓存进行解读。解读用户输入时，会从上一次解读遗留的第一个字符开始，直到读完缓存，或者遇到第一个不符合条件的字符为止。

```
int x;
float y;

// 用户输入 "    -13.45e12# 0"
scanf("%d", &x);
scanf("%f", &y);
```

上面示例中，`scanf()` 读取用户输入时，`%d` 占位符会忽略起首的空格，从 `-` 处开始获取数据，读取到 `-13` 停下来，因为后面的 `.` 不属于整数的有效字符。这就是说，占位符 `%d` 会读到 `-13`。

第二次调用 `scanf()` 时，就会从上一次停止解读的地方，继续往下读取。这一次读取的首字符是 `.`，由于对应的占位符是 `%f`，会读取到 `.45e12`，这是采用科学计数法的浮点数格式。后面的 `#` 不属于浮点数的有效字符，所以会停在这里。

由于 `scanf()` 可以连续处理多个占位符，所以上面的例子也可以写成下面这样。

```
scanf("%d%f", &x, &y);
```

`scanf()` 的返回值是一个整数，表示成功读取的变量个数。如果没有读取任何项，或者匹配失败，则返回 `0` 。如果读取到文件结尾，则返回常量 EOF。

2）占位符

`scanf()` 常用的占位符如下，与 `printf()` 的占位符基本一致。

- `%c` ： 字符。
- `%d` ： 整数。
- `%f` ： `float` 类型浮点数。
- `%lf` ： `double` 类型浮点数。
- `%Lf` ： `long double` 类型浮点数。
- `%s` ： 字符串。
- `%[]` ： 在方括号中指定一组匹配的字符（比如 `%[0-9]` ），遇到不在集合之中的字符，匹配将会停止。

上面所有占位符之中，除了 `%c` 以外，都会自动忽略起首的空白字符。 `%c` 不忽略空白字符，总是返回当前第一个字符，无论该字符是否为空格。如果要强制跳过字符前的空白字符，可以写成 `scanf(" %c", &ch)` ，即 `%c` 前加上一个空格，表示跳过零个或多个空白字符。

下面要特别说一下占位符 `%s` ，它其实不能简单地等同于字符串。它的规则是，从当前第一个非空白字符开始读起，直到遇到空白字符（即空格、换行符、制表符等）为止。因为 `%s` 不会包含空白字符，所以无法用来读取多个单词，除非多个 `%s` 一起使用。这也意味着， `scanf()` 不适合读取可能包含空格的字符串，比如书名或歌曲名。另外， `scanf()` 遇到 `%s` 占位符，会在字符串变量末尾存储一个空字符 `\0` 。

`scanf()` 将字符串读入字符数组时，不会检测字符串是否超过了数组长度。所以，储存字符串时，很可能会超过数组的边界，导致预想不到的结果。为了防止这种情况，使用 `%s` 占位符时，应该指定读入字符串的最长长度，即写成 `%[m]s` ，其中的 `[m]` 是一个整数，表示读取字符串的最大长度，后面的字符将被丢弃。

```
char name[11];
scanf("%10s", name);
```

上面示例中， `name` 是一个长度为11的字符数组， `scanf()` 的占位符 `%10s` 表示最多读取用户输入的10个字符，后面的字符将被丢弃，这样就不会有数组溢出的风险了。

3）赋值忽略符

有时，用户的输入可能不符合预定的格式。

```
scanf("%d-%d-%d", &year, &month, &day);
```

上面示例中，如果用户输入 `2020-01-01` ，就会正确解读出年、月、日。问题是用户可能输入其他格式，比如 `2020/01/01` ，这种情况下， `scanf()` 解析数据就会失败。

为了避免这种情况，`scanf()` 提供了一个赋值忽略符（assignment suppression character）`*` 。只要把 `*` 加在任何占位符的百分号后面，该占位符就不会返回值，解析后将被丢弃。

```
scanf("%d%*c%d%*c%d", &year, &month, &day);
```

上面示例中，`%*c` 就是在占位符的百分号后面，加入了赋值忽略符 `*` ，表示这个占位符没有对应的变量，解读后不必返回。

4、sscanf()

`sscanf()` 函数与 `scanf()` 很类似，不同之处是 `sscanf()` 从字符串里面，而不是从用户输入获取数据。它的原型定义在头文件 `stdio.h` 里面。

```
int sscanf(const char* s, const char* format, ...);
```

`sscanf()` 的第一个参数是一个字符串指针，用来从其中获取数据。其他参数都与 `scanf()` 相同。

`sscanf()` 主要用来处理其他输入函数读入的字符串，从其中提取数据。

```
fgets(str, sizeof(str), stdin);
sscanf(str, "%d%d", &i, &j);
```

上面示例中，`fgets()` 先从标准输入获取了一行数据（`fgets()` 的介绍详见下一章），存入字符数组 `str` 。然后，`sscanf()` 再从字符串 `str` 里面提取两个整数，放入变量 `i` 和 `j` 。

`sscanf()` 的一个好处是，它的数据来源不是流数据，所以可以反复使用，不像 `scanf()` 的数据来源是流数据，只能读取一次。

`sscanf()` 的返回值是成功赋值的变量的数量，如果提取失败，返回常量 EOF。

5、getchar()，putchar()

（1）getchar()

`getchar()` 函数返回用户从键盘输入的一个字符，使用时不带有任何参数。程序运行到这个命令就会暂停，等待用户从键盘输入，等同于使用 `scanf()` 方法读取一个字符。它的原型定义在头文件 `stdio.h` 。

```
char ch;
ch = getchar();

// 等同于
scanf("%c", &ch);
```

`getchar()` 不会忽略起首的空白字符，总是返回当前读取的第一个字符，无论是否为空格。如果读取失败，返回常量 EOF，由于 EOF 通常是 `-1` ，所以返回值的类型要设为 int，而不是 char。

由于 `getchar()` 返回读取的字符，所以可以用在循环条件之中。

```
while (getchar() != '\n')
  ;
```

上面示例中，只有读到的字符等于换行符（`\n`），才会退出循环，常用来跳过某行。`while` 循环的循环体没有任何语句，表示对该行不执行任何操作。

下面的例子是计算某一行的字符长度。

```
int len = 0;
while(getchar() != '\n')
  len++;
```

上面示例中，`getchar()` 每读取一个字符，长度变量 `len` 就会加1，直到读取到换行符为止，这时 `len` 就是该行的字符长度。

下面的例子是跳过空格字符。

```
while ((ch = getchar()) == ' ')
  ;
```

上面示例中，结束循环后，变量 `ch` 等于第一个非空格字符。

（2）putchar()

