



C语言培训

李绍华

Build a Smarter World

培训课程安排



7月	6号 (周三)	7号	8号	9号 (周六)	10号 (周日 修课)	11号 (周一)	12号
上午 (9:30~12:00)	基本语法、 变量、数 据类型	控制语句、 函数	数组	指针		内存段、 栈帧	排序算 法、内 存管理 (待定)
下午 (14:00~18:00)	运算符、 表达式	结构体、 枚举、联 合	指针	预处理 、编译 流程		队列、 栈、树、 链表	
晚上 (19:00~21:30)	课后练习+现场答疑						



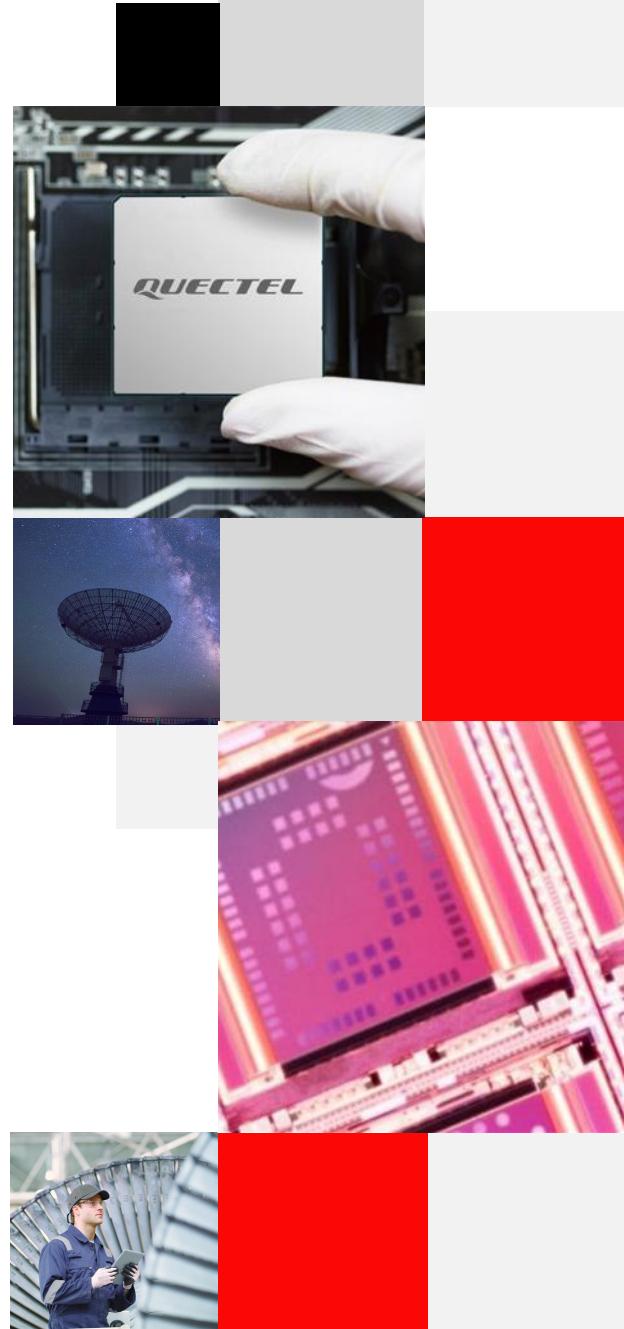
第一课 基本语法、变量、数据类型、运算符、表达式

Build a Smarter World

www.quectel.com

izzul.irfan@quectel.com
Thu Oct 03 09:21:00 CST 2024

izzul.irfan@quectel.com
Thu Oct 03 09:21:00 CST 2024



C语言结构、基本语法



C程序主要包括以下部分：

- 预处理器指令
- 函数
- 变量
- 语句 & 表达式
- 注释

```
#include <stdio.h>

int main()
{
    /* 我的第一个 C 程序 */
    printf("Hello, World! \n");

    return 0;
}
```

从较低层面考察，程序是由“符号”(token)序列组成的；符号是程序的一个基本组成单元，作用相当于一个句子中的单词。一个单词无论出现在哪个句子，它代表的意思是一样的，是一个表达意思的基本单元，与此类似，符号就是程序中的一个基本信息单元。然而组成符号的字符序列就不同，同一组字符序列在不同的上下文环境中所表达的意思是不同的。例如：

p->s="->" 前后两个->代表的意义完全不同。符号可以是关键字、标识符、常量、字符串值等。
编译器中负责将程序分解为一个一个符号的部分，一般称为“词法分析器”。

对于PPT而言：

单词的组合形成句子，句子的集合形成一页PPT；

对于程序而言：

符号的组合形成功能语句和声明，语句和声明的集合形成C程序。
符号如何组成更大的单元的语法细节最终决定了语义。

C语言结构、基本语法



注释：

```
/*单行注释*/
```

C99新增了另一种风格的注释，普遍用于C++和Java。这种新风格使用//符号创建注释，仅限于单行。

```
//这种注释只能写成一行。
```

```
int rigue; //这种注释也可置于此。
```

因为一行末尾就标志着注释的结束，所以这种风格的注释只需在注释开始处标明//符号即可。这种新形式的注释是为了解决旧形式注释存在的潜在问题。

```
/*
希望能运行。
```

```
y = 200;
```

```
/*其他内容已省略。 */
```

分号；

在C程序中，分号是语句结束符。也就是说，每个语句必须以分号结束。

花括号、函数体和块

```
{ ..  
.}  
}
```

花括号把main()函数括起来。一般而言，所有的C函数都使用花括号标记函数体的开始和结束。这是规定，不能省略。只有花括号（{}）能起这种作用，圆括号（()）和方括号（[]）都不行。

```
#include <stdio.h>
int main(void) /* 一个简单的C程序 */
{
    int num; /* 定义一个名为num的变量 */

    num = 1; /* 为num赋一个值 */
    printf("I am a simple "); /* 使用printf()函数 */
    printf("computer.\n");
    printf("My favorite number is %d because it is first.\n", num);

    return 0;
}
```

C语言结构、基本语法

声明：

声明（declaration）。声明是C语言最重要的特性之一。在该例中，声明完成了两件事。其一，在函数中有一个名为num的变量（variable）。其二，int表明num是一个整数（即，没有小数点或小数部分的数）。int是一种数据类型。编译器使用这些信息为num变量在内存中分配存储空间。分号在C语言中是大部分语句和声明的一部分。
int是C语言的一个关键字（keyword），表示一种基本的C语言数据类型。关键字是语言定义的单词，不能做其他用途。例如，不能用int作为函数名和变量名。示例中的num是一个标识符（identifier），也就一个变量、函数或其他实体的名称。在C语言中，所有变量都必须先声明才能使用。这意味着必须列出程序中用到的所有变量名及其类型。

```
#include <stdio.h>
int main(void) /* 一个简单的C程序 */
{
    int num; /* 定义一个名为num的变量 */

    num = 1; /* 为num赋一个值 */
    printf("I am a simple "); /* 使用printf()函数 */
    printf("computer.\n");
    printf("My favorite number is %d because it is first.\n", num);

    return 0;
}
```

标识符、命名：

C 标识符是用来标识变量、函数，或任何其他用户自定义项目的名称。一个标识符以字母 A-Z 或 a-z 或下划线 _ 开始，后跟零个或多个字母、下划线和数字（0-9）。C 标识符内不允许出现标点字符，比如 @、\$ 和 %。C 是区分大小写的编程语言。因此，在 C 中，Manpower 和 manpower 是两个不同的标识符。C99和C11允许使用更长的标识符名，但是编译器只识别前63个字符。（以前 C90 只允许6个字符，这是一个很大的进步。旧式编译器通常最多只允许使用8个字符。）实际上，你可以使用更长的字符，但是编译器会忽略超出的字符。也就是说，如果有两个标识符名都有63个字符，只有一个字符不同，那么编译器会识别这是两个不同的名称。如果两个标识符都是64个字符，只有最后一个字符不同，那么编译器可能将其视为同一个名称，也可能不会。标准并未定义在这种情况下会发生什么

关键字

关键字	说明
auto	声明自动变量
break	跳出当前循环
case	开关语句分支
char	声明字符型变量或函数返回值类型
const	定义常量, 如果一个变量被 const 修饰, 那么它的值就不能再被改变
continue	结束当前循环, 开始下一轮循环
default	开关语句中的"其它"分支
do	循环语句的循环体
double	声明双精度浮点型变量或函数返回值类型
else	条件语句否定分支 (与 if 连用)
enum	声明枚举类型
extern	声明变量或函数是在其它文件或本文件的其他位置定义
float	声明浮点型变量或函数返回值类型
for	一种循环语句

if	条件语句
int	声明整型变量或函数
long	声明长整型变量或函数返回值类型
register	声明寄存器变量
return	子程序返回语句 (可以带参数, 也可不带参数)
short	声明短整型变量或函数
signed	声明有符号类型变量或函数
sizeof	计算数据类型或变量长度 (即所占字节数)
static	声明静态变量
struct	声明结构体类型
switch	用于开关语句
typedef	用以给数据类型取别名
unsigned	声明无符号类型变量或函数
union	声明共用体类型
void	声明函数无返回值或无参数, 声明无类型指针
volatile	说明变量在程序执行中可被隐含地改变
while	循环语句的循环条件

关键字

C99 新增关键字

_Bool	_Complex	_Imaginary	inline	restrict
-------	----------	------------	--------	----------

C11 新增关键字

_Alignas	_Alignof	_Atomic	_Generic	_Noreturn
_Static_assert	_Thread_local			

这些关键字是C中的保留符号，不能使用其作为常量名、变量名或者其他标识符名称。

什么是内存？作用是什么？

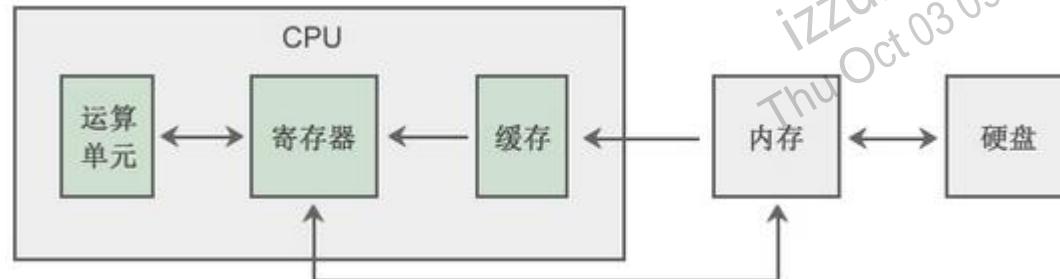
C程序 = 代码(语句)+ 数据(变量)

