C 语言笔记

2016-2-27

作者：朱林峰

目录

C 语言笔记

1 图表

1.1 C 语言关键字

auto	double	break	else	restrict
case	enum	char	extern	
const	float	continue	for	
default	goto	do	if	
int	long	register	return	
short	signed	sizeof	static	
struct	switch	typedef	union	
unsigned	void	volatile	while	

1.2 基本类型

数据类型	字节数	最小值	最大值
void	1	无定义	无定义
char	1	-128	127
short	2	-32768	32767
int	4	-2147483648	2147483647
long	8		
long long	8		
float	4		
double	8		
long double	16		

int通常代表特定机器中整数的自然长度，现在一般为32位。

short 类型通常为16位。

long 类型通常为32位。

各编译器可以根据硬件特性自主选择合适的类型长度，但要遵循以下限制：

short 与 int 类型至少为16位

long 类型至少为32位，这个根据处理器字长而定

short 长度不得长于 int 类型

int 长度不得长于 long 类型

1.3 常量后缀

常量类型	后缀表示
long	l/L
unsigned	u/U
float	f/F
long double	l/L

1.4 进制前缀

进制	前缀表示
8进制	0
16进制	0x
用 8 进制或16进制表示的常量，同样可以有 UL 这样的后缀	

1.4 转义字符

换行符	NL(LF)	\n	反斜杠	\	\\
横向制表符	HT	\t	问号	?	\?
纵向制表符	VT	\v	单引号	'	\'
回退符	BS	\b	双引号	"	\"
回车符	CR	\r	八进制数	ooo	\ooo
换页符	FF	\f	十六进制数	hh	\xhh
响铃符	BEL	\a			

1.5 格式化限定符

符号	含义	符号	含义
%d	十进制有符号整数	%p	指针的值
%u	十进制无符号整数	%e %E	指数形式
%f	浮点数	%x %X	无符号十六进制整数
%s	字符串	%o	无符号八进制整数
%c	单个字符	%g	合适的 double 类型
%ld %lu	长整型	%%	% 本身

1.6 格式化规定符

符号	含义
%m.nd	m 最小场宽，n 最大限长
%m.nf	m 最小场宽，n 精度
%m.ns	m 最小场宽，n 最大限长
%0m.nd	0 前面不足用 0 补齐
%-m.nd	- 左对齐，没有 - 默认为右对齐
%*.*d	* 用下一参数替换，参数必须为 int 型

1.7 运算符优先级

符号	计算顺序	符号	优先级
() [] -> .	从左至右	^	从左至右
! - ++ -- * (type) sizeof	从右至左	|	从左至右
* % /	从左至右	&&	从左至右
+ -	从左至右	||	从左至右
<< >>	从左至右	?:	从左至右
< <= > >=	从左至右	+ += - -= *= /= %= &= ^= |= <<== >>==	从右至左
!= ==	从左至右	,	从右至左
&	从左至右		

1.8 文件访问模式

符号	含义
“r”	打开文本文件用于读
“w"	打开文本文件用于写，并覆盖原有内容
“a"	打开或创建文本文件，并追加内容
“r+"	打开文本文件用于更新，即读和写
“w+"	创建文本文件用于更新，并覆盖原有内容
“a+"	打开或创建文本文件用于更新，追加内容

如果在上述访问模式之后再加上 b，如“rb”或“w+b”等，则表示对二进制文件进行操作。

1.9 常用标准库头文件

```
<assert.h>    <float.h>    <math.h>    <stdarg.h>    <stdlib.h>
<ctype.h>     <limits.h>   <setjmp.h>  <stddef.h>    <string.h>
<errno.h>     <locale.h>   <signal.h>  <stdio.h>     <time.h>
```

2 关键字

2.1 enum

枚举常量是另外一种类型的常量。枚举是一个常量整型值的列表。例如：

```
enum boolean { NO, YES };
```

在没有显示声明的情况下，enum 类型中第一个枚举名的值默认为 0，第二个为 1，依此类推。如果只指定了部分枚举名的值，那么未指定值的枚举名的值将依着最后一个指定值向后递增。枚举值必须是 int 型，当然可以取负值。