`putchar()` 函数将它的参数字符输出到屏幕，等同于使用 `printf()` 输出一个字符。它的原型定义在头文件 `stdio.h` 。

```
putchar(ch);
// 等同于
printf("%c", ch);
```

操作成功时，`putchar()` 返回输出的字符，否则返回常量 EOF。

（3）小结

由于 `getchar()` 和 `putchar()` 这两个函数的用法，要比 `scanf()` 和 `printf()` 更简单，而且通常是用宏来实现，所以要比 `scanf()` 和 `printf()` 更快。如果操作单个字符，建议优先使用这两个函数。

6、puts()

`puts()` 函数用于将参数字符串显示在屏幕（stdout）上，并且自动在字符串末尾添加换行符。它的原型定义在头文件 `stdio.h` 。

```
puts("Here are some messages:");
puts("Hello World");
```

上面示例中，`puts()` 在屏幕上输出两行内容。

写入成功时，`puts()` 返回一个非负整数，否则返回常量 EOF。

7、gets()

`gets()` 函数以前用于从 `stdin` 读取整行输入，现在已经被废除了，仍然放在这里介绍一下。

该函数读取用户的一行输入，不会跳过起始处的空白字符，直到遇到换行符为止。这个函数会丢弃换行符，将其余字符放入参数变量，并在这些字符的末尾添加一个空字符 `\0` ，使其成为一个字符串。

它经常与 `puts()` 配合使用。

```
char words[81];

puts("Enter a string, please");
gets(words);
```

上面示例使用 `puts()` 在屏幕上输出提示，然后使用 `gets()` 获取用户的输入。

由于 `gets()` 获取的字符串，可能超过字符数组变量的最大长度，有安全风险，建议不要使用，改为使用 `fgets()` 。

（二）文件操作

1、文件指针

C 语言提供了一个 FILE 数据结构，记录了操作一个文件所需要的信息。该结构定义在头文件 `stdio.h` ，所有文件操作函数都要通过这个数据结构，获取文件信息。

开始操作一个文件之前，就要定义一个指向该文件的 FILE 指针，相当于获取一块内存区域，用来保存文件信息。

```
FILE* fp;
```

上面示例定义了一个 FILE 指针 `fp` 。

下面是一个读取文件的完整示例。

```
#include <stdio.h>
int main(void) {
  FILE* fp;
  char c;
  fp = fopen("hello.txt", "r");
  if (fp == NULL) {
    return -1;
  }
  c = fgetc(fp);
  printf("%c\n", c);
  fclose(fp);
  return 0;
}
```

上面示例中，新建文件指针 `fp` 以后，依次使用了下面三个文件操作函数，分成三个步骤。其他的文件操作，大致上也是这样的步骤。

第一步，使用 `fopen()` 打开指定文件，返回一个 File 指针。如果出错，返回 NULL。

它相当于将指定文件的信息与新建的文件指针 `fp` 相关联，在 FILE 结构内部记录了这样一些信息：文件内部的当前读写位置、读写报错的记录、文件结尾指示器、缓冲区开

始位置的指针、文件标识符、一个计数器（统计拷贝进缓冲区的字节数）等等。后继的操作就可以使用这个指针（而不是文件名）来处理指定文件。

同时，它还为文件建立一个缓存区。由于存在缓存区，也可以说 `fopen()` 函数“打开一个了流”，后继的读写文件都是流模式。

第二步，使用读写函数，从文件读取数据，或者向文件写入数据。上例使用了 `fgetc()` 函数，从已经打开的文件里面，读取一个字符。

`fgetc()` 一调用，文件的数据块先拷贝到缓冲区。不同的计算机有不同的缓冲区大小，一般是512字节或是它的倍数，如4096或16384。随着计算机硬盘容量越来越大，缓冲区也越来越大。

`fgetc()` 从缓冲区读取数据，同时将文件指针内部的读写位置指示器，指向所读取字符的下一个字符。所有的文件读取函数都使用相同的缓冲区，后面再调用任何一个读取函数，都将从指示器指向的位置，即上一次读取函数停止的位置开始读取。

当读取函数发现已读完缓冲区里面的所有字符时，会请求把下一个缓冲区大小的数据块，从文件拷贝到缓冲区中。读取函数就以这种方式，读完文件的所有内容，直到文件结尾。不过，上例是只从缓存区读取一个字符。当函数在缓冲区里面，读完文件的最后一个字符时，就把 FILE 结构里面的文件结尾指示器设置为真。于是，下一次再调用读取函数时，会返回常量 EOF。EOF 是一个整数值，代表文件结尾，一般是 `-1` 。

第三步， `fclose()` 关闭文件，同时清空缓存区。

上面是文件读取的过程，文件写入也是类似的方式，先把数据写入缓冲区，当缓冲区填满后，缓存区的数据将被转移到文件中。

2、文件基本操作

- fopen()

`fopen()` 函数用来打开文件。所有文件操作的第一步，都是使用 `fopen()` 打开指定文件。这个函数的原型定义在头文件 `stdio.h` 。

```
FILE* fopen(char* filename, char* mode);
```

它接受两个参数。第一个参数是文件名(可以包含路径)，第二个参数是模式字符串，指定对文件执行的操作，比如下面的例子中， `r` 表示以读取模式打开文件。

```
fp = fopen("in.dat", "r");
```

成功打开文件以后， `fopen()` 返回一个 FILE 指针，其他函数可以用这个指针操作文件。如果无法打开文件（比如文件不存在或没有权限），会返回空指针 NULL。所以，执行 `fopen()` 以后，最好判断一下，有没有打开成功。

```
fp = fopen("hello.txt", "r");

if (fp == NULL) {
  printf("Can't open file!\n");
  exit(EXIT_FAILURE);
}
```