程序是保存在硬盘中的，要载入内存才能运行，CPU也被设计为只能从内存中读取数据和指令。

对于CPU来说，**内存仅仅是一个存放指令和数据的地方(CPU可以读取的存储介质)**，并**不能在内存中完成计算功能**，内存是通过地址来管理的。

例如要计算 $a = b + c$ ；必须将 a、b、c 变量所在内存地址上的内容都读取到CPU内部才能进行加法运算。

寄存器也可以理解为内存，其有地址，用于临时保存变量的值。



C变量和数据类型

前面我们说了内存，以及内存地址。C程序最终生成的指令和需要操作的数据也都存放于内存上，那么如何访问内存？

1、显而易见的是通过内存地址---后续介绍的指针

2、可以通过给内存命名的方式---变量

内存很大，需要将内存划分出很多块，因此要指定划分出每块内存的大小，也即在C语言中要声明变量的类型
右图为C语言的数据类型。

```
int main(void)
{
    int num;

    printf("hello world\n");

    return 0;
}
```

int num;

在内存中找了一块区域，命名为num，用它来存放整形int数据；变量num，关于变量的命名规则，前面已经介绍。

C语言包含的数据类型如下图所示：



作用域

当变量在程序的某个部分被声明，它只有在程序的一定区域才能被访问。这个区域由标识符的作用域决定。标识符的作用域就是程序中该标识符可以被使用的区域。

编译器可以确定4种不同类型的作用域----文件作用域、函数作用域、代码块作用域和原型作用域。
标识符声明的位置决定它的作用域。

代码块作用域：

位于一对花括号之间的所有语句成为一个代码块。任何在代码块的开始位置位置声明的标识符都具有代码块作用域。当代码块处于嵌套时，内层代码块有一个标识符的名字与外层代码块的一个标识符同名，内层的那个标识符就将隐藏外层的标识符--外层的那个标识符无法在内层代码块中通过名字访问。**C语言的就近原则**

文件作用域

任何在所有代码块之外声明的标识符具有文件作用域，它表示这些标识符从他们的声明之处直到它所在的源文件结尾处都是可以访问的。**如定义在C文件起始的全局变量,其内存地址在全局区内,在程序运行开始的时候被加载和初始化**

原型作用域

只适用于在函数原型中声明的参数名。**如函数的形参, 其在函数调用的时候才分配内存, 可能分配在内存上, 也可能在寄存器里, 函数调用结束, 形参的内存被释放。**

函数作用域

适用于语句标签，语句标签用于goto语句。一个函数中所有语句标签必须唯一

```

main.c func.c 1.C
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int a;
5 int b(int c);
6 int d(int e);
7 {
8     int f;
9     int g(int h); // Function g is declared here
10    ...
11 }
12     int f,g,i;
13     ...
14 }
15 {
16     int i;
17     ...
18 }
19 }
```

extern变量

程序的编译以源程序(单个C文件)为单位;
一个C文件可能要引用在另一个C文件里定义的变量;
这个时候需要使用extern 声明一个外部变量。

这样可以实现几个源文件可以共享同一个变量。

变量的extern声明不是定义，这个规则有一个例外；

extern int i = 0; 等效于int i=0; -----为了防止一个变量被使用extern 不同的类型声明

```
extern int i; // 静态存储期限  
              // 文件作用域  
              // 什么链接?  
  
void f(void)  
{  
    extern int j; // 静态存储期限  
                  // 块作用域  
                  // 什么链接?  
}
```

声明与定义

什么是声明？什么是定义？有何区别？

```
int num;  
extern int num;
```

什么是定义：

所谓的定义就是(编译器)创建一个对象，为这个对象分配一块内存并给它取上一个名字，这个名字就是我们经常所说的变量名或对象名。但注意，这个名字一旦和这块内存匹配起来，它们就同生共死，终生不离不弃。并且这块内存的位置也不能被改变。一个变量或对象在一定的作用域内只能被定义一次，如果定义多次，编译器会提示你重复定义同一个变量或对象。

什么是声明：

- 1: 告诉编译器，这个名字已经匹配到一块内存上了，下面的代码用到变量或对象是在别的地方定义的。声明可以出现多次。
- 2: 告诉编译器，我这个名字我先预定了，别的地方再也不能用它来作为变量名或对象名。

定义声明最重要的区别：定义创建了对象并为这个对象分配了内存(变量的类型决定分配大小)，声明没有分配内存；

常量与变量



有些数据类型在程序使用之前已经预先设定好了，在整个程序的运行过程中没有变化，这些称为常量（constant）。其他数据类型在程序运行期间可能会改变或被赋值，这些称为变量（variable）。

const标识符用来表示一个对象的不可变的性质，

```
const int b = 20;
```

在后面的代码中就不能改变变量b的值了，b中的值永远是20。

如果代码中执行了b++；则编译器会报错；

static 变量



static 可用于全部变量,而无需要考虑变量声明的位置

```
int i; // 静态存储期限  
        // 文件作用域  
        // 外部链接  
  
void f(void)  
{  
    int j; // 自动存储期限  
           // 块作用域  
           // 无链接  
}
```

```
static int i; // 静态存储期限  
              // 文件作用域  
              // 内部链接  
  
void f(void)  
{  
    static int j; // 静态存储期限  
                 // 块作用域  
                 // 无链接  
}
```

```
void trystat(void)  
{  
    int fade = 1;  
    static int stay = 1;  
    printf("fade = %d and stay = %d\n", fade++, stay++);  
}
```

- 块内的static变量只在程序执行前进行一次初始化, 而auto变量则会在每次出现时进行初始化(当然, 需假设它有初始化式)。
- 每次函数被递归调用时, 它都会获得一组新的auto变量。但是, 如果函数含有static变量, 那么此函数的全部调用都可以共享这个static变量。
- 虽然函数不应该返回指向auto变量的指针, 但是函数返回指向static变量的指针是没有错误的。

static全局变量与普通的全局变量有什么区别?

【标准答案】static全局变量只初使化一次, 同时限定作用域; 防止在其他源文件单元中被引用;

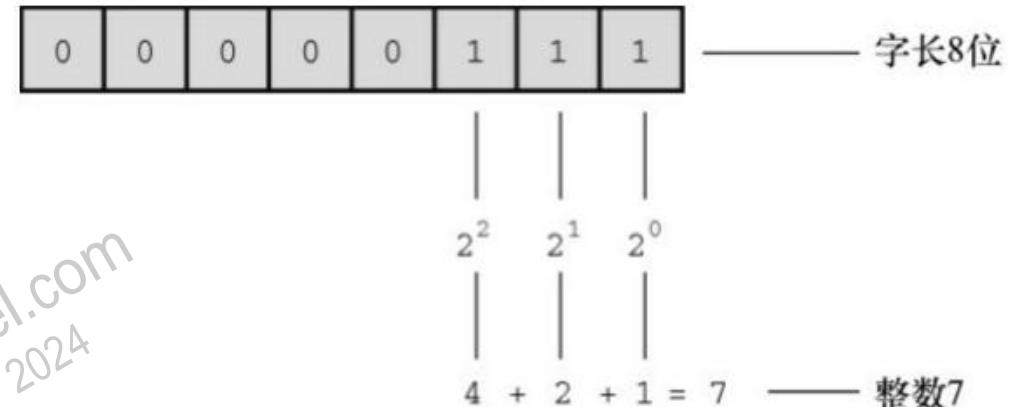
static局部变量和普通局部变量有什么区别

【标准答案】static局部变量只被初始化一次, 下一次依据上一次结果值;

C变量和数据类型

类型	存储大小	值范围
char	1 字节	-128 到 127 或 0 到 255
unsigned char	1 字节	0 到 255
signed char	1 字节	-128 到 127
int	2 或 4 字节	-32,768 到 32,767 或 -2,147,483,648 到 2,147,483,647
unsigned int	2 或 4 字节	0 到 65,535 或 0 到 4,294,967,295
short	2 字节	-32,768 到 32,767
unsigned short	2 字节	0 到 65,535
long	4 字节	-2,147,483,648 到 2,147,483,647
unsigned long	4 字节	0 到 4,294,967,295

使用二进制码在内存中存储整数7



类型	存储大小	值范围	精度
float	4 字节	1.2E-38 到 3.4E+38	6 位有效位
double	8 字节	2.3E-308 到 1.7E+308	15 位有效位
long double	16 字节	3.4E-4932 到 1.1E+4932	19 位有效位

输入与输出

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

int main(void)
{
    float weight; /* 你的体重 */
    float value; /* 相等重量的白金价值 */
    printf("Are you worth your weight in platinum?\n");
    printf("Let's check it out.\n");
    printf("Please enter your weight in pounds: ");
    /* 获取用户的输入 */
    scanf("%f", &weight);
    /* 假设白金的价格是每盎司$1700 */
    /* 14.5833用于把英镑常衡盎司转换为金衡盎司 */
    value = 1700.0 * weight * 14.5833;

    printf("Your weight in platinum is worth %.2f.\n", value);
    printf("You are easily worth that! If platinum prices drop,\n");
    printf("eat more to maintain your value.\n");
    return 0;
}
```

Printf的参数陷阱:

程序员要负责确保转换说明的数量、类型与后面参数的数量、类型相匹配。现在，C语言通过函数原型机制检查函数调用时参数的个数和类型是否正确。但是，该机制对printf()和scanf()不起作用，因为这两个函数的参数个数可变。

```
printf("%d\n", n, m); /* 参数太多 */
```

```
printf("%d %d %d\n", n); /* 参数太少 */
```

所有编译器都能顺利编译并运行该程序，但其中大部分会给出警告。的确，有些编译器会捕获到这类问题，然而C标准对此未作要求。因此，计算机在运行时可能不会捕获这类错误。如果程序正常运行，很难觉察出来。如果程序没有打印出期望值或打印出意想不到的值，你才会检查printf()函数中的参数个数和类型是否得当。

整数溢出(int/unsigned int)

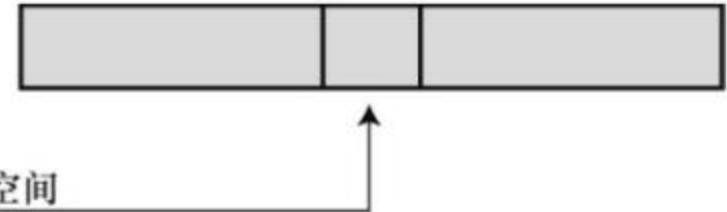
声明int类型变量

```
int erns;  
int hogs, cows, goats;
```

21、32、14和94）都是整型常量或整型字面量。C语言把不含小数点和指数的数作为整数。因此，22和-44都是整型常量，但是22.0和2.2E1则不是。C语言把大多数整型常量视为int类型

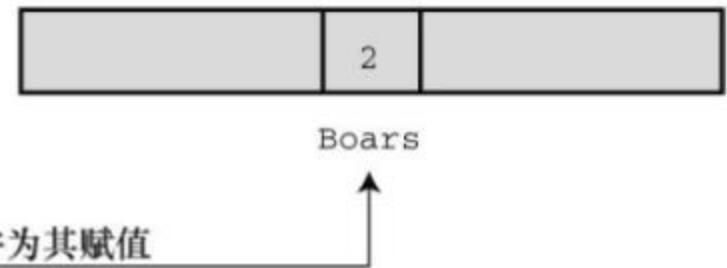
int sows;

创建内存空间



int boars=2;

创建内存空间并为其赋值



整数溢出(int/unsigned int)

如果整数超出了相应类型的取值范围会怎样？下面分别将有符号类型和无符号类型的整数设置为比最大值略大，看看会发生什么（printf()函数使用%u说明显示unsigned int类型的值）。