不同枚举中的名字必须互不相同，但同一枚举中不同的名字可以有相同的值。

2.2 const

任何变量的声明都可以用 const 限定符限定。该限定符指定的变量值不能被修改。对数组而言，const限定符指定数组所有元素的值不能被修改。const 变量位于程序的只读数据段。

```
const int value = 0;
int const value = 0;
const int values[] = {};
int const values[] = {};
const int *block = (int*)malloc(size);
int const *block = (int*)malloc(size);
//以上所有 const 声明皆表示不能改变变量指向内容的值
```

```
int *const values = (int*)malloc(size);;
//这种 const 声明表示不能改变指针的值，但指针指向内容的值是可以改变的
```

值得注意的是数组变量名本身是个常量指针，不可以改变此指针的值。

2.3 continue

continue 语句只用于循环控制，用于放弃当前循环，跳转至下一轮循环。

```
for (i = 0; i < n; i++)
{
     ...
     continue;
     ...
}
//在 for 语句中执行 continue 后立即执行 i++，然后开始下一轮循环。
```

```
while (i < n)
{
     ...
     continue;
     ...
     i++;
}
//在 while 语句中，执行 continue 后立即开始 while (i < n) 循环判断。
```

2.4 static

static 作用于全局变量和函数，使全局变量和函数只在本源文件可见。

static 作用于局部变量，使局部变量存储于进程的静态存储区。

static 声明的变量或全局变量保存在进程的静态存储区，静态存储区的数据会被初始化为 0。

2.5 register

register 关键字只能用来声明局部变量，建议将变量保存在寄存器中以加快程序执行速度，但编译器可以忽略此项。

由 register 关键字声明的变量，其地址无法访问：

```
register int a = 1;
int *b = &a;     //非法的，a 的地址不允许被访问
```

2.6 sizeof

2.6.1 sizeof 定义

sizeof 是 C 语言的一个关键词，是一个操作符，用于返回一个对象或类型所占内存的字节数。sizeof 是一个常量表达式，其值在编译时确认，返回值类型为 size_t，在头文件 stddef.h 中定义。这是一个依赖于编译系统的值(我的系统上是 unsigned long)，一般定义为：

```
typedef unsigned int size_t;
```

2.6.2 sizeof 语法

1. 一般用法

```
typedef struct tagType
{
    char c;
    int i;
}Type;

int v;
int array[5];

//sizeof 用法
sizeof v;         //int 的大小
sizeof(v);        //int 的大小
sizeof(&v);       //int 型指针的大小
/************************/
sizeof int;       //错误用法
/************************/
sizeof(int);      //int 的大小
sizeof(int*);     //int 型指针的大小
sizeof(array)     //数组占用字节的大小
sizeof(Type);     //结构体 Type 的大小
```

2. sizeof 还可以对一个表达式求值，编译器根据表达式的最终结果类型来确定大小，一般不会对表达式进行计算

```
sizeof(2) = sizeof(int);
sizeof(2+3.14) = sizeof(double);
```

3. sizeof 还可以以函数为参数

```
typedef int (*func)();

int main()
{
    func f;

    sizeof(main);         //值为 1，不知什么含义
    sizeof(main());       //main()返回值类型 int 的大小
    sizeof(&main);        //指向 main 函数指针的大小
    sizeof(func);         //函数指针的大小
    sizeof(f);            //函数指针的大小
    sizeof(*f);           //值为 1，  不知什么含义
    sizeof(f());          //函数返回值类型 int 的大小
}

//sizeof的计算发生在编译时刻，所以在 sizeof 内进行函数调用，函数是不会执行的
//如 sizeof(main()) 中的 main() 函数是不会执行的
```

4. sizeof (指针/数组)

sizeof(指针) 的值在同一系统中都是一样的，32 位系统值为 4，64 位系统值为 8，与指针所指向对象的类型无关。在数组声明的作用域范围内，sizeof(数组) 返回的是数组占用空间的大小。

```
int intArray[10];
char str1[10];
char str2[10] = “hello”;
char *str3 = “hello”;
```

```
sizeof(intArray);     //40
sizeof(str1);         //10
sizeof(str2);         //10
sizeof(str3);         //指针大小
sizeof("hello");      //6，字符串常量占用的空间

void func(char str2[])
{
    sizeof(str2);     //指针大小
}
```