上面示例中，如果 `fopen()` 返回一个空指针，程序就会报错。

`fopen()` 的模式字符串有以下几种。

- `r`：读模式，只用来读取数据。如果文件不存在，返回 NULL 指针。
- `w`：写模式，只用来写入数据。如果文件存在，文件长度会被截为0，然后再写入；如果文件不存在，则创建该文件。
- `a`：写模式，只用来在文件尾部追加数据。如果文件不存在，则创建该文件。
- `r+`：读写模式。如果文件存在，指针指向文件开始处，可以在文件头部添加数据。如果文件不存在，返回 NULL 指针。
- `w+`：读写模式。如果文件存在，文件长度会被截为0，然后再写入数据。这种模式实际上读不到数据，反而会擦掉数据。如果文件不存在，则创建该文件。
- `a+`：读写模式。如果文件存在，指针指向文件结尾，可以在现有文件末尾添加内容。如果文件不存在，则创建该文件。

上一小节说过，`fopen()` 函数会为打开的文件创建一个缓冲区。读模式下，创建的是读缓存区；写模式下，创建的是写缓存区；读写模式下，会同时创建两个缓冲区。C 语言通过缓存区，以流的形式，向文件读写数据。

数据在文件里面，都是以二进制形式存储。但是，读取的时候，有不同的解读方法：以原本的二进制形式解读，叫做“二进制流”；将二进制数据转成文本，以文本形式解读，叫做“文本流”。写入操作也是如此，分成以二进制写入和以文本写入，后者会多一个文本转二进制的步骤。

`fopen()` 的模式字符串，默认是以文本流读写。如果添加 `b` 后缀（表示 binary），就会以“二进制流”进行读写。比如，`rb` 是读取二进制数据模式，`wb` 是写入二进制数据模式。

模式字符串还有一个 `x` 后缀，表示独占模式（exclusive）。如果文件已经存在，则打开文件失败；如果文件不存在，则新建文件，打开后不再允许其他程序或线程访问当前文件。比如，`wx` 表示以独占模式写入文件，如果文件已经存在，就会打开失败。

- freopen()

`freopen()` 用于新打开一个文件，直接关联到某个已经打开的文件指针。这样可以复用文件指针。它的原型定义在头文件 `stdio.h`。

```
FILE* freopen(char* filename, char* mode, FILE stream);
```

它跟 `fopen()` 相比，就是多出了第三个参数，表示要复用的文件指针。其他两个参数都一样，分别是文件名和打开模式。

```
freopen("output.txt", "w", stdout);
printf("hello");
```

上面示例将文件 `output.txt` 关联到 `stdout`，此后向 `stdout` 写入的内容，都会写入 `output.txt`。由于 `printf()` 默认就是输出到 `stdout`，所以运行上面的代码以后，文件 `output.txt` 会被写入 `hello`。

`freopen()` 的返回值是它的第三个参数（文件指针）。如果打开失败（比如文件不存在），会返回空指针 NULL。

`freopen()` 会自动关闭原先已经打开的文件，如果文件指针并没有指向已经打开的文件，则 `freopen()` 等同于 `fopen()` 。

下面是 `freopen()` 关联 `scanf()` 的例子。

```
int i, i2;

scanf("%d", &i);

freopen("someints.txt", "r", stdin);
scanf("%d", &i2);
```

上面例子中，一共调用了两次 `scanf()` ，第一次调用是从键盘读取，然后使用 `freopen()` 将 `stdin` 指针关联到某个文件，第二次调用就会从该文件读取。

某些系统允许使用 `freopen()` ，改变文件的打开模式。这时， `freopen()` 的第一个参数应该是 NULL。

```
freopen(NULL, "wb", stdout);
```

上面示例将 `stdout` 的打开模式从 `w` 改成了 `wb` 。

- fclose()

`fclose()` 用来关闭已经使用 `fopen()` 打开的文件。它的原型定义在 `stdin.h` 。

```
int fclose(FILE* stream);
```

它接受一个文件指针 `fp` 作为参数。如果成功关闭文件， `fclose()` 函数返回整数 `0` ；如果操作失败（比如磁盘已满，或者出现 I/O 错误），则返回一个特殊值 EOF（详见下一小节）。

```
if (fclose(fp) != 0)
  printf("Something wrong.");
```

不再使用的文件，都应该使用 `fclose()` 关闭，否则无法释放资源。一般来说，系统对同时打开的文件数量有限制，及时关闭文件可以避免超过这个限制。

- remove()

`remove()` 函数用于删除指定文件。它的原型定义在头文件 `stdio.h` 。

```
int remove(const char* filename);
```

它接受文件名作为参数。如果删除成功， `remove()` 返回 `0` ，否则返回非零值。

```
remove("foo.txt");
```

上面示例删除了 `foo.txt` 文件。

注意，删除文件必须是在文件关闭的状态下。如果是用 `fopen()` 打开的文件，必须先用 `fclose()` 关闭后再删除。

- rename()

`rename()` 函数用于文件改名，也用于移动文件。它的原型定义在头文件 `stdio.h` 。

```
int rename(const char* old_filename, const char* new_filename);
```

它接受两个参数，第一个参数是现在的文件名，第二个参数是新的文件名。如果改名成功， `rename()` 返回 `0` ，否则返回非零值。

```
rename("foo.txt", "bar.txt");
```

上面示例将 `foo.txt` 改名为 `bar.txt` 。

注意，改名后的文件不能与现有文件同名。另外，如果要改名的文件已经打开了，必须先关闭，然后再改名，对打开的文件进行改名会失败。

下面是移动文件的例子。

```
rename("/tmp/evidence.txt", "/home/beej/nothing.txt");
```

3、标准流

Linux 系统默认提供三个已经打开的文件，它们的文件指针如下。

- `stdin` （标准输入）：默认来源为键盘，文件指针编号为 `0` 。
- `stdout` （标准输出）：默认目的地为显示器，文件指针编号为 `1` 。
- `stderr` （标准错误）：默认目的地为显示器，文件指针编号为 `2` 。

Linux 系统的文件，不一定是数据文件，也可以是设备文件，即文件代表一个可以读或写的设备。文件指针 `stdin` 默认是把键盘看作一个文件，读取这个文件，就能获取用户的键盘输入。同理， `stdout` 和 `stderr` 默认是把显示器看作一个文件，将程序的运行结果写入这个文件，用户就能看到运行结果了。它们的区别是， `stdout` 写入的是程序的正常运行结果， `stderr` 写入的是程序的报错信息。