```
int main(void)
{
    int i = 2147483647;
    unsigned int j = 4294967295;
    printf("%d-%d %d %d\n", sizeof(int), i, i+1, i+2);
    printf("%u %u %u\n", j, j+1, j+2);
    return 0;
}
```

整数 i 也是类似的情况。它们主要的区别是，在超过最大值时，unsigned int 类型的变量 j 从 0 开始；而 int 类型的变量 i 则从 -2147483648 开始。注意，当 i 超出（溢出）其相应类型所能表示的最大值时，系统并未通知用户。因此，在编程时必须自己注意这类问题。

溢出行为是未定义的行为，C 标准并未定义有符号类型的溢出规则。以上描述的溢出行为比较有代表性，但是也可能会出现其他情况。

整数溢出(int/unsigned int)

十进制

2157也可以写成：

$$2 \times 10^3 + 1 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$$

因为这种书写数字的方法是基于10的幂， 所以称以10为基底书写2157。

二进制：

//合法的二进制

```
int a = 0b101; //换算成十进制为 5
```

//非法的二进制

```
int m = 101010; //无前缀 0B, 相当于十进制
```

计算机只是识别0和1，认为关闭和打开

因此，计算机适用基底为2的数制系统。它用2的幂而不是10的幂。以2为基底表示的数字被称为二进制数（binary number）。

二进制中的2和十进制中的10作用相同。例如，二进制数1101可表示为： $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

以十进制数表示为： $1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 = 13$

十六进制

（简写为hex或下标16）是一种基数为16的计数系统，是一种逢16进1的进位制。通常用数字0、1、2、3、4、5、6、7、8、9和字母A、B、C、D、E、F (a、b、c、d、e、f) 表示，其中:A~F表示10~15，这些称作十六进制数字。

如何知道10000是十进制、十六进制还是二进制？在C语言中，用特定的前缀表示使用哪种进制。0x或0X前缀表示十六进制值，所以十进制数16表示成十六进制是0x10或0X10

整数

short int类型（或者简写为short） 占用的存储空间可能比int类型少， 常用于较小数值的场合以节省空间。 与int类似， short是有符号类型。

long int或long占用的存储空间可能比int多， 适用于较大数值的场合。 与int类似， long是有符号类型。

long int或long占用的存储空间可能比int多， 适用于较大数值的场合。 与int类似， long是有符号类型。

在C90标准中， 添加了unsigned long int或unsigned long和unsigned int或unsigned short类型。 C99标准又添加了unsigned long long int或unsigned long long。

为什么说short类型“可能”比int类型占用的空间少， long类型“可能”比int类型占用的空间多？ 因为C语言只规定了short占用的存储空间不能多于int， long占用的存储空间不能少于int。 这样规定是为了适应不同的机器。

通常， 程序代码中使用的数字（如， 2345） 都被储存为int类型。 如果使用1000000这样的大数字， 超出了int类型能表示的范围， 编译器会将其视为long int类型（假设这种类型可以表示该数字）。 如果数字超出long可表示的最大值， 编译器则将其视为unsigned long类型。 如果还不够大， 编译器则将

其视为long long或unsigned long long类型（前提是编译器能识别这些类型

有符号还是无符号 (char)

char类型用于储存字符（如，字母或标点符号），但是从技术层面看，char是整数类型。因为char类型实际上储存的是整数而不是字符。计算机使用数字编码来处理字符，即用特定的整数表示特定的字符。美国最常用的编码是ASCII编码，大多数编辑器也使用此编码格式。标准ASCII码的范围是0~127，只需7位二进制数即可表示。通常，char类型被定义为8位的存储单元，因此容纳标准ASCII码绰绰有余。C语言把1字节定义为char类型占用的位（bit）数，因此无论是16位还是32位系统，都可以使用char类型。

如果要把一个字符常量初始化为字母 A，不必背下 ASCII 码，用计算机语言很容易做到。通过以下初始化把字母A赋给grade即可：

```
char grade = 'A'; //单引号必不可少
```

实际上，字符是以数值形式储存的，所以也可使用数字代码值来赋值：

```
char grade = 65; /* 对于ASCII，这样做没问题，但这是一种不好的编程风格 */
```

在本例中，虽然65是int类型，但是它在char类型能表示的范围内，所以将其赋值给grade没问题。由于65是字母A对应的ASCII码，因此本例是把A赋给grade。注意，能这样做的前提是系统使用ASCII码。

使用ASCII码时，注意数字和数字字符的区别。例如，字符4对应的ASCII码是52。'4'表示字符4，而不是数值4。

有符号还是无符号 (char)

奇怪的是， C语言将字符常量视为int类型而非char类型。

```
9 |     char grade = 'B';|
```

本来'B'对应的数值66储存在32位的存储单元中， 现在却可以储存在8位的存储单元中（grade）。 利用字符常量的这种特性， 可以定义一个字符常量'FATE'， 即把4个独立的8位ASCII码储存在一个32位存储单元中。 如果把这样的字符常量赋给char类型变量grade， 只有最后8位有效。 因此， grade的值是'E'。

```
int main(void)
{
    int i = 2147483647;
    unsigned int j = 4294967295;

    char grade = 'FATE';

    printf("%c-%d-%d %d %d\n", grade, sizeof(int), i, i+1, i+2);
}
```

C:\Users\sharon.l\Desktop\C\项目2.exe

E-4-2147483647 -2147483648 -2147483647
4294967295 0 1

有些C编译器把char实现为有符号类型， 这意味着char可表示的范围是-128~127。 而有些C编译器把char实现为无符号类型， 那么char可表示的范围是0~255。 请查阅相应的编译器手册， 确定正在使用的编译器如何实现char类型。 根据C90标准， C语言允许在关键字char前面使用signed或unsigned。 这样， 无论编译器默认char是什么类型， signed char表示有符号类型， 而unsigned char表示无符号类型。 这在用char类型处理小整数时很有用。 如果只用char处理字符， 那么char前面无需使用任何修饰符。

转义字符

转义序列	含义
\a	警报 (ANSI C)
\b	退格
\f	换页
\n	换行
\r	回车
\t	水平制表符
\v	垂直制表符
\\\	反斜杠 (\)
\'	单引号
\"	双引号
\?	问号
\ooo	八进制值 (oo 必须是有效的八进制数, 即每个 o 可表示 0~7 中的一个数)
\xhh	十六进制值 (hh 必须是有效的十六进制数, 即每个 h 可表示 0~f 中的一个数)

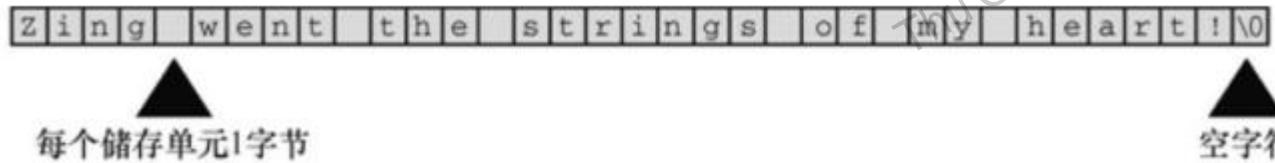
使用C90新增的警报字符 (\a)

是否能产生听到或看到的警报, 取决于计算机的硬件, 蜂鸣是最常见的警报 (在一些系统中, 警报字符不起作用)。

C99标准添加了 _Bool类型, 用于表示布尔值, 即逻辑值true和false。因为C语言用值1表示true, 值0表示false, 所以 _Bool类型实际上也是一种整数类型。但原则上它仅占用1位存储空间, 因为对0和1而言, 1位的存储空间足够了

字符串

字符串（character string）是一个或多个字符的序列，如下所示：“Quectel”
双引号不是字符串的一部分。 双引号仅告知编译器它括起来的是字符串， 正如单引号用于标识单个字符一样。



C语言没有专门用于储存字符串的变量类型， 字符串都被储存在char类型的数组中。 数组由连续的存储单元组成， 字符串中的字符被储存在相邻的存储单元中， 每个单元储存一个字符

数组末尾位置的字符\0。 这是空字符（null character）， C语言用它标记字符串的结束。 空字符不是数字0， 它是非打印字符， 其ASCII码值是（或等价于）0。 C中的字符串一定以空字符结束， 这意味着数组的容量必须至少比待存储字符串中的字符数多1。

什么是数组？ 可以把数组看作是一行连续的多个存储单元。用更正式的说法是， 数组是同类型数据元素的有序序列
`char name[40];`

类型大小

sizeof是C语言的内置运算符，以字节为单位给出指定类型的大小。

```
/*
C99为类型大小提供%zd转换说明 */
```

```
printf("Type int has a size of %zd bytes.\n", sizeof(int));
```

```
#define PRAISE "You are an extraordinary being."
int main(void)
{
    char name[40];
    printf("What's your name? ");
    scanf("%s", name);
    printf("Hello, %s.%s\n", name, PRAISE);
    printf("Your name of %zd letters occupies %zd memory cells.\n",
        strlen(name), sizeof name);
    printf("The phrase of praise has %zd letters ",
        strlen(PRAISE));
    printf("and occupies %zd memory cells.\n", sizeof PRAISE);