5. sizeof 与 enum

枚举值是 int 型常量。

```
enum Answer {YES, NO};

sizeof(enum Answer);     //int 的大小
sizeof(YES);             //int 的大小
```

2.6.3. sizeof 与 strlen 的比较

相同点：返回值都为 size_t 类型，一般定义为 unsigned int 类型

不同点：

- sizeof 是 C 语言关键字，内建的算符，而 strlen 是函数
- sizeof 的值在编译阶段确定，是个常量，而 strlen 的值是在程序运行阶段计算出
- strlen 只能用 char* 类型的指针作为参数，而 sizeof 的参数可以为类型、函数等
- 数组做 sizeof 的参数不退化，传递给 strlen 就变为指针了
- sizeof 返回的是整个数组占用空间的大小，strlen 返回的是从指针首地址开始，以 '\0' 结束的字符串的长度，其可能小于也可能大于数组占用空间的大小。

2.6.4. 经典案例

```
double* (*a)[3][6];
//指向二维指针数组的指针
sizeof(a);      //指针的大小，a 是一个指向二维指针数组的指针，所以它只是个指针而已
sizeof(*a);     //18 * 指针的大小，*a 是一个二维指针数组
sizeof(**a);    //6 * 指针的大小，**a 是一个一维指针数组
sizeof(***a);   //指针的大小，***a 是数组的一个元素，数组的元素是指向 double 的指针
sizeof(****a); //8，****a 是一个 double 类型的变量
```

2.7 volatile

volatile 声明的变量表示其会被编译器不可预期原因的发生改变，如操作系统、多线程访问、硬件等，因此：

1. 编译器不对 volatile 声明的变量做优化

在这里例子中，代码将 foo 的值设置为 0。然后开始不断地轮询它的值直到它变成 255：

```
static int foo;
void bar(void) {
    foo = 0;
    while (foo != 255);
}
```

一个执行优化的编译器会提示没有代码能修改 foo 的值，并假设它永远都只会是0.因此编译器将用类似下列的无限循环替换函数体：

```
void bar_optimized(void) {
    foo = 0;
    while (true);
}
```

但是，foo 可能指向一个随时都能被计算机系统其他部分修改的地址，例如一个连接到中央处理器的设备的硬件暂存器，上面的代码永远检测不到这样的修改。如果不使用 volatile 关键字，编译器将假设当前程序是系统中唯一能改变这个值部分（这是到目前为止最广泛的一种情况。 为了阻止编译器像上面那样优化代码，需要使用 volatile 关键字：

```
static volatile int foo;
void bar (void) {
    foo = 0;
    while (foo != 255);
}
```

这样修改以后循环条件就不会被优化掉，当值改变的时候系统将会检测到。

2. volatile 变量存储于内存，不会加载进寄存器，这样每次对 volatile 的访问必须从内存读取，若 volatile 变量发生了改变，程序会立马感知得到。

```
int gi_a = 0;
int a = gi_a;
int b = gi_a;
```

赋值期间程序察觉不到 gi_a 的值会发生变化，因此会对 gi_a 的值只读取一遍，放在寄存器内，在分别给 a 和 b 赋值。但可能 a 赋值后 gi_a 的值意外改变了，变为 gi_a = 1，那么在这段代码里面就无法体现这种改变，

加上 volatile 限定，则 gi_a 变量只会在内存，每次获取 gi_a 的值都必须从内存读取，当 gi_a 的值被意外改变，程序可以立马感知到。

多线程环境中，变量声明为 volatile 可以实现轻量的同步。一个线程对变量的更改能立即同步到其他线程。但需要注意，volatile 变量的优点在于同步性好，但对 volatile 变量的操作并非是原子的，因此并不能避免竞争。