这三个输入和输出渠道，是 Linux 默认提供的，所以分别称为标准输入（stdin）、标准输出（stdout）和标准错误（stderr）。因为它们的实现是一样的，都是文件流，所以合称为“标准流”。

Linux 允许改变这三个文件指针（文件流）指向的文件，这称为重定向（redirection）。

如果标准输入不绑定键盘，而是绑定其他文件，可以在文件名前面加上小于号 `<` ，跟在程序名后面。这叫做“输入重定向”（input redirection）。

```
$ demo < in.dat
```

上面示例中， `demo` 程序代码里面的 `stdin` ，将指向文件 `in.dat` ，即从 `in.dat` 获取数据。

如果标准输出绑定其他文件，而不是显示器，可以在文件名前加上大于号 `>` ，跟在程序名后面。这叫做“输出重定向”（output redirection）。

```
$ demo > out.dat
```

上面示例中，`demo` 程序代码里面的 `stdout`，将指向文件 `out.dat`，即向 `out.dat` 写入数据。

输出重定向 `>` 会先擦去 `out.dat` 的所有原有的内容，然后再写入。如果希望写入的信息追加在 `out.dat` 的结尾，可以使用 `>>` 符号。

```
$ demo >> out.dat
```

上面示例中，`demo` 程序代码里面的 `stdout`，将向文件 `out.dat` 写入数据。与 `>` 不同的是，写入的开始位置是 `out.dat` 的文件结尾。

标准错误的重定向符号是 `2>`。其中的 `2` 代表文件指针的编号，即 `2>` 表示将2号文件指针的写入，重定向到 `err.txt`。2号文件指针就是标准错误 `stderr`。

```
$ demo > out.dat 2> err.txt
```

上面示例中，`demo` 程序代码里面的 `stderr`，会向文件 `err.txt` 写入报错信息。而 `stdout` 向文件 `out.dat` 写入。

输入重定向和输出重定向，也可以结合在一条命令里面。

```
$ demo < in.dat > out.dat

// or
$ demo > out.dat < in.dat
```

重定向还有另一种情况，就是将一个程序的标准输出 `stdout`，指向另一个程序的标准输入 `stdin`，这时要使用 `|` 符号。

```
$ random | sum
```

上面示例中，`random` 程序代码里面的 `stdout` 的写入，会从 `sum` 程序代码里面的 `stdin` 被读取。

4、EOF

C 语言的文件操作函数的设计是，如果遇到文件结尾，就返回一个特殊值。程序接收到这个特殊值，就知道已经到达文件结尾了。

头文件 `stdio.h` 为这个特殊值定义了一个宏 `EOF`（end of file 的缩写），它的值一般是 `-1`。这是因为从文件读取的二进制值，不管作为无符号数字解释，还是作为 ASCII 码解释，都不可能是负值，所以可以很安全地返回 `-1`，不会跟文件本身的数据相冲突。

需要注意的是，不像字符串结尾真的存储了 `\0` 这个值，`EOF` 并不存储在文件结尾，文件中并不存在这个值，完全是文件操作函数发现到达了文件结尾，而返回这个值。

5、文件编辑

1）fgetc()、getc()

`fgetc()` 和 `getc()` 用于从文件读取一个字符。它们的用法跟 `getchar()` 类似，区别是 `getchar()` 只用来从 `stdin` 读取，而这两个函数是从任意指定的文件读取。它们的原型

定义在头文件 `stdio.h` 。

```
int fgetc(FILE *stream)
int getc(FILE *stream);
```

`fgetc()` 与 `getc()` 的用法是一样的，都只有文件指针一个参数。两者的区别是，`getc()` 一般用宏来实现，而 `fgetc()` 是函数实现，所以前者的性能可能更好一些。注意，虽然这两个函数返回的是一个字符，但是它们的返回值类型却不是 `char` ，而是 `int` ，这是因为读取失败的情况下，它们会返回 EOF，这个值一般是 `-1` 。

2）fputc()、putc()

`fputc()` 和 `putc()` 用于向文件写入一个字符。它们的用法跟 `putchar()` 类似，区别是 `putchar()` 是向 `stdout` 写入，而这两个函数是向文件写入。它们的原型定义在头文件 `stdio.h` 。

```
int fputc(int char, FILE *stream);
int putc(int char, FILE *stream);
```

`fputc()` 与 `putc()` 的用法是一样，都接受两个参数，第一个参数是待写入的字符，第二个参数是文件指针。它们的区别是， `putc()` 通常是使用宏来实现，而 `fputc()` 只作为函数来实现，所以理论上， `putc()` 的性能会好一点。

写入成功时，它们返回写入的字符；写入失败时，返回 EOF。

3）fprintf()

`fprintf()` 用于向文件写入格式化字符串，用法与 `printf()` 类似。区别是 `printf()` 总是写入 `stdout` ，而 `fprintf()` 则是写入指定的文件，它的第一个参数必须是一个文件指针。它的原型定义在头文件 `stdio.h` 。

```
int fprintf(FILE* stream, const char* format, ...)
```

`fprintf()` 可以替代 `printf()` 。

```
printf("Hello, world!\n");
fprintf(stdout, "Hello, world!\n");
```

上面例子中，指定 `fprintf()` 写入 `stdout` ，结果就等同于调用 `printf()` 。

```
fprintf(fp, "Sum: %d\n", sum);
```

上面示例是向文件指针 `fp` 写入指定格式的字符串。

下面是向 `stderr` 输出错误信息的例子。

```
fprintf(stderr, "Something number.\n");
```

4）fscanf()

`fscanf()` 用于按照给定的模式，从文件中读取内容，用法跟 `scanf()` 类似。区别是 `scanf()` 总是从 `stdin` 读取数据，而 `fscanf()` 是从文件读入数据，它的原型定义在头文件 `stdio.h` ，第一个参数必须是文件指针。

```
int fscanf(FILE* stream, const char* format, ...);
```