    return 0;
}
```

strlen()函数

函数给出字符串中的字符长度。

二进制浮点数：

一个普通的浮点数0.527， 表示如下： $5/10 + 2/100 + 7/1000$ 从左往右， 各分母都是10的递增次幂。

在二进制小数中， 使用2的幂作为分母， 所以二进制小数.101表示为： $1/2 + 0/4 + 1/8$

用十进制表示法为： $0.50 + 0.00 + 0.125$ 即是0.625。

100.101(2)--->1.00101*2^2(2进制的科学计数法)

乘2取整法， 即每一步将十进制小数部分乘以2， 所得积的小数点左边的数字（0或1）作为二进制表示法中的数字， 直到满足你的精确度为止

转换过程：

0.874的转换过程（取精度为6位）：

$0.874 \times 2 = 1.748$ 小数点左边为 1

$0.748 \times 2 = 1.496$ 小数点左边为 1

$0.496 \times 2 = 0.992$ 小数点左边为 0

$0.992 \times 2 = 1.984$ 小数点左边为 1

$0.984 \times 2 = 1.968$ 小数点左边为 1

$0.968 \times 2 = 1.936$ 小数点左边为 1

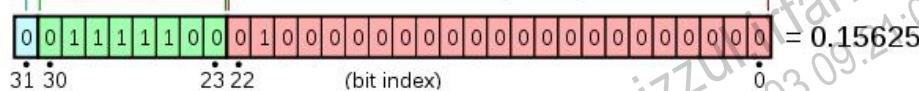
十进制：123.874 二进制：1111011.110111

浮点数

float

单精度浮点值。单精度是这样的格式，1位符号，8位指数，23位小数。

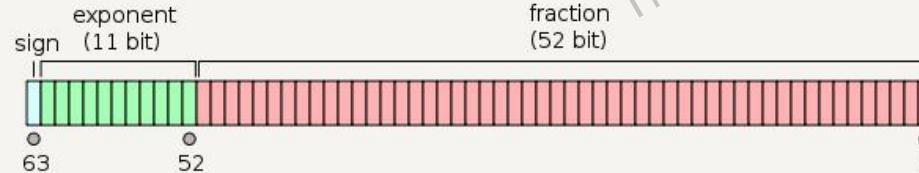
sign exponent (8 bits) fraction (23 bits)



$$= 0.15625$$

double

双精度浮点值。双精度是1位符号，11位指数，52位小数。



用二进制的科学计数法第一位都是1,可以将小数点前面的1省略, 所以23bit的尾数部分, 可以表示的精度却变成了 24bit, 道理就是在这里。

那24bit能精确到小数点后几位呢, 我们知道9的二进制表示为1001, 所以4bit能精确十进制中的1位小数点, 24bit就能使float能精确到小数点后6位, 而对于指数部分, 因为指数可正可负, 8位的指数位能表示的指数范围就应该为:-127-128了, 所以指数部分的存储采用移位存储, 存储的数据为原数据+127。

首先看下8.25, 用二进制的科学计数法表示为

:1.00001*2^3 按照上面的存储方式, 符号位为0, 表示为正; 指数位为3+127=130, 小数部分为 1.00001, 故8.25的存储方式如下: 0xbffff380: 01000001000001000000000000000000

分解如下: 0--10000010--000010000000000000000000

符号位为0, 指数部分为10000010, 小数部分为 000010000000000000000000

Printf格式控制符

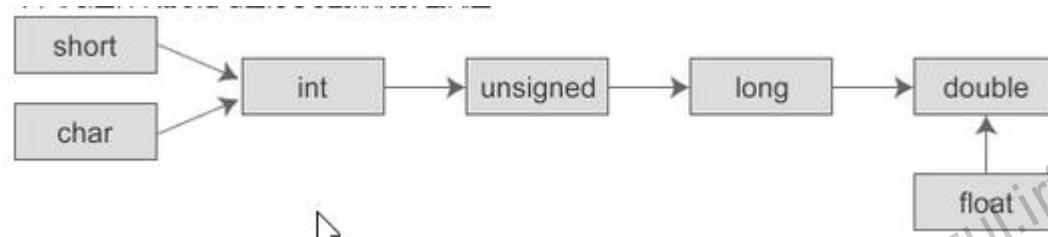
格式控制符	说明
%c	输出一个单一的字符
%hd、%d、%ld	以十进制、有符号的形式输出 short、int、long 类型的整数
%hu、%u、%lu	以十进制、无符号的形式输出 short、int、long 类型的整数
%ho、%o、%lo	以八进制、不带前缀、无符号的形式输出 short、int、long 类型的整数
%#ho、%#o、%#lo	以八进制、带前缀、无符号的形式输出 short、int、long 类型的整数
%hx、%x、%lx %hX、%X、%lX	以十六进制、不带前缀、无符号的形式输出 short、int、long 类型的整数。如果 x 小写，那么输出的十六进制数字也小写；如果 X 大写，那么输出的十六进制数字也大写。
%#hx、%#x、%#lx %#hX、%#X、%#lX	以十六进制、带前缀、无符号的形式输出 short、int、long 类型的整数。如果 x 小写，那么输出的十六进制数字和前缀都小写；如果 X 大写，那么输出的十六进制数字和前缀都大写。
%f、%lf	以十进制的形式输出 float、double 类型的小数
%e、%le %E、%IE	以指数的形式输出 float、double 类型的小数。如果 e 小写，那么输出结果中的 e 也小写；如果 E 大写，那么输出结果中的 E 也大写。
%g、%lg %G、%IG	以十进制和指数中较短的形式输出 float、double 类型的小数，并且小数部分的最后不会添加多余的 0。如果 g 小写，那么当以指数形式输出时 e 也小写；如果 G 大写，那么当以指数形式输出时 E 也大写。
%s	输出一个字符串

类型转化

1) 将一种类型的数据赋值给另外一种类型的变量时就会发生自动类型转换，例如：float f = 100;
100 是 int 类型的数据，需要先转换为 float 类型才能赋值给变量 f。再如：int n = f;
f 是 float 类型的数据，需要先转换为 int 类型才能赋值给变量 n。

在赋值运算中，赋值号两边的数据类型不同时，需要把右边表达式的类型转换为左边变量的类型，这可能会导致数据失真，或者精度降低；所以说，自动类型转换并不一定是安全的。对于不安全的类型转换，编译器一般会给出警告。

- 在不同类型的混合运算中，编译器也会自动地转换数据类型，将参与运算的所有数据先转换为同一种类型，然后再进行计算。转换的规则如下：转换按数据长度增加的方向进行，以保证数值不失真，或者精度不降低。例如，int 和 long 参与运算时，先把 int 类型的数据转成 long 类型后进行运算。
- 所有的浮点运算都是以双精度进行的，即使运算中只有 float 类型，也要先转换为 double 类型，才能进行运算。
- char 和 short 参与运算时，必须先转换成 int 类型。



```
int main()
{
    float PI = 3.14159;
    int s1, r = 5;
    double s2;
    s1 = r * r * PI;
    s2 = r * r * PI;
    printf("s1=%d, s2=%f\n", s1, s2);
    return 0;
}
```

类型转化

强制类型转换

自动类型转换是编译器根据代码的上下文环境自行判断的结果，有时候并不是那么“智能”，不能满足所有的需求。如果需要，程序员也可以自己在代码中明确地提出要进行类型转换，这称为强制类型转换。

自动类型转换是编译器默默地、隐式地进行的一种类型转换，不需要在代码中体现出来；强制类型转换是程序员明确提出的、需要通过特定格式的代码来指明的一种类型转换。换句话说，自动类型转换不需要程序员干预，强制类型转换必须有程序员干预。

强制类型转换的格式为：(type_name) expression

```
int main()
{
    int sum = 103; //总数
    int count = 7; //数目
    double average; //平均数
    average = sum / count;
    printf("Average is %lf!\n", average);
    return 0;
}
```

思考Average 输出多少？为什么没小数，如何改进？

类型转换只是临时性的

```
int main()
{
    double total = 400.8; //总价
    int count = 5; //数目
    double unit; //单价
    int total_int = (int)total;
    unit = total / count;
    printf("total=%lf, total_int=%d, unit=%lf\n", total, total_int, unit);
    return 0;
}
```

算术运算符

下表显示了 C 语言支持的所有算术运算符。假设变量 A 的值为 10，变量 B 的值为 20，则：

运算符	描述	实例
+	把两个操作数相加	A + B 将得到 30
-	从第一个操作数中减去第二个操作数	A - B 将得到 -10
*	把两个操作数相乘	A * B 将得到 200
/	分子除以分母	B / A 将得到 2
%	取模运算符，整除后的余数	B % A 将得到 0
++	自增运算符，整数值增加 1	A++ 将得到 11
--	自减运算符，整数值减少 1	A-- 将得到 9

关系运算符

下表显示了 C 语言支持的所有关系运算符。假设变量 A 的值为 10，变量 B 的值为 20，则：

运算符	描述	实例
==	检查两个操作数的值是否相等，如果相等则条件为真。	(A == B) 为假。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。	(A != B) 为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。	(A > B) 为假。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。	(A < B) 为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。	(A >= B) 为假。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。	(A <= B) 为真。



C 语言允许用++和--，简单来看++表示操作数加1,--表示操作数减1；但实际上是一种误导，实际上自增和自减运算符的使用很复杂，因为，++和-既可以作为前缀运算符(++i 和 --i)，也可以作为后缀运算符(i++ , i--) 使用。另一个，和赋值运算符一样，++和-也有副作用：他们会改变操作数的值。