2.8 restrict

restrict 是 c99 标准引入的一种类型限定符（Type Qualifiers），它只可以用于限定和约束指针，并表明指针是访问一个数据对象的唯一且初始的方式。即它告诉编译器，所有修改该指针所指向内存中内容的操作都必须通过该指针来修改，而不能通过其它途径(其它变量或指针)来修改；这样做的好处是,能帮助编译器进行更好的优化代码，生成更有效率的汇编代码。如 int *restrict ptr, ptr 指向的内存单元只能被 ptr 访问到，任何同样指向这个内存单元的其他指针都是未定义的，直白点就是无效指针。restrict 的出现是因为 C 语言本身固有的缺陷，C 程序员应当主动地规避这个缺陷，而编译器也会很配合地优化你的代码。

3 语法

3.1 赋值运算符

```
x *= y + 1
x = x * (y + 1)
//二者等价
```

3.2 初始化

在不进行显示初始化的情况下

外部变量和静态变量都初始化为 0

局部变量和寄存器变量初始化为任意值

3.3 指针

```
sizeof(void) = 1
sizeof(void*) = 4 或 8
```

指针大小根据机器字长而定，处理器是 32 位的，指针长度就是 4， 处理器是 64 位的，指针长度就是 8。

```
void test1(char a[])
{
    a++;
}
```

```
void test2(char *a)
{
    a++;
}
//两者传参是一样的，没有区别
```

3.4 结构体

3.4.1. 结构体定义、声明、初始化

```
struct tagObj
{
    char *name;
    int count;
}s1, s2;            //定义结构体类型 tagObj，并声明 s1, s2两个结构体变量

struct tagObj
{
    char *name;
    int count;
};                      //定义结构体类型 tagObj
struct tagObj s1, s2;   //声明 s1, s2两个结构体变量

struct tagObj s1;
s1.name = “myname”;     //结构体通过成员赋值初始化
s1.count = 0;

struct tagObj s2 = {“myname”, 0};     //结构体通过初值表初始化

typedef struct tagObj
{
    char *name;
    int count;
}Obj;    //定义结构体类型 Obj

Obj s1;  //使用结构体类型 Obj 声明结构体变量
```

3.4.2. 结构体用于函数传参

结构体直接用于函数传参或者函数返回结构体，都是按值传递，即会临时复制一个结构体用于传参或返回。这样的结构体都是存储于堆之中，是临时的。同时，直接用结构体传参，因为大量的复制会导致程序开销大，这时传递结构体指针更为合适。

3.4.3. 结构体对齐原则

- 每个成员变量都需要对齐，为了对齐，可以跳过前面成员留下的空隙
- 整个结构体的大小要能整除最大成员的大小，不能整除就在结构体末填充空隙。
- 结构体嵌套无影响，全部按照基本类型清算

基本类型中，最长的基本类型为16字节，故 malloc 申请内存时，内存首地址总是以16字节对齐，这样就可以保证内存中的数据总能够对齐。

```
struct a
{
    char c;
};//size 1

struct b
{
    char c;
    short s;
};//size 4
```

```
struct c
{
    short s;
    char c;
};//size 4

struct d
{
    short s;
    int i;
};//size 8

struct e
{
    int i;
    short s;
};//size 8

struct f
{
    int i;
    long l;
};//size 16

struct g
{
     long l;
     int i;
};//size 16

struct h
{
    char c;
    short s;
    int i;
    long l;
};//size 16

struct i
{
    long l;
    int i;
    short s;
    char c;
};//size 16

struct j
{
    char c;
    struct i i1;
};//size 24

struct k
{
    struct i i1;
    char c;
};//size 24
```

3.5 位字段

3.5.1. 语法

```
struct bits
{
    char b1 : 1;
    unsigned char b2 : 1;
    int b3 : 1;
    unsigned int b4 : 1;
    long b5 : 1;
    unsigned long b6 : 1;

    float b7 : 1;    //错误
};
//位段成员类型必须是整型，不能为浮点型

struct bits
{
    char b1 : 9;    //错误
};
//位段位数不能超过其指定类型的长度，这里 9bit 比 char 类型的 8bit 要长，故非法

struct bits
{
    unsigned int a : 1;
    unsigned int b : 2;
    unsigned int c : 0;    //错误
    unsigned int d : 1;
};
//位段长度不能为零

struct bits
{
    unsigned int a : 1;    //在一个 int 里面
    unsigned int b : 2;    //在一个 int 里面
    unsigned int   : 0;    //正确
    unsigned int c : 1;    //在另一个 int 里面
};
//匿名位段的长度可以为零，表示后面的位段从另一个存储单元开始

struct bits
{
    char a : 5;
    char   : 2;    //匿名位段
    char c : 2;
};
//匿名位段表示该段内的 bit 不可用

struct bits
{
    char a : 5;    //在一个 char 里面
    char b : 2;    //在一个 char 里面
    char c : 2;    //在另一个 char 里面
};
//一个位段必须在同一个存储单元中，不能跨两个单元。如果一个单元空间不够容纳这个位段，那么该空间不用，而从下一个单元起开始存放
```