下面是一个例子。

```
fscanf(fp, "%d%d", &i, &j);
```

上面示例中，`fscanf()` 从文件 `fp` 里面，读取两个整数，放入变量 `i` 和 `j` 。

使用 `fscanf()` 的前提是知道文件的结构，它的占位符解析规则与 `scanf()` 完全一致。

由于 `fscanf()` 可以连续读取，直到读到文件尾，或者发生错误（读取失败、匹配失败），才会停止读取，所以 `fscanf()` 通常放在循环里面。

```
while(fscanf(fp, "%s", words) == 1)
  puts(words);
```

上面示例中，`fscanf()` 依次读取文件的每个词，将它们一行打印一个，直到文件结束。

`fscanf()` 的返回值是赋值成功的变量数量，如果赋值失败会返回 EOF。

5）fgets()

`fgets()` 用于从文件读取指定长度的字符串，它名字的第一个字符是 `f` ，就代表 `file` 。它的原型定义在头文件 `stdio.h` 。

```
char* fgets(char* str, int STRLEN, File* fp);
```

它的第一个参数 `str` 是一个字符串指针，用于存放读取的内容。第二个参数 `STRLEN` 指定读取的长度，第三个参数是一个 FILE 指针，指向要读取的文件。

`fgets()` 读取 STRLEN – 1 个字符之后，或者遇到换行符与文件结尾，就会停止读取，然后在已经读取的内容末尾添加一个空字符 `\0` ，使之成为一个字符串。注意，`fgets()` 会将换行符（ `\n` ）存储进字符串。

如果 `fgets` 的第三个参数是 `stdin` ，就可以读取标准输入，等同于 `scanf()` 。

```
fgets(str, sizeof(str), stdin);
```

读取成功时，`fgets()` 的返回值是它的第一个参数，即指向字符串的指针，否则返回空指针 NULL。

`fgets()` 可以用来读取文件的每一行，

6）fputs()

`fputs()` 函数用于向文件写入字符串，和 `puts()` 函数只有一点不同，那就是它不会在字符串末尾添加换行符。这是因为 `fgets()` 保留了换行符，所以 `fputs()` 就不添加了。`fputs()` 函数通常与 `fgets()` 配对使用。

它的原型定义在 `stdio.h` 。

```
int fputs(const char* str, FILE* stream);
```

它接受两个参数，第一个参数是字符串指针，第二个参数是要写入的文件指针。如果第二个参数为 `stdout` （标准输出），就是将内容输出到计算机屏幕，等同于 `printf()` 。

```
char words[14];
```

```
puts("Enter a string, please.");
fgets(words, 14, stdin);

puts("This is your string:");
fputs(words, stdout);
```

上面示例中，先用 `fgets()` 从 `stdin` 读取用户输入，然后用 `fputs()` 输出到 `stdout` 。写入成功时， `fputs()` 返回一个非负整数，否则返回 EOF。

7）fwrite()

`fwrite()` 用来一次性写入较大的数据块，主要用途是将数组数据一次性写入文件，适合写入二进制数据。它的原型定义在 `stdio.h` 。

```
size_t fwrite(
  const void* ptr,
  size_t size,
  size_t nmemb,
  FILE* fp
);
```

它接受四个参数。

- `ptr` ： 数组指针。
- `size` ： 每个数组成员的大小，单位字节。
- `nmemb` ： 数组成员的数量。
- `fp` ： 要写入的文件指针。

注意， `fwrite()` 原型的第一个参数类型是 `void*` ，这是一个无类型指针，编译器会自动将参数指针转成 `void*` 类型。正是由于 `fwrite()` 不知道数组成员的类型，所以才需要知道每个成员的大小（第二个参数）和成员数量（第三个参数）。

`fwrite()` 函数的返回值是成功写入的数组成员的数量（注意不是字节数）。正常情况下，该返回值就是第三个参数 `nmemb` ，但如果出现写入错误，只写入了一部分成员，返回值会比 `nmemb` 小。

要将整个数组 `arr` 写入文件，可以采用下面的写法。

```
fwrite(
  arr,
  sizeof(arr[0]),
  sizeof(arr) / sizeof(arr[0]),
  fp
);
```

上面示例中， `sizeof(a[0])` 是每个数组成员占用的字节， `sizeof(a) / sizeof(a[0])` 是整个数组的成员数量。

下面的例子是将一个大小为256字节的字符串写入文件。

```
char buffer[256];

```

```
fwrite(buffer, 1, 256, fp);
```

上面示例中，数组 `buffer` 每个成员是1个字节，一共有256个成员。由于 `fwrite()` 是连续内存复制，所以写成 `fwrite(buffer, 256, 1, fp)` 也能达到目的。

`fwrite()` 没有规定一定要写入整个数组，只写入数组的一部分也是可以的。

任何类型的数据都可以看成是1字节数据组成的数组，或者是一个成员的数组，所以 `fwrite()` 实际上可以写入任何类型的数据，而不仅仅是数组。比如， `fwrite()` 可以将一个 Struct 结构写入文件保存。

```
fwrite(&s, sizeof(s), 1, fp);
```

上面示例中， `s` 是一个 Struct 结构指针，可以看成是一个成员的数组。注意，如果 `s` 的属性包含指针，存储时需要小心，因为保存指针可能没意义，还原出来的时候，并不能保证指针指向的数据还存在。

`fwrite()` 以及后面要介绍的 `fread()` ，比较适合读写二进制数据，因为它们不会对写入的数据进行解读。二进制数据可能包含空字符 `\0` ，这是 C 语言的字符串结尾标记，所以读写二进制文件，不适合使用文本读写函数（比如 `fprintf()` 等）。

下面是一个写入二进制文件的例子。

```
#include <stdio.h>

int main(void) {
  FILE* fp;
  unsigned char bytes[] = {5, 37, 0, 88, 255, 12};

  fp = fopen("output.bin", "wb");
  fwrite(bytes, sizeof(char), sizeof(bytes), fp);
  fclose(fp);
  return 0;
}
```

上面示例中，写入二进制文件时， `fopen()` 要使用 `wb` 模式打开，表示二进制写入。

`fwrite()` 可以把数据解释成单字节数组，因此它的第二个参数是 `sizeof(char)` ，第三个参数是数组的总字节数 `sizeof(bytes)` 。