```
49 int i = 1;
50 printf("i is %d\n", ++i);
51 printf("i is %d\n", i);
52
53 int i = 1;
54 printf("i is %d\n", i++);
55 printf("i is %d\n", i);
```

++i 意味着立即自增i，而i++则先用i的原始值，稍后再自增i，这个稍后？C 并没有给出精确的时间，但是可以放心的假设i将在下一条语句前进行自增。

i=1;

j=2;

K = ++i + j++; 程序结束后值分别是多少？

需要记住的是，后缀的++和-比一元的正号、负号优先级高，而且都是左结合的。前缀++和-与一元的正号、负号优先级相同，都是右结合的。

运算符



逻辑运算符

下表显示了 C 语言支持的所有关系逻辑运算符。假设变量 A 的值为 1, 变量 B 的值为 0, 则:

运算符	描述	实例
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。	(A && B) 为假。
	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	(A B) 为真。
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。	!(A && B) 为真。

杂项运算符 ↳ sizeof & 三元

下表列出了 C 语言支持的其他一些重要的运算符，包括 `sizeof` 和 `? :`。

运算符	描述	实例
<code>sizeof()</code>	返回变量的大小。	<code>sizeof(a)</code> 将返回 4, 其中 a 是整数。
<code>&</code>	返回变量的地址。	<code>&a;</code> 将给出变量的实际地址。
<code>*</code>	指向一个变量。	<code>*a;</code> 将指向一个变量。
<code>? :</code>	条件表达式	如果条件为真 ? 则值为 X : 否则值为 Y

位运算符

运算符	描述	实例
&	按位与操作，按二进制位进行“与”运算。运算规则： 0&0=0; 0&1=0; 1&0=0; 1&1=1;	(A & B) 将得到 12, 即为 0000 1100
	按位或运算符，按二进制位进行“或”运算。运算规则： 0 0=0; 0 1=1; 1 0=1; 1 1=1;	(A B) 将得到 61, 即为 0011 1101
^	异或运算符，按二进制位进行“异或”运算。运算规则： 0^0=0; 0^1=1; 1^0=1; 1^1=0;	(A ^ B) 将得到 49, 即为 0011 0001
~	取反运算符，按二进制位进行“取反”运算。运算规则： ~1=-2; ~0=-1;	(~A) 将得到 -61, 即为 1100 0011, 一个有符号二进制数的补码形式。
<<	二进制左移运算符。将一个运算对象的各二进制位全部左移若干位（左边的二进制位丢弃，右边补0）。	A << 2 将得到 240, 即为 1111 0000
>>	二进制右移运算符。将一个数的各二进制位全部右移若干位，正数左补0；负数左补1，右边丢弃。	A >> 2 将得到 15, 即为 0000 1111

赋值运算符

赋值运算符

下表列出了 C 语言支持的赋值运算符：

运算符	描述	实例
=	简单的赋值运算符，把右边操作数的值赋给左边操作数	$C = A + B$ 将把 $A + B$ 的值赋给 C
+=	加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数	$C += A$ 相当于 $C = C + A$
-=	减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数	$C -= A$ 相当于 $C = C - A$
*=	乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数	$C *= A$ 相当于 $C = C * A$
/=	除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数	$C /= A$ 相当于 $C = C / A$
%=	求模且赋值运算符，求两个操作数的模赋值给左边操作数	$C %= A$ 相当于 $C = C \% A$
<<=	左移且赋值运算符	$C <<= 2$ 等同于 $C = C << 2$
>>=	右移且赋值运算符	$C >>= 2$ 等同于 $C = C >> 2$
&=	按位与且赋值运算符	$C &= 2$ 等同于 $C = C \& 2$
^=	按位异或且赋值运算符	$C ^= 2$ 等同于 $C = C ^ 2$
=	按位或且赋值运算符	$C = 2$ 等同于 $C = C 2$

位操作

$\sim(10011010) \text{ ----} \Rightarrow (01100101)$

需要打开一个值中的特定位， 同时保持其他位不变

`flags = flags | MASK;`

因为使用`|`运算符， 任何位与0组合， 结果都为本身； 任何位与1组合， 结果都为1

假设`flags`是00001111， `MASK`是10110110。 下面的表达式：

`(00001111) | (10110110) // 表达式`

其结果为： `(10111111) // 结果值`

`flags |= MASK;`

和打开特定的位类似， 有时也需要在不影响其他位的情况下关闭指定的位。

`flags = flags & ~MASK;`

使用`&`， 任何位与1组合都得本身， 所以这条语句保持1号位不变， 改变其他各位。 另外， 使用`&`， 任何位与0组合都得0。 所以无1号位的初始值是什么， 都将其设置为0。

例如， 假设`flags`是00001111， `MASK`是10110110。 下面的表达式：

`flags & ~MASK`

即是： `(00001111) & ~(10110110) // 表达式`

其结果为： `(00001001) // 结果值`

`flags &= ~MASK;`

运算符优先级问题

类别	运算符	结合性
后缀	<code>() [] > . ++ --</code>	从左到右
一元	<code>+ - ! ~ ++ -- (type)* & sizeof</code>	从右到左
乘除	<code>* / %</code>	从左到右
加减	<code>+ -</code>	从左到右
移位	<code><< >></code>	从左到右
关系	<code>< <= > >=</code>	从左到右
相等	<code>== !=</code>	从左到右
位与 AND	<code>&</code>	从左到右
位异或 XOR	<code>^</code>	从左到右
位或 OR	<code> </code>	从左到右
逻辑与 AND	<code>&&</code>	从左到右
逻辑或 OR	<code> </code>	从左到右
条件	<code>?:</code>	从右到左
赋值	<code>= += -= *= /= %= >>= <<= &= ^= =</code>	从右到左
逗号	<code>,</code>	从左到右

- 1、后缀类别，优先级最高者，并不是真正意义上的运算符。
- 2、单目运算符的优先级仅次于后缀类别
- 3、双目运算符比单目运算符要低
- 4、双目运算符中算术运算符>移位>关系>逻辑>条件>赋值

*任何一个逻辑运算符的优先级低于任何一个关系运算符
 *移位运算符的优先级比算术运算符要低，比关系要高

求值顺序

`a >b && b > c || b > d` 相当于 `((a > b) && (b > c)) || (b > d)`

尽管对于该例没必要使用圆括号，但是许多程序员更喜欢使用带圆括号的第 2 种写法。这样做即使不记得逻辑运算符的优先级，表达式的含义也很清楚

C 通常不保证先对复杂表达式中哪部分求值。例如，下面的语句，可能先对表达式 `5 + 3` 求值，也可能先对表达式 `9 + 6` 求值：

```
apples = (5 + 3) * (9 + 6);
```

C 把先计算哪部分的决定权留给编译器的设计者，以便针对特定系统优化设计。但是，对于逻辑运算符是个例外，C 保证逻辑表达式的求值顺序是从左往右。`&&` 和 `||` 运算符都是序列点，所以程序在从一个运算对象执行到下一个运算对象之前，所有的副作用都会生效。而且，C 保证一旦发现某个元素让整个表达式无效，便立即停止求值。正是由于有这些规定，才能写出这样结构的代码：

```
while ((c = getchar()) != ' ' && c != '\n')
```

`x != 0 && (20 / x) < 5` 只有当 `x` 不等于 0 时，才会对第 2 个表达式求值

```
if (range >= 90 && range <= 100)  
    printf("Good show!\n");
```

千万不要模仿数学上的写法：

```
if (90 <= range <= 100) // 千万不要这样写！
```

这样写的问题是代码有语义错误，而不是语法错误，所以编译器不会捕获这样的问题（虽然可能会给出警告）。由于 `<=` 运算符的求值顺序是从左往右，所以编译器把测试表达式解释为 `(90 <= range) <= 100`

课后练习题

1、指出下列常量的类型和含义

- a.\b'
- b.1066
- c.99.44
- d.0XAA

2、编写一个程序，要求提示输入一个ASCII码值，然后打印输入字符？

3、编写一个程序，发出一声警报。

4、

```
#include <stdio.h>
int main(void)
{
    int a, b;
    a = 5;
    b = 2; /* 第7行 */
    b = a; /* 第8行 */
    a = b; /* 第9行 */
    printf("%d %d\n", b, a);

    return 0;
}
```

程序在执行7、8、9行后，a、b、的值分别是多少

5、'X' 和"X" 分别占用多大的内存空间

课后练习题



1.6编写一个程序，以月/日/年的格式接受用户录入的日期信息，并以年月日的格式将其显示出出来。



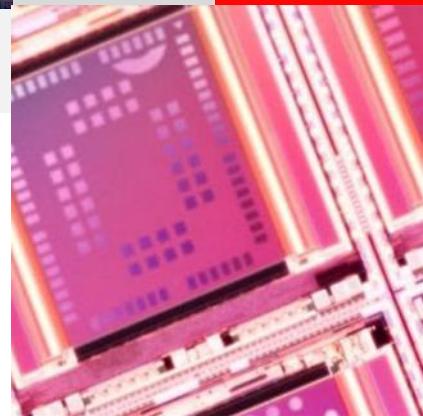
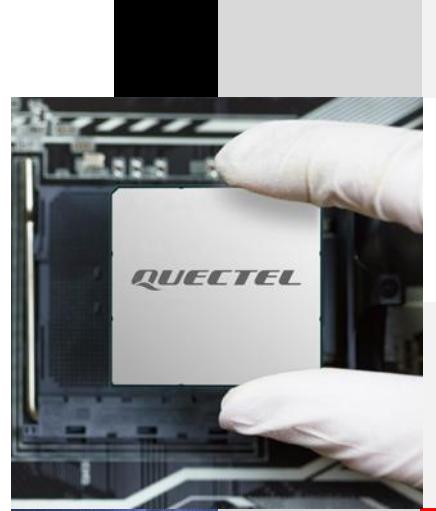
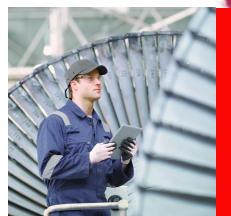
第二课 控制语句、函数、结构体 、枚举、联合

Build a Smarter World

www.quectel.com

izzul.irfan@quectel.com
Thu Oct 03 09:21:00 CST 2024

izzul.irfan@quectel.com
Thu Oct 03 09:21:00 CST 2024



C控制语句



循环类型	描述
<u>while</u> 循环	当给定条件为真时，重复语句或语句组。它会在执行循环主体之前测试条件。
<u>for</u> 循环	多次执行一个语句序列，简化管理循环变量的代码。
<u>do...while</u> 循环	除了它是在循环主体结尾测试条件外，其他与 while 语句类似。
<u>嵌套循环</u>	您可以在 while、for 或 do..while 循环内使用一个或多个循环。
控制语句	描述
<u>break</u> 语句	终止循环或 switch 语句，程序流将继续执行紧接着循环或 switch 的下一条语句。
<u>continue</u> 语句	告诉一个循环体立刻停止本次循环迭代，重新开始下次循环迭代。
<u>goto</u> 语句	将控制转移到被标记的语句。但是不建议在程序中使用 goto 语句。

for语句使用3个表达式控制循环过程，分别用分号隔开。initialize表达式在执行for语句之前只执行一次；然后对test表达式求值，如果表达式为真（或非零），执行循环一次；接着对update表达式求值，并再次检查test表达式。for语句是一种入口条件循环，即在执行循环之前就决定了是否执行循环。因此，for循环可能一次都不执行。statement部分可以是一条简单语句或复合语句。

形式：

```
for ( initialize; test; update )  
statement
```

在test为假或0之前，重复执行statement部分。

while(expression)
statement

statement部分可以是以分号结尾的简单语句，也可以是用花括号括起来的复合语句。

- 1、所有的非零值都视为真，只有0被视为假
- 2、= 不等 ==

```
int n = 3;  
n = -3;  
while (n)  
    printf("%2d is true\n", n++);
```

do while语句创建一个循环，在expression为假或0之前重复执行循环体中的内容。do while语句是一种出口条件循环，即在执行完循环体后才根据测试条件决定是否再次执行循环。因此，该循环至少必须执行一次。
statement部分可是一条简单语句或复合语句。

形式：

```
do  
statement  
while ( expression );
```

在test为假或0之前，重复执行statement部分。

C控制语句

语句	描述
<u>if</u> 语句	一个 if 语句 由一个布尔表达式后跟一个或多个语句组成。
<u>if...else</u> 语句	一个 if 语句 后可跟一个可选的 else 语句, else 语句在布尔表达式为假时执行。
<u>嵌套 if</u> 语句	您可以在一个 if 或 else if 语句内使用另一个 if 或 else if 语句。
<u>switch</u> 语句	一个 switch 语句允许测试一个变量等于多个值时的情况。
<u>嵌套 switch</u> 语句	您可以在一个 switch 语句内使用另一个 switch 语句。

expression1 ? expression2 : expression3

如果 expression1 为真（非 0）, 那么整个条件表达式的值与 expression2 的值相同; 如果 expression1 为假（0）, 那么整个条件表达式的值与 expression3 的值相同。

需要把两个值中的一个赋给变量时, 就可以用条件表达式。典型的例子是, 把两个值中的最大值赋给变量: $\text{max} = (\text{a} > \text{b}) ? \text{a} : \text{b};$

3种循环都可以使用 continue 语句。执行到该语句时, 会跳过本次迭代的剩余部分, 并开始下一轮迭代。如果 continue 语句在嵌套循环内, 则只会影响包含该语句的内层循环。

```

31 if (number > 6)
32     if(number < 12)
33         printf("You're close!\n");
34     else
35         printf("Sorry, you lose a turn!\n");
...

```

```
switch(color)
{
```

```
case 1: printf ("red");
case 2: printf ("yellow");
case 3: printf ("blue");
```

函数



函数是一组一起执行一个任务的语句。每个 C 程序都至少有一个函数，即主函数 **main()**，所有简单的程序都可以定义其他额外的函数。您可以把代码划分到不同的函数中。如何划分代码到不同的函数中是由您来决定的，但在逻辑上，划分通常是根据每个函数执行一个特定的任务来进行的。**函数声明**告诉编译器函数的名称、返回类型和参数。**函数定义**提供了函数的实际主体。

定义函数

C 语言中的函数定义的一般形式如下：