3.5.2. 取值

在我机器中结构体中先定义的位段成员占低位，后定义的位段成员占高位。这与具体实现有关。

位段结构体可以整体当做一个整型值看待，如果结构体过长，那么取值时会截取。

```
struct bits
{
```

```
    char a : 1;
    char b : 2;
};

struct bits b = {2, 5};
b.a == 0;
b.b == 1;
//给位段成员赋值，可以超出其取值范围，实际赋值时，是进行截取
```

3.5.3. 对齐

位段结构体对齐原则和一般结构体一样。

位字段中的每个成员没有地址，因此不可以进行取址操作，也不可进行 sizeof 操作。

3.6 长度为零的数组

长度为 0 的数组在标准 c 和 c++ 中是不允许的，如果使用长度为 0 的数组，编译时会产生错误，提示数组长度不能为 0。但在GNUc中，这种用法却是合法的。

```
char carry[0];    //sizeof(carry) = 0
int  iarry[0];    //sizeof(iarry) = 0
//长度为零的数组不占用存储空间
```

长度为 0 的数组主要是方便结构体内存缓冲区的管理。为如下缓冲区分配存储空间时，需要两次分配内存空间，两个成员的内存地址不连续，释放空间时也要释放两次。

```
struct sbuf
{
    unsigned int length;
    char *buf;
};//sizeof(sbuf) = 16
```

而使用长度为零的数组，如下，则不用如此麻烦，只用申请一次内存空间。

```
struct sbuf
{
    unsigned int length;
    char buf[0];
};//sizeof(sbuf) = 4

unsigned int length = 1024u;
struct sbuf *psbuf = (struct sbuf*)malloc(sizeof(sbuf) + sizeof(int)*length);
psbuf->length = length;
psbuf->buf[length-1] = 0;
```

4 预处理

预处理功能是C语言特有的功能，它是在对源程序正式编译前由预处理程序完成的。程序员在程序中用预处理命令来调用这些功能。使用预处理功能便于程序的修改、阅读、移植和调试，也便于实现模块化程序设计。预处理过程读入源代码，检查包含预处理指令的语句和宏定义，并对源代码进行响应的转换。预处理过程还会删除程序中的注释和多余的空白字符。

预处理指令是以#号开头的代码行。#号必须是该行除了任何空白字符外的第一个字符。#后是指令关键字，在关键字和#号之间允许存在任意个数的空白字符。整行语句构成了一条预处理指令，该指令将在编译器进行编译之前对源代码做某些转换。

预处理功能主要包括:

- 宏定义
- 文件包含
- 条件编译

4.1 预处理指令

#	空指令，无任何效果
#include	包含一个源代码文件
#define	定义宏
#undef	取消已定义的宏
#if	如果条件为真，则编译下面代码
#ifdef	如果宏已经定义，则编译下面代码
#ifndef	如果宏没有定义，则编译下面代码
#elif	前面 #if 条件不为真，当前条件为真，编译下面代码
#endif	结束一个 #if……#else 条件编译块
#error	停止编译并显示错误信息
#else	略
#line	可改变编译器用来指出警告和错误信息的文件和行号
#pragma	没有正式的定义。编译器可以自定义其用途。典型的用法是禁止或允许某些烦人的警告信息。