上面例子写入的文件 `output.bin` ，使用十六进制编辑器打开，会是下面的内容。

```
05 25 00 58 ff 0c
```

`fwrite()` 还可以连续向一个文件写入数据。

```
struct clientData myClient = {1, 'foo bar'};

for (int i = 1; i <= 100; i++) {
  fwrite(&myClient, sizeof(struct clientData), 1, cfPtr);
}
```

上面示例中， `fwrite()` 连续将100条数据写入文件。

8）fread()

`fread()` 函数用于一次性从文件读取较大的数据块，主要用途是将文件内容读入一个数组，适合读取二进制数据。它的原型定义在头文件 `stdio.h` 。

```
size_t fread(
  void* ptr,
  size_t size,
  size_t nmemb,
  FILE* fp
);
```

它接受四个参数，与 `fwrite()` 完全相同。

- `ptr` ： 数组地址。
- `size` ： 每个数组成员的大小，单位为字节。
- `nmemb` ： 数组的成员数量。
- `fp` ： 文件指针。

要将文件内容读入数组 `arr` ，可以采用下面的写法。

```
fread(
  arr,
  sizeof(arr[0]),
  sizeof(arr) / sizeof(arr[0]),
  fp
);
```

上面示例中，数组长度（第二个参数）和每个成员的大小（第三个参数）的乘积，就是数组占用的内存空间的大小。 `fread()` 会从文件（第四个参数）里面读取相同大小的内容，然后将 `ptr` （第一个参数）指向这些内容的内存地址。

下面的例子是将文件内容读入一个10个成员的双精度浮点数数组。

```
double earnings[10];
fread(earnings, sizeof(double), 10, fp);
```

上面示例中，每个数组成员的大小是 `sizeof(double)` ，一个有10个成员，就会从文件 `fp` 读取 `sizeof(double) * 10` 大小的内容。

`fread()` 函数的返回值是成功读取的数组成员的数量。正常情况下，该返回值就是第三个参数 `nmemb` ，但如果出现读取错误或读到文件结尾，该返回值就会比 `nmemb` 小。所以，检查 `fread()` 的返回值是非常重要的。

`fread()` 和 `fwrite()` 可以配合使用。在程序终止之前，使用 `fwrite()` 将数据保存进文件，下次运行时再用 `fread()` 将数据还原进入内存。

下面是读取上一节生成的二进制文件 `output.bin` 的例子。

```
#include <stdio.h>

int main(void) {
  FILE* fp;
```

```
  unsigned char c;

  fp = fopen("output.bin", "rb");
  while (fread(&c, sizeof(char), 1, fp) > 0)
    printf("%d\n", c);
  return 0;
}
```

运行后，得到如下结果。

```
5
37
0
88
255
12
```

9）feof()

`feof()` 函数判断文件的内部指针是否指向文件结尾。它的原型定义在头文件 `stdio.h` 。

```
int feof(FILE *fp);
```

`feof()` 接受一个文件指针作为参数。如果已经到达文件结尾，会返回一个非零值（表示 true），否则返回 `0` （表示 false）。

诸如 `fgetc()` 这样的文件读取函数，如果返回 EOF，有两种可能，一种可能是已读取到文件结尾，另一种可能是出现读取错误。 `feof()` 可以用来判断到底是那一种情况。

`feof()` 为真时，可以通过 `fseek()` 、 `rewind()` 、 `fsetpos()` 函数改变文件内部读写位置的指示器，从而清除这个函数的状态。

10）fseek()

每个文件指针都有一个内部指示器（内部指针），记录当前打开的文件的读写位置（file position），即下一次读写从哪里开始。文件操作函数（比如 `getc()` 、 `fgets()` 、 `fscanf()` 和 `fread()` 等）都从这个指示器指定的位置开始按顺序读写文件。

如果希望改变这个指示器，将它移到文件的指定位置，可以使用 `fseek()` 函数。它的原型定义在头文件 `stdio.h` 。

```
int fseek(FILE* stream, long int offset, int whence);
```

`fseek()` 接受3个参数。

- `stream` ：文件指针。
- `offset` ：距离基准（第三个参数）的字节数。类型为 long int，可以为正值（向文件末尾移动）、负值（向文件开始处移动）或 0（保持不动）。
- `whence` ：位置基准，用来确定计算起点。它的值是以下三个宏（定义在 `stdio.h` ）： `SEEK_SET` （文件开始处）、 `SEEK_CUR` （内部指针的当前位置）、

`SEEK_END`（文件末尾）

请看下面的例子。

```
// 定位到文件开始处
fseek(fp, 0L, SEEK_SET);

// 定位到文件末尾
fseek(fp, 0L, SEEK_END);

// 从当前位置后移2个字节
fseek(fp, 2L, SEEK_CUR);

// 定位到文件第10个字节
fseek(fp, 10L, SEEK_SET);

// 定位到文件倒数第10个字节
fseek(fp, -10L, SEEK_END);
```

上面示例中，`fseek()` 的第二个参数为 long 类型，所以移动距离必须加上后缀 `L`，将其转为 long 类型。

下面的示例逆向输出文件的所有字节。

```
for (count = 1L; count <= size; count++) {
  fseek(fp, -count, SEEK_END);
  ch = getc(fp);
}
```

注意，`fseek()` 最好只用来操作二进制文件，不要用来读取文本文件。因为文本文件的字符有不同的编码，某个位置的准确字节位置不容易确定。

正常情况下，`fseek()` 的返回值为0。如果发生错误（如移动的距离超出文件的范围），返回值为非零值（比如 `-1`）。

11）ftell()

`ftell()` 函数返回文件内部指示器的当前位置。它的原型定义在头文件 `stdio.h` 。

```
long int ftell(FILE* stream);
```

它接受一个文件指针作为参数。返回值是一个 long 类型的整数，表示内部指示器的当前位置，即文件开始处到当前位置的字节数，`0` 表示文件开始处。如果发生错误，`ftell()` 返回 `-1L` 。

`ftell()` 可以跟 `fseek()` 配合使用，先记录内部指针的位置，一系列操作过后，再用 `fseek()` 返回原来的位置。

```
long file_pos = ftell(fp);

// 一系列文件操作之后
fseek(fp, file_pos, SEEK_SET);
```

下面的例子先将指示器定位到文件结尾，然后得到文件开始处到结尾的字节数。

```
fseek(fp, 0L, SEEK_END);
size = ftell(fp);
```

12）rewind()