```
return_type function_name( parameter list )
{
    body of the function
}
```

在 C 语言中，函数由一个函数头和一个函数主体组成。下面列出一个函数的所有组成部分：

- **返回类型**：一个函数可以返回一个值。`return_type` 是函数返回的值的数据类型。有些函数执行所需的操作而不返回值，在这种情况下，`return_type` 是关键字 `void`。
- **函数名称**：这是函数的实际名称。函数名和参数列表一起构成了函数签名。
- **参数**：参数就像是占位符。当函数被调用时，您向参数传递一个值，这个值被称为实际参数。参数列表包括函数参数的类型、顺序、数量。参数是可选的，也就是说，函数可能不包含参数。
- **函数主体**：函数主体包含一组定义函数执行任务的语句。

函数

函数声明

函数声明会告诉编译器函数名称及如何调用函数。函数的实际主体可以单独定义。

函数声明包括以下几个部分：

```
return_type function_name( parameter list );
```

针对上面定义的函数 max(), 以下是函数声明：

```
int max(int num1, int num2);
```

void show_n_char(char ch, int num); void show_n_char(char, int);

在原型中使用变量名并没有实际创建变量， char仅代表了一个char类型的变量，

函数的返回类型：

函数的返回类型指的是函数返回值的类型。如果返回值的类型与声明的返回类型不匹配， 返回值将被转换成函数声明的返回类型。

函数



在函数调用时即使函数不带参数，也应该包括参数列表，因此如果f是一个函数，
F();
是一个函数调用语句，而f; 这个语句计算函数f的地址，并不是调用函数。

实际参数是出现在函数调用圆括号中的表达式。形式参数是函数定义的函数头中声明的变量。调用函数时，创建了声明为形式参数的变量并初始化为实际参数的求值结果。

如何更改主调函数中的变量？

形式参数出现在函数定义中，它以假名字来表示函数调用时需要提供的值；实际参数是出现在函数调用中的表达式。

C语言中，**实际参数是通过值传递的：函数调用时，计算出每个实际参数的值并且把他赋值给相应的形式参数**。在函数执行过程中，对形式参数的改变不会影响实际参数的值

```
2 #include <stdlib.h>
3
4 int imin(int, int);
5
6 int main(void)
7 {
8     int evil1, evil2;
9     printf("Enter a pair of integers (q to quit):\n");
10    while (scanf("%d %d", &evil1, &evil2) == 2)
11    {
12        printf("The lesser of %d and %d is %d.\n",
13               evil1, evil2, imin(evil1, evil2));
14        printf("Enter a pair of integers (q to quit):\n");
15    }
16    printf("Bye.\n");
17    return 0;
18 }
19
20 int imin(int n, int m)
21 {
22     int min;
23     if (n < m)
24         min = n;
25     else
26         min = m;
27
28     return min;
29 }
```

函数的几个要点



- 1、函数声明与定义的三要素 返回值， 函数名称， 函数参数；
- 2、函数传参是值的传递，理解函数是传值，而不是传变量本身；**参数传递为值传递；**
- 3、函数的形参在什么时候创建实体保存在哪里？函数的局部变量的作用域为什么仅限于函数体内？
- 4、函数调用如何实现栈的切换和CPU寄存器现场的保存？
- 5、return如何实现？

结构变量—定义声明

数组是所有具有相同类型元素的集合。结构体具有的特性与数组很不相同，结构的元素(成员)可能具有不同的类型。

每个结构成员都有名字，在其他一些语言中，也把结构的成员成为字段。

结构声明

```
struct book
{
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};
```

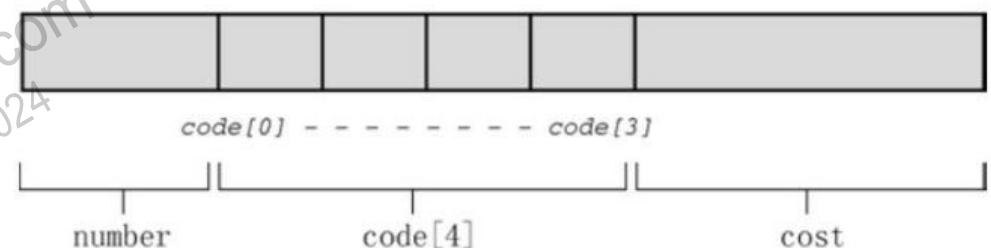
结构定义

```
struct book library;
```

编译器执行这行代码便创建了一个结构变量library。编译器使用book模板为该变量分配空间：一个内含MAXTITL个元素的char数组、一个内含MAXAUTL个元素的char数组和一个float类型的变量。

```
20 struct book
21 {
22     char title[MAXTITL];
23     char author[MAXAUTL];
24     float value;
25 } library; /* 声明的右花括号后跟变量名 */
26
27
28 struct
29 {
    /* 无结构标记 */
30     char title[MAXTITL];
31     char author[MAXAUTL];
32     float value;
33 } library;
```

```
struct stuff {
    int number;
    char code[4];
    float cost;
};
```



结构变量—初始化

结构体初始化

```
struct book surprise =
{
    .value = 10.99
};

struct book gift =
{
    .value = 25.99,
    .author = "James Broadfool",
    .title = "Rue for the Toad"
};
```

```
struct
{
    int number;
    char name[NAME_LEN];
    int on_hand;
}part1={528,"Disk driver",10};
```

528	number
Disk drive	name
10	on_hand

```
struct book gift=
{
    .value = 18.90,
    .author = "Phlionna Pestle",
    .on_hand = 0.25
};
```

对特定成员的最后一次赋值才是它实际获得的值

是否可以使用== 来比较两个结构是否相等?
可以使用方法?

结构体初始化遵循的原则类似于数组初始化原则。用于结构初始化的表达式必须是常量，例如不能用变量来初始化成员on_band。但是C99标准中放宽了这一限制

结构变量—成员引用

访问结构成员----使用结构成员运算符(.)访问结构中的成员

成员变量的性质和其他同类型的变量一致；

例如， library.value即访问library的value部分。 可以像使用任何float类型变量那样使用library.value。

虽然library是一个结构， 但是library.value是一个float类型的变量， 可以像使用其他 float 类型变量那样使用它。 例如，
scanf("%f",...)需要一个 float 类型变量的地址， 而&library.float正好符合要求。 .比&的优先级高，因此这个表达式和
&(library.float)一样。

一旦创建了标记part， 就可以利用他来声明变量 struct part part1,part2;
还可以用typedef来定义真实的类型名。 Part part1,part2;

```
typedef struct{
    int number;
    char name[NAME_LEN+1];
    int on_hand;
    int on_hand;■
}Part;
```

结构变量—内存对齐

理论上讲结构体的各个成员在内存中是连续存储的，和数组非常类似。

name	num	age	group	score
------	-----	-----	-------	-------

理论上这个结构体应该占用， $4+4+4+1+4=17\text{byte}$

但是由于编译器的实现，结构体的内存需要对齐。

name	num	age	group		score
------	-----	-----	-------	--	-------

则需要占用 $4+4+4+4+4=20\text{ byte}$ ；

由于CPU需要从内存上取值；

但是由于访问的限制，不在内存对齐位置上的数据，可能出现两次访问内存，才能完整的取值到；

而如何对齐的内存，只需要访问一次；

这样的做法也是典型的空间换时间；

#pragma pack() 这个预处理指令，可以改变我们的默认对齐数。

sizeof 可以求出结构体占用的内存大小

```
struct{ //没有写 stu
    char *name; //姓名
    int num; //学号
    int age; //年龄
    char group; //所在学习小组
    float score; //成绩
} stu1, stu2;
```

结构变量—函数返回结构体

函数的返回值可以是结构体变量，函数的形参也可以是结构体变量。

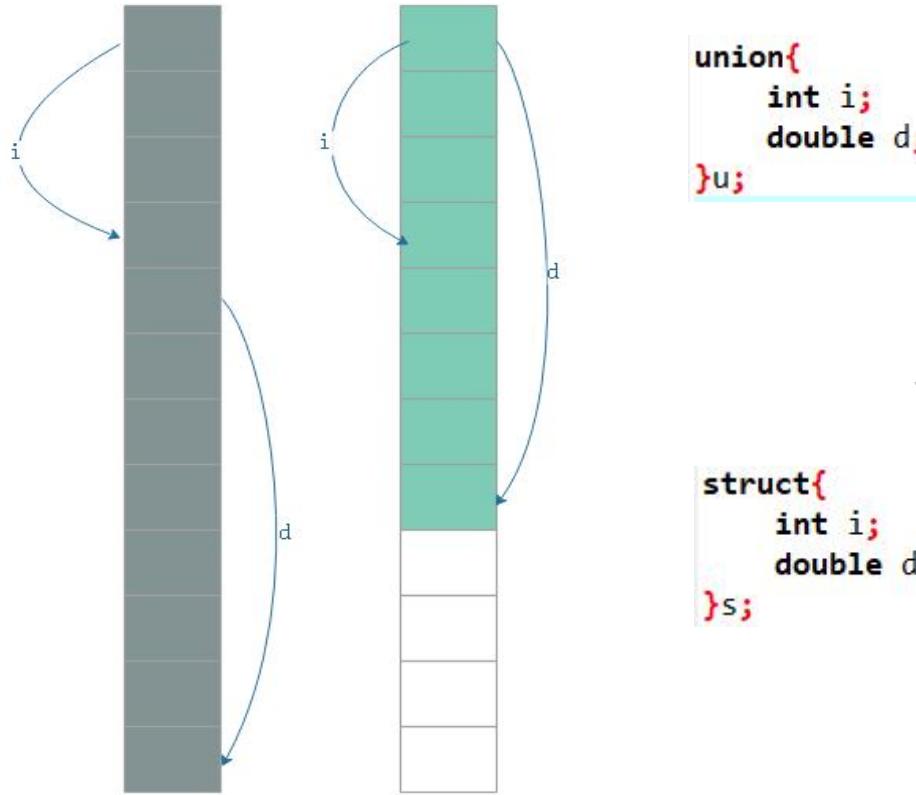
```
#include "stdio.h"
#include "string.h"
struct ABC{
    char name[20];
    int n;
};
struct ABC myfun(void){
    struct ABC x={"Lining", 99}; // 声明一个结构体局部变量x并初始化
    return x; // 返回局部变量结构体x
}
int test_struct_fun(void){
    struct ABC y=myfun(); // 声明一个同类型结构体变量y并将函数返回值赋给它
    printf("%s %d\n", y.name, y.n); // 打出来看看
    return 0;
}
```

联合

联合（union）是一种数据类型，它能在同一个内存空间中储存不同的数据类型（不是同时储存）。其也是由一个或多个成员构成。编译器只为联合中最大的成员分配足够的内存空间。

联合的定义方式与结构是一样的，只是把关键字 struct 改成 union

假设int类型的值要占用4个字节内存,而double类型的值占用8个字节



联合在声明和初始化上与结构体类似。
使用联合来构造混合的数据结构

数组中每个元素都是Number联合,Number联合既可以存储int类型的值又可以存double类型的值。
number_array[0].i = 5;
number_array[0].d=3.14

结构体和联合共同使用?
使用结构体成员标记联合