4.2 文件包含

```
1.  #include <my.h>
2.  #include "my.h"
```

- 第一种方法是用尖括号把头文件括起来。这种格式告诉预处理程序在编译器自带的或外部库的头文件中搜索被包含的头文件。
- 第二种方法是用双引号把头文件括起来。这种格式告诉预处理程序在当前被编译的应用程序的源代码文件中搜索被包含的头文件，如果找不到，再搜索编译器自带的头文件。

采用两种不同包含格式的理由在于，编译器是安装在公共子目录下的，而被编译的应用程序是在它们自己的私有子目录下的。一个应用程序既包含编译器提供的公共头文件，也包含自定义的私有头文件。采用两种不同的包含格式使得编译器能够在很多头文件中区别出一组公共的头文件。

4.3 宏定义

- 宏定义是用宏名来表示一个字符串，在宏展开时又以该字符串取代宏名，这只是一种简单的代换，字符串中可以含任何字符，可以是常数，也可以是表达式，预处理程序对它不作任何检查，如有错误，只能在编译已被宏展开后的源程序时发现。
- 宏定义不是说明或语句，在行末不必加分号，如加上分号则连分号也一起置换。
- 宏定义必须写在函数之外，其作用域为宏定义命令起到源程序结束，如要终止其作用域可以使用#undef命令
- 宏名若出现在字符串中，则不作替换
- 宏定义允许嵌套，在宏定义的字符串中可以使用已经定义的宏名,在宏展开时由预处理程序层层代换。
- 习惯上宏名用大写字母表示，以便于与变量区别，但也允许用小写字母
- 可以宏定义表示数据类型，使书写方便。
- 对输出格式作宏定义可以减少书写麻烦，便于调试程序。

4.3.1. 变量式宏定义

宏只是进行简单的字符串替换：

```
#define IN
#define OUT
```

```
#define TRUE        1u
#define FALSE       0u
#define MAX_SIZE    1024u
```

4.3.2. 函数式宏定义

宏可以类似函数一样使用：

```
#define MAX(a, b) ((a)>(b)?(a):(b))
```

- 参数没有类型，只是形式上的替换，不做类型检查，传参数时要格外小心。但正因为没有类型检查，所以可以将同一个宏定义用于任何数据类型，而无需针对每个数据类型都定义一个一样的函数。
- 定义时要注意每个参数要加括号，最外层也要加括号
- 有可能重复求值，小心定义这类宏

```
MAX(i++, j++) —>  ((i++)>(j++)?(i++):(j++))
//传入表达式，宏定义展开时表达式重复计算，产生错误
```

宏定义中，# 运算符能将其后的参数转换为字符串

```
#define TOSTR(str)    #str    //正确
#define TOSTR(str)    # str   //正确
#define HELLO         #helloworld    //错误

printf("%s", TOSTR(hello world));
//out: hello world
```

宏定义中，## 运算符能把前后两个预处理 token 连接成一个预处理 token

```
#define SIZEOF    size##of    //正确
#define SIZEOF    size ## of  //正确
#define SIZEOF(a,b)  a##b

int i;
printf("%d", SIZEOF i);
printf("%d", SIZEOF(size, of) i);
//out: 4
```

4.3.3. 宏定义与 typedef

- 宏定义仅仅是简单的字符串替换，不作类型检查，而 typedef 是为数据类型定义别名。两者看来相似，实则不同。在进行数据类型定义时，优先使用 typedef。

```
#define PSTR    char*
const PSTR str1, str2; —>  const char *str1, str2;
//宏定义仅仅是简单的字符串替换
//str1 是一个指针，指向一个常量字符串
//str2 是一个字符常量

typedef char* PSTR;
const PSTR str1, str2; —>  char* const str1, *const str2;
//typedef 不是字符串替换，这里是定义 PSTR 为一个字符指针
//因此 const PSTR str1 表示 str1 是一个字符指针常量，指针的值不可改变
//str2 和 str1 的类型一样
```

4.4 条件编译

- 如果用 #undef 取消宏定义，取消一个没有定义的宏不会报错。

C 语言头文件一般形式如下：

```
#ifndef MYAPP_H     //防止头文件重复包含
#define MYAPP_H
```

```
#ifdef _cplusplus //保证在 c++ 环境下， C 语言部分按照 C 语言方式编译链接
Extern “C”
{
#endif

/******************/
/*      body      */
/******************/

#ifdef _cplusplus
}
#endif

#endif
```