`rewind()` 函数可以让文件的内部指示器回到文件开始处。它的原型定义在 `stdio.h` 。

```
void rewind(file* stream);
```

它接受一个文件指针作为参数。

`rewind(fp)` 基本等价于 `fseek(fp, 0l, seek_set)` ，唯一的区别是 `rewind()` 没有返回值，而且会清除当前文件的错误指示器。

13）fgetpos()、fsetpos()

`fseek()` 和 `ftell()` 有一个潜在的问题，那就是它们都把文件大小限制在 long int 类型能表示的范围内。这看起来相当大，但是在32位计算机上，long int 的长度为4个字节，能够表示的范围最大为 4GB。随着存储设备的容量迅猛增长，文件也越来越大，往往会超出这个范围。鉴于此，C 语言新增了两个处理大文件的新定位函数：`fgetpos()` 和 `fsetpos()` 。

它们的原型都定义在头文件 `stdio.h` 。

```
int fgetpos(FILE* stream, fpos_t* pos);
int fsetpos(FILE* stream, const fpos_t* pos);
```

`fgetpos()` 函数会将文件内部指示器的当前位置，存储在指针变量 `pos` 。该函数接受两个参数，第一个是文件指针，第二个存储指示器位置的变量。

`fsetpos()` 函数会将文件内部指示器的位置，移动到指针变量 `pos` 指定的地址。注意，变量 `pos` 必须是通过调用 `fgetpos()` 方法获得的。 `fsetpos()` 的两个参数与 `fgetpos()` 必须是一样的。

记录文件内部指示器位置的指针变量 `pos` ，类型为 `fpos_t*` （file position type 的缩写，意为文件定位类型）。它不一定是整数，也可能是一个 Struct 结构。

下面是用法示例。

```
fpos_t file_pos;
fgetpos(fp, &file_pos);

// 一系列文件操作之后
fsetpos(fp, &file_pos);
```

上面示例中，先用 `fgetpos()` 获取内部指针的位置，后面再用 `fsetpos()` 恢复指针的位置。

执行成功时， `fgetpos()` 和 `fsetpos()` 都会返回 `0` ，否则返回非零值。

14）ferror()、clearerr()

所有的文件操作函数如果执行失败，都会在文件指针里面记录错误状态。后面的操作只要读取错误指示器，就知道前面的操作出错了。

`ferror()` 函数用来返回错误指示器的状态。可以通过这个函数，判断前面的文件操作是否成功。它的原型定义在头文件 `stdio.h` 。

```
int ferror(FILE *stream);
```

它接受一个文件指针作为参数。如果前面的操作出现错误， `ferror()` 就会返回一个非零整数（表示 true），否则返回 `0` 。

`clearerr()` 函数用来重置出错指示器。它的原型定义在头文件 `stdio.h` 。

```
void clearerr(FILE* fp);
```

它接受一个文件指针作为参数，没有返回值。

下面是一个例子。

```
FILE* fp = fopen("file.txt", "w");
char c = fgetc(fp);

if (ferror(fp)) {
  printf("读取文件：file.txt 时发生错误\n");
}

clearerr(fp);
```

上面示例中， `fgetc()` 尝试读取一个以”写模式“打开的文件，读取失败就会返回 EOF。这时调用 `ferror()` 就可以知道上一步操作出错了。处理完以后，再用 `clearerr()` 清除出错状态。

文件操作函数如果正常执行， `ferror()` 和 `feof()` 都会返回零。如果执行不正常，就要判断到底是哪里出了问题。

```
if (fscanf(fp, "%d", &n) != 1) {
  if (ferror(fp)) {
    printf("io error\n");
  }
  if (feof(fp)) {
    printf("end of file\n");
  }

  clearerr(fp);

  fclose(fp);
}
```

上面示例中，当 `fscanf()` 函数报错时，通过检查 `ferror()` 和 `feof()` ，确定到底发生什么问题。这两个指示器改变状态后，会保持不变，所以要用 `clearerr()` 清除它们，`clearerr()` 可以同时清除两个指示器。

五、声明

> C 语言允许声明变量的时候，加上一些特定的说明符（specifier），为编译器提供变量行为的额外信息。它的主要作用是帮助编译器优化代码，有时会对程序行为产生影响。

```
声明指定符 声明符
```

声明指定符有以下类型：

- 存储类型：定义C程序中变量/函数的范围（可见性、作用域）和生命周期。
  - auto
  - static
  - extern
  - register
- 类型限定符：
  - const
  - volatile
  - restrict
- 类型指定符：void、char、int、float...signed和unsigned

1、const

`const` 说明符表示变量是只读的，不得被修改。

```
const double PI = 3.14159;
PI = 3; // 报错
```

上面示例里面的 `const` ，表示变量 `PI` 的值不应改变。如果改变的话，编译器会报错。

对于数组， `const` 表示数组成员不能修改。

```
const int arr[] = {1, 2, 3, 4};
arr[0] = 5; // 报错
```

上面示例中， `const` 使得数组 `arr` 的成员无法修改。

对于指针变量， `const` 有两种写法，含义是不一样的。如果 `const` 在 `*` 前面，表示指针指向的值不可修改。

```
// const 表示指向的值 *x 不能修改
int const * x
// 或者
const int * x
```

下面示例中，对 `x` 指向的值进行修改导致报错。

```
int p = 1
const int* x = &p;

(*x)++; // 报错
```

如果 `const` 在 `*` 后面，表示指针包含的地址不可修改。

```
// const 表示地址 x 不能修改
int* const x
```

下面示例中，对 `x` 进行修改导致报错。

```
int p = 1
int* const x = &p;

x++; // 报错
```

这两者可以结合起来。

```
const char* const x;
```

上面示例中，指针变量 `x` 指向一个字符串。两个 `const` 意味着， `x` 包含的内存地址以及 `x` 指向的字符串，都不能修改。

`const` 的一个用途，就是防止函数体内修改函数参数。如果某个参数在函数体内不会被修改，可以在函数声明时，对该参数添加 `const` 说明符。这样的话，使用这个函数的人看到原型里面的 `const` ，就知道调用函数前后，参数数组保持不变。

```
void find(const int* arr, int n);
```