```

typedef union{
    int i;
    double d;
}Number;

Number number_array[1000];

```

```

typedef struct{
    int kind;
    union{
        int i;
        double d;
    }u;
};

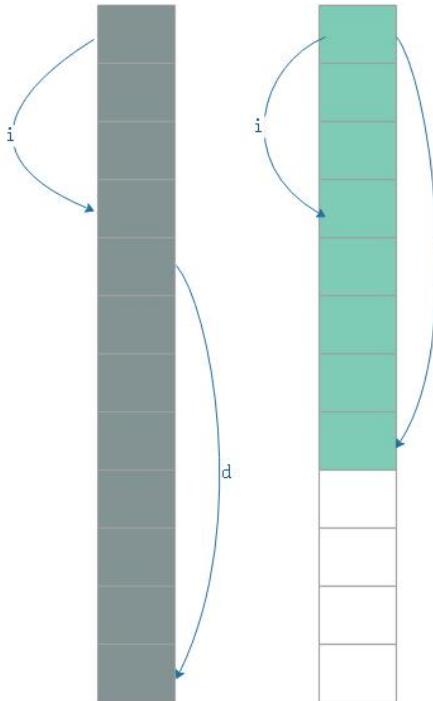
```

联合



获取联合成员的方式和获取结构成员的方式一样。唯一的差异在于，当改变一个联合成员的值时，实际上修改了该联合所有成员的值。

- 1、union中可以定义多个成员，union的大小由最大的成员的大小决定。
- 2、union成员共享同一块大小的内存，一次只能使用其中的一个成员。
- 3、对某一个成员赋值，会覆盖其他成员的值。
- 4、联合体union的存放顺序是所有成员都从低地址开始存放的。



机器大小端

这是因为在计算机系统中我们是以字节为单位的，每个地址单元都对应着一个字节，一个字节为8bit。但在c语言中除了8bit的char之外，还有16bit的short型，32bit的long型（要看具体的编译器），另外，对于位数大于8位的处理器，例如16位或者32位处理器，由于寄存器宽度大于一个字节，那么必然存在着一个如何将多个字节安排的问题。因此就有了大端存储和小端存储模式。

例如一个16bit的short型x，在内存中的地址为0x0010，x的值为0x1122，那么0x11位高字节，0x22为低字节。

对于大端模式，就将0x11放在低地址中，即0x0010中，0x22放在高地址中，即0x001中。小端模式，刚好相反。

我们常用的x86结构是小端模式，而KELL C51则为大端模式。很多的ARM,DSP都为小端模式。

地址	小端模式	大端模式
0x7E000000	0x22	0x11
0x7E000001	0x11	0x22

```

union U
{
    char c;
    int i;
}u;//联合体变量创建方法类比结构体

int main()
{
    u.i = 1;
    //0x 00 00 00 01
    //低地址----->高地址
    //01 00 00 00 小端存储 低位放低地址 ("小低低")
    //00 00 00 01 大端存储 低位放高地址
    if (u.c == 1)
    {
        printf("小端");
    }
    else
    {
        printf("大端");
    }
}

```

枚举



在程序中，我们需要变量只具有少量有意义的值，如布尔类型的变量应该只有2中可能的值：真或者假。

可以用枚举类型（enumerated type）声明符号名称来表示整型常量。使用enum关键字，可以创建一个新“类型”并指定它可具有的值（实际上，**enum常量是int类型**，因此，只要能使用int类型的地方就可以使用枚举类型）。枚举类型的是提高程序的可读性。它的语法与结构的语法相同。

```
enum spectrum {  
    red,  
    orange,  
    yellow,  
    green,  
    blue,  
    violet  
};  
  
enum spectrum color;  
  
int i;  
i = blue; // i的值为4  
color = 0;  
i = color+2;
```

编译器会把color作为整数变量来处理，而red/orange只是数0,1,2..的名字
虽然把枚举的值作为整数使用非常方便，但是把整数用作枚举的值却是非常危险的。
例如，你把6赋值给color，而6不能对应任何颜色

typedef

typedef 工具是一个高级的数据特性,利用typedef可以为某一类型自定义名称。
typedef由编译器解释, 不是预处理器。

假设要用BYTE表示1字节的数组。 只需像定义个char类型变量一样定义BYTE, 然后在定义前面加上关键字typedef即可:

```
typedef unsigned char BYTE;
```

随后, 便可使用BYTE来定义变量:

```
BYTE x, y[10], * z;
```

```
typedef char * STRING;
```

没有typedef关键字, 编译器将把STRING识别为一个指向char的指针变量。有了typedef关键字, 编译器则把STRING解释成一个类型的标识符, 该类型是指向char的指针。因此:

```
STRING name, sign;
```

相当于:

```
char * name, * sign;
```

课后作业



- 2.1、**编写一个程序，从标准输入读取C源代码，并验证所有的花括号都正确地成对出现（{}）。不用关心注释、字符串常量内部形式的花括号。
- 2.2、**编写一个程序，提示用户输入一个数n，然后显示出1~n的所有偶数平方值
- 2.3、**打印一个99乘法表
- 2.4、**编程实现用户输入一个数值，把这个数值的二进制位模式从左到右变换一下
例如：
在32位机器上，内存上：00000000 00000000 00000000 00011001
 10011000 00000000 00000000 00000000
注意编写函数时不要让他依赖于机器上整型值的长度
- 2.5**输入某年某月某日，判断这一天是这一年的第几天？

课后作业



2.6、

编写函数，使得函数返回下列的值(假设a和n是形式参数，其中a是int类型数组，而n则是数组的长度)

- a) 数组a中最大的值
- b) 数组a中所有元素的平均值
- c) 数组a中正数元素的数量



第三课 数组、指针

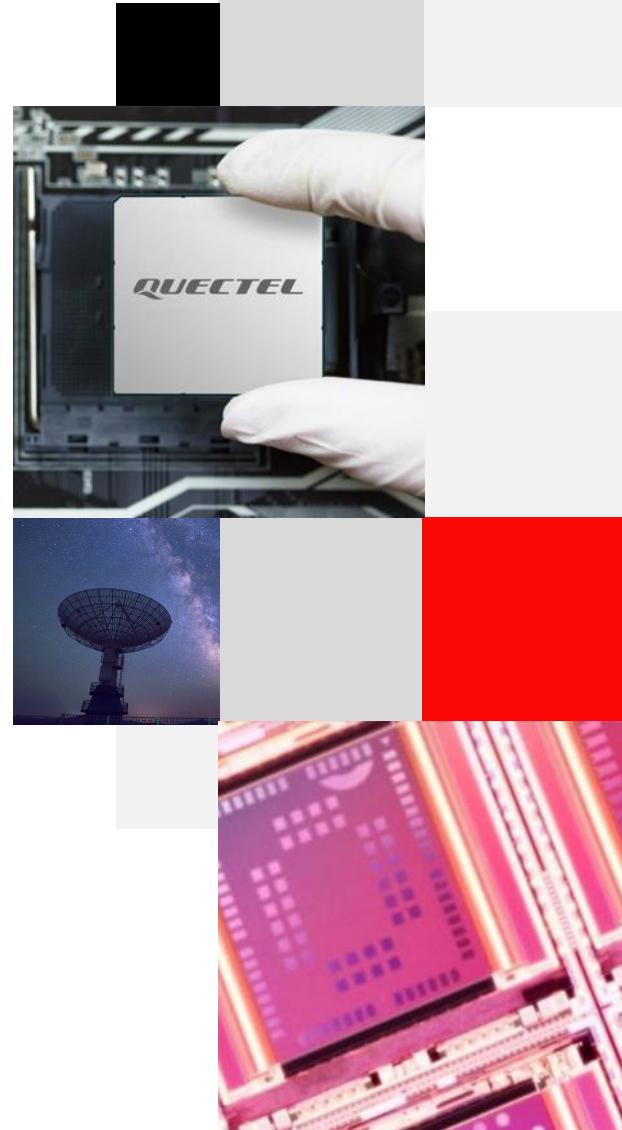
Build a Smarter World

www.quectel.com

izzul.irfan@quectel.com
Thu Oct 03 09:21:00 CST 2024

izzul.irfan@quectel.com
Thu Oct 03 09:21:00 CST 2024

izzul.irfan@quectel.com
Thu Oct 03 09:21:00 CST 2024



数组



数组由数据类型相同的一系列元素组成。 需要使用数组时，通过声明数组告诉编译器数组中内含多少元素和这些元素的类型。 编译器根据这些信息正确地创建数组。普通变量可以使用的类型， 数组元素都可以用。可以根据元素在数组中所处的位置把它们一个个地选出来。

如定义一个含有10个int类型变量的数组， int a[10]; char b[10];//定义一个含有10个char类型的变量

为了存取特定的数组元素，可以在写数组名的同时在后边加上一个用方括号围绕的整数值(称这是对数组取下标或进行索引)。数组元素始终从0开始，所以长度为n的数组元素的索引是从0到n-1

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
------	------	------	------	------	------	------	------	------	------

数组初始化

1、

```
int powers[8] = {1, 2, 4, 6, 8, 16, 32, 64};
```

2、当初始化列表中的值少于数组元素个数时，编译器会把剩余的元素都初始化为0。也可能是垃圾数据

```
#define SIZE 4  
int some_data[SIZE]={1492, 1066};
```

3、如果初始化列表的项数多于数组元素个数，编译器可没那么仁慈，它会毫不留情地将其视为错误。但是，没必要因此嘲笑编译器。其实，可以省略方括号中的数字，让编译器自动匹配数组大小和初始化列表中的项数

```
const int days[]={31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

4、C99新增指定初始化

第一，如果指定初始化器后面有更多的值，如该例中的初始化列表中的片段：[4]=31,30,31，那么后面这些值将被用于初始化指定元素后面的元素。也就是说，在days[4]被初始化为31后，days[5]和days[6]将分别被初始化为30和31。第二，如果再次初始化指定的元素，那么最后的初始化将会取代之前的初始化。初始化列表开始时把days[1]初始化为28，但是days[1]又被后面的指定初始化[1]=29初始化为29

```
int days[]={31, 28, [4]=31, 30, 31, [1]=29};
```

int stuff[] = {1, [6]=23}; //会发生什么？

int staff[] = {1, [6]=4, 9, 10}; //会发生什么？

编译器会把数组的大小设置为足够装得下初始化的值。所以，stuff数组有7个元素，编号为0~6；而staff数组的元素比stuff数组多两个（即有9个元素）。

数组



C不要求检查下标的范围。但是访问超出范围时，程序可能执行不可预知的行为。对于某些编译器来说，下面的for语句可能产生超过10次的循环。（从内存排列的角度思考可能存在的原因）