C++ 是面向对象语言，支持函数重载，而面向过程的 C 语言不支持，这导致 C 语言编译器和 C++编译器对函数名的处理不同。当 C 和 C++混合使用时，程序使用 C++编译器，程序中对 C 库函数的引用等也都会按照 C++的方式进行名字处理，这样，在链接时，将导致无法链接至 C 语言的库函数（因为 obj 文件里面函数名不一致），从而链接失败。

extern “C” 能够保证 C++ 程序的编译过程中，被引用的 C 语言函数还是按照 C 编译器的处理方式处理，这样最终能够正确链接至相应的函数。

4.5 内联函数

C 语言十分注重效率，往往有一些功能简单的小函数，其调用开销明显大于其运行开销，这时为了避免产生调用开销，往往会使用函数式宏定义，在调用处进行宏替换，以此避免调用。使用函数式宏定义，并没有定义函数，只是预处理时，把宏定义的代码替换到调用处，增加了可自行文件的大小，节约了调用开销。

但宏定义有很明显的缺点，它仅仅是文本替换，不作类型检查，实际使用时，很多细节也容易导致错误，可读性也很差。内联函数就是为了避免宏定义的这些缺点而提出的。

内联函数也会在函数调用处展开，避免调用开销，因而也会时可执行文件变大。内联函数是真正的函数，它的定义与一般函数一样，只是增加了关键字 inline。

4.6 其他预处理特性

4.6.1. line

预处理命令主要用于编译调试。

```
1. #include <stdio.h>
2.
3. int main()
4. {
5.     printf("%-5s : %-5d\n", __FILE__, __LINE__);
6.     int i;
7.     for (i = 0; i < 5; i++)
8.     {
9.         printf("%-5s : %-5d\n", __FILE__, __LINE__);
10.    }
11.        printf("%-5s : %-5d\n", __FILE__, __LINE__);
12.
13.    return 0;
14.}
15.//out:
16.//test.c : 5
17.//test.c : 9
18.//test.c : 9
19.//test.c : 9
20.//test.c : 9
21.//test.c : 9
22.//test.c : 11
```

以上代码可见，宏定义 __FILE__ 代指当前源文件的文件名，是一个字符串；

宏定义 __LINE__ 代指其当前在源文件中的行号。

```
1. #include <stdio.h>
2. #line 100    //指定 __LINE__ 在源文件中行号
3.
4. int main()
5. {
6.     printf("%-5s : %-5d\n", __FILE__, __LINE__);
7.     int i;
8.     for (i = 0; i < 5; i++)
9.     {
10.        printf("%-5s : %-5d\n", __FILE__, __LINE__);
11.    }
12.        printf("%-5s : %-5d\n", __FILE__, __LINE__);
13.
14.    return 0;
15.}
16.//out:
17.//test.c : 103
18.//test.c : 107
19.//test.c : 107
20.//test.c : 107
21.//test.c : 107
22.//test.c : 107
23.//test.c : 109
```

在当前源文件中重新指定 #line 的取值后，__LINE__ 的值发生了变化：

line(__LINE__) = line(__LINE__) - line(#line) + line

5. C 语言库

一个 C 语言运行库大致包含如下功能：

- 启动与退出：包括入口函数以及入口函数所依赖的其他函数等
- 标准函数：由 C 语言标准规定的 C 语言标准库所拥有的函数实现
- I/O：I/O 功能的封装和实现
- 堆：堆的封装和实现
- 语言实现：语言中一些特殊功能的实现
- 调试：实现调试功能的代码

ANSI C 标准库由 24 个 C 头文件组成，非常轻量。包括：

- 标准输入输出<stdio.h>
- 文件操作<tdio.h>
- 字符操作<type.h>
- 字符串操作<string.h>
- 数学函数<math.h>
- 资源管理<stdlib.h>
- 格式转换<stdlib.h>
- 日期/时间<time.h>
- 断言<assert.h>
- 各种类型的常数<limitsh>
- 变长参数<stdarg.h>
- 非局部跳转<setjmp.h>