上面示例中，函数 `find` 的参数数组 `arr` 有 `const` 说明符，就说明该数组在函数内部将保持不变。

有一种情况需要注意，如果一个指针变量指向 `const` 变量，那么该指针变量也不应该被修改。

```
const int i = 1;
int* j = &i;
*j = 2; // 报错
```

上面示例中， `j` 是一个指针变量，指向变量 `i` ，即 `j` 和 `i` 指向同一个地址。 `j` 本身没有 `const` 说明符，但是 `i` 有。这种情况下， `j` 指向的值也不能被修改。

2、static

`static` 说明符对于全局变量和局部变量有不同的含义。

（1）用于局部变量（位于块作用域内部）。

`static` 用于函数内部声明的局部变量时，表示该变量的值会在函数每次执行后得到保留，下次执行时不会进行初始化，就类似于一个只用于函数内部的全局变量。由于不必每次执行函数时，都对该变量进行初始化，这样可以提高函数的执行速度，详见《函数》一章。

（2）用于全局变量（位于块作用域外部）。

`static` 用于函数外部声明的全局变量时，表示该变量只用于当前文件，其他源码文件不可以引用该变量，即该变量不会被链接（link）。

`static` 修饰的变量，初始化时，值不能等于变量，必须是常量。

```
int n = 10;
static m = n; // 报错
```

上面示例中，变量 `m` 有 `static` 修饰，它的值如果等于变量 `n` ，就会报错，必须等于常量。

只在当前文件里面使用的函数，也可以声明为 `static` ，表明该函数只在当前文件使用，其他文件可以定义同名函数。

```
static int g(int i);
```

3、auto

`auto` 说明符表示该变量的存储，由编译器自主分配内存空间，且只存在于定义时所在的作用域，退出作用域时会自动释放。

由于只要不是 `extern` 的变量（外部变量），都是由编译器自主分配内存空间的，这属于默认行为，所以该说明符没有实际作用，一般都省略不写。

```
auto int a;
// 等同于
int a;
```

4、extern

`extern` 说明符表示，该变量在其他文件里面声明，没有必要在当前文件里面为它分配空间。通常用来表示，该变量是多个文件共享的。

```
extern int a;
```

上面代码中， `a` 是 `extern` 变量，表示该变量在其他文件里面定义和初始化，当前文件不必为它分配存储空间。

但是，变量声明时，同时进行初始化， `extern` 就会无效。

```
// extern 无效
extern int i = 0;

// 等同于
int i = 0;
```

上面代码中， `extern` 对变量初始化的声明是无效的。这是为了防止多个 `extern` 对同一个变量进行多次初始化。

函数内部使用 `extern` 声明变量，就相当于该变量是静态存储，每次执行时都要从外部获取它的值。

函数本身默认是 `extern` ，即该函数可以被外部文件共享，通常省略 `extern` 不写。如果只希望函数在当前文件可用，那就需要在函数前面加上 `static` 。

```
extern int f(int i);
```

```
// 等同于
int f(int i);
```

5、register

`register` 说明符向编译器表示，该变量是经常使用的，应该提供最快的读取速度，所以应该放进寄存器。但是，编译器可以忽略这个说明符，不一定按照这个指示行事。

```
register int a;
```

上面示例中，`register` 提示编译器，变量 `a` 会经常用到，要为它提供最快的读取速度。

`register` 只对声明在代码块内部的变量有效。

设为 `register` 的变量，不能获取它的地址。

```
register int a;
int *p = &a; // 编译器报错
```

上面示例中，`&a` 会报错，因为变量 `a` 可能放在寄存器里面，无法获取内存地址。

如果数组设为 `register`，也不能获取整个数组或任一个数组成员的地址。

```
register int a[] = {11, 22, 33, 44, 55};

int p = a;  // 报错
int a = *(a + 2); // 报错
```

历史上，CPU 内部的缓存，称为寄存器（register）。与内存相比，寄存器的访问速度快得多，所以使用它们可以提高速度。但是它们不在内存之中，所以没有内存地址，这就是为什么不能获取指向它们的指针地址。现代编译器已经有巨大的进步，会尽可能优化代码，按照自己的规则决定怎么利用好寄存器，取得最佳的执行速度，所以可能会忽视代码里面的 `register` 说明符，不保证一定会把这些变量放到寄存器。

6、volatile

`volatile` 说明符表示所声明的变量，可能会预想不到地发生变化（即其他程序可能会更改它的值），不受当前程序控制，因此编译器不要对这类变量进行优化，每次使用时都应该查询一下它的值。硬件设备的编程中，这个说明符很常用。

```
volatile int foo;
volatile int* bar;
```

`volatile` 的目的是阻止编译器对变量行为进行优化，请看下面的例子。

```
int foo = x;
// 其他语句，假设没有改变 x 的值
int bar = x;
```

上面代码中，由于变量 `foo` 和 `bar` 都等于 `x`，而且 `x` 的值也没有发生变化，所以编译器可能会把 `x` 放入缓存，直接从缓存读取值（而不是从 x 的原始内存位置读取），

然后对 `foo` 和 `bar` 进行赋值。如果 `x` 被设定为 `volatile` ，编译器就不会把它放入缓存，每次都从原始位置去取 `x` 的值，因为在两次读取之间，其他程序可能会改变 `x` 。

7、restrict

`restrict` 说明符允许编译器优化某些代码。它只能用于指针，表明该指针是访问数据的唯一方式。

```
int* restrict pt = (int*) malloc(10 * sizeof(int));
```

上面示例中， `restrict` 表示变量 `pt` 是访问 malloc 所分配内存的唯一方式。
下面例子的变量 `foo` ，就不能使用 `restrict` 修饰符。

```
int foo[10];
int* bar = foo;
```

上面示例中，变量 `foo` 指向的内存，可以用 `foo` 访问，也可以用 `bar` 访问，因此就不能将 `foo` 设为 restrict。
如果编译器知道某块内存只能用一个方式访问，可能可以更好地优化代码，因为不用担心其他地方会修改值。

`restrict` 用于函数参数时，表示参数的内存地址之间没有重叠。

```
void swap(int* restrict a, int* restrict b) {
  int t;
  t = *a;
  *a = *b;
  *b = t;
}
```

上面示例中，函数参数声明里的 `restrict` 表示，参数 `a` 和参数 `b` 的内存地址没有重叠。