常见的数组越界访问----在编译的时候，编译可能会发出警告信息

```
int test_shuzu(void)
{
    int ar[10], n=0;

    for(n=1; n<=10; n++)
    {
        ar[n]=0;
    }

    return 0;
}
```

多维数组

如果一个数组的维数不为1，一般称这个数组为多维数组；下面是以二维数组（数学上的术语称为矩阵）为例。

定义一个二维数组：int arr2[3][3]；

```
int arr2[3][3]={1,2,3,4,5,6,7,8,9};
```

这个数组包含9个元素。

数组的行和列的下表都是从0开始索引的，为了访问 i 行 j 列的元素，需要写成 $\text{arr2}[i][j]$ ；表达式 $\text{arr2}[i]$ 指明了 arr2 数组的第 i 行，而 $\text{arr2}[i][j]$ 则选择了此行中第 j 个元素。

可以通过 $\text{arr2}[1][2]=100$ ；//将数组中第5个元素的值赋值为100；

思考二维数组在内存中的存放形式？

多维数组

可以通过嵌套一维数组初始化的方法初始化二维数组；

```
int m[2][3]={  
    {1,2,3},  
    {4,5,6}  
};
```

这里也可以将内层的花括号省略掉。

*如果初始化没有大到足以填满整个多维数组，那么数组中剩余的元素赋值为0，或者无效的值。例如：

```
int mm[5][3]={  
    {1,2,3},  
    {4,5,6},  
    {7,8,9}  
};
```

```
int mm[5][3]={  
    {1,2},  
    {4,5,6},  
    {7,9}  
};
```

字符串数组



C语言允许省略数组的行数(因为在初始化式中元素的数量求出),但是C语言明确要求指明列数。
并非所有的字符串都足矣填满数组的一整行, C语言用空字符来填补。这样就导致内存空间的浪费。

可以定义 `char planets[][][8]={"Mercury","Venus","Earth","Mars","Jupiter","Saturn","Uranus","Neptune","Pluto"}`

	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	\0	\0
2	E	a	r	t	h	\0	\0	\0
3	M	a	r	s	\0	\0	\0	\0
4	J	u	p	i	t	e	r	\0
5	S	a	t	u	r	n	\0	\0
6	U	r	a	n	u	s	\0	\0
7	N	e	p	t	u	n	e	\0
8	P	l	u	t	o	\0	\0	\0

sizeof



sizeof 可以计算指定类型值所需要的存储空间的大小

表达式 sizeof(类型名);

在32位机器上,表达式sizeof(int) 的值通常为4。

sizeof 可以用于常量、变量和表达式。如果i和j 是整形变量, 那么sizeof(i) //表达式的值是4,
这和sizeof(i+j)

sizeof i

sizeof i+j // (sizeof i)+j

对数组使用sizeof运算符

int a[10];

如果数组a有10个整数,那么sizeof(a) 通常为40;

计算数组的个数: sizeof(a)/sizeof(a[0])

注意sizeof() 不是函数; 其表达式的值, 在编译完成后就可以得出

const数组

代码中的const告诉编译器，该函数不能修改ar指向的数组中的内容。如果在函数中不小心使用类似ar[i]++的表达式，编译器会捕获这个错误，并生成一条错误信息。

```
int sum(const int ar[], int n) /* 函数定义 */  
{  
    int i;  
    int total = 0;  
    for( i = 0; i < n; i++)  
        total += ar[i];  
    ar[i]++; //编译器会出现报错  
  
    return total;  
}
```

指针



从根本上看，指针（pointer）是一个值为内存地址的变量（或数据对象）。

指针本身是一个4字节变量、其中保存的是要指向的目标内存地址的值，指针的类型是对目标内存地址的格式化。

```
int i = 0x11223344;  
  
int *ptr = (int *)(&i);  
printf("0x%x", *ptr)//输出0x11223344  
  
char *c = (char *) (0x7E00001);  
printf("0x%x", *c)//输出0x44  
  
int * pi; // pi是指向int类型变量的指针  
  
char * pc; // pc是指向char类型变量的指针  
  
float * pf, * pg; // pf、pg都是指向float类型变量的指针  
  
%p 打印指针本身的值
```

地址编码	0x44	0x33	0x22	0x11	i
0x7E00001					
0x7E00005					
0x7E00009					
0x7E00013					
0x7E00070	0x01	0x00	0x00	0x7E	c
0x7E00080	0x01	0x00	0x00	0x7e	ptr

指针与函数

编写程序时，可以认为变量有两个属性：名称和值（还有其他性质，如类型，暂不讨论）。计算机编译和加载程序后，认为变量也有两个属性：地址和值。地址就是变量在计算机内部的名称。在许多语言中，地址都归计算机管，对程序员隐藏。然而在 C 中，可以通过&运算符访问地址，通过*运算符获得地址上的值。

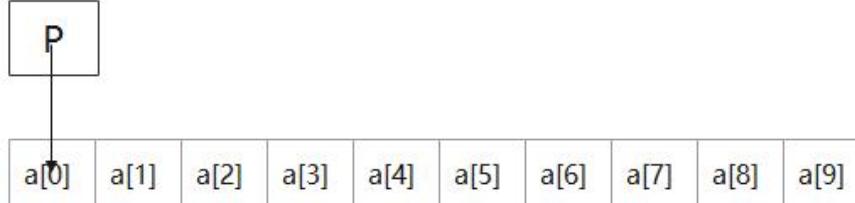
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void interchange(int * u, int * v);
5 int main(void)
6 {
7     int x = 5, y = 10;
8
9     printf("Originally x = %d and y = %d.\n", x, y);
10    interchange(&x, &y); // 把地址发送给函数
11
12    printf("Now x = %d and y = %d.\n", x, y);
13
14    return 0;
15 }
16
17 void interchange(int * u, int * v)
18 {
19     int temp;
20     temp = *u; // temp获得 u 所指向对象的值
21     *u = *v;
22     *v = temp;
23 }
```

该函数传递的不是x和y的值，而是它们的地址。这意味着出现在interchange()原型和定义中的形式参数u和v将把地址作为它们的值。因此，应把它们声明为指针。

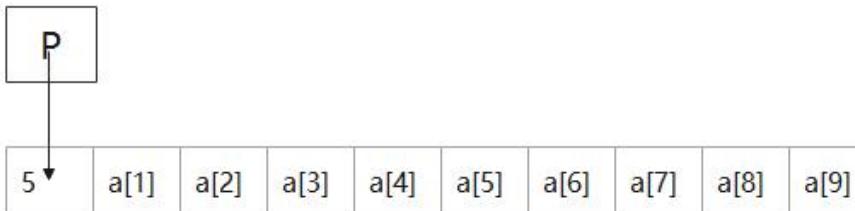
指针和数组

指针可以指向数组的元素。

```
int a[10], *p; p = &a[0];
```



```
*p=5;
```



这样完成了对数组a第一个元素的赋值。同样通过在 P 上执行指针算术运算可以访问数组a的其他所有元素。

```
int *p, *q, i;
```

指针加上整数

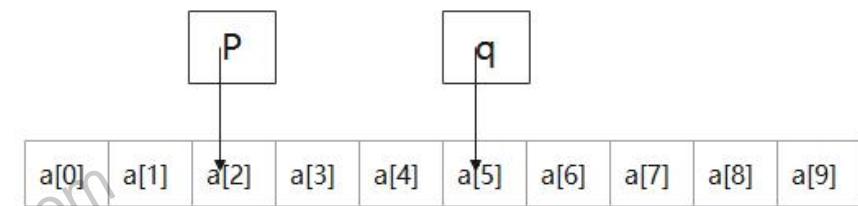
- 1、指针加上整数
- 2、指针减去整数
- 3、两个指针相减

指针加1：指针类型加上内存地址的图

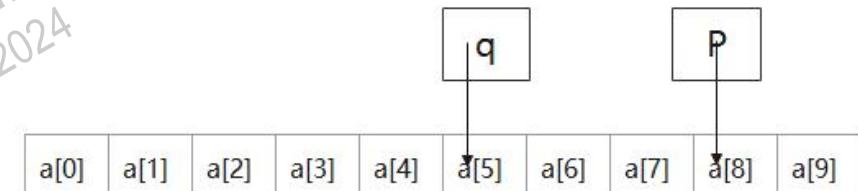
```
P=&a[2];
```



```
q=p+3;
```



```
p+=6;
```



字符串指针

在学习数组的时候，当使用二维数组存储字符串的时候，会造成内存空间浪费；

可以是用指针数组，存储长度不一致的数组；

可以定义 `char *planets[]={"Mercury","Venus","Earth","Mars","Jupiter","Saturn","Uranus","Neptune","Pluto"}`



Planets的每一个元素都是指向以空字符结尾的字符串的指针。虽然必须为planets数组中的指针分配空间，但是字符串中不再有任何浪费的字符。

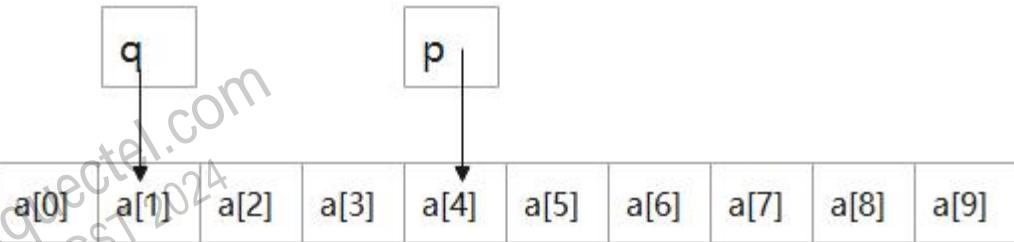
指针和数组

在一个不指向任何数组元素的指针上执行算术运算会导致未定义的行为。
此外，只有在两个指针指向同一个数组时，把他们相减才有意义

当两个指针相减时，结果为指针之间的距离。
如果p指向a[i]且q指向a[j],那么p-q就等于i-j。

```
p=&a[5];  
q=&a[1];  
  
i=p-q; /*i is 4*/  
i=q-p; /*i is -4*/
```

指向复合字面量
`int *p = (int []){3,0,3,4,1};`



指针用于处理数组
`#define N 10`
`...`
`int a[N],sum,*p;`
`...`
`sum = 0`
`for(p=&a[0];p<&a[N];p++)//尽管元素a[N]不存在,但是对它使用取地址运算符是合法的`
`{`
`sum +=*p;`
`}`

指针和数组

*运算符和++运算符的结合

把值存入一个数组元素中，然后进到下一个元素。利用数组下标可以这样写:a[i++] = j;

P指向数组元素，可以这样写*p++ = j;

因为后缀++的优先级高于*，所以可以把上面的语句看成*(p++) = j;

P++的值是p。(因为使用后缀++，所以p只有在表达式计算出来后才可以自增)，因此，*(p++)的值将是*p。

```
*p++ 或 *(p++) //自增前表达式的值是*p,以后在自增p  
(*p)++          //自增前表达式的值是*p, 以后再自增*p  
*++p或*(++p)   //先自增p,自增后表达式的值是*p  
++*p或++(*p)   //先自增*p, 自增后表达式的值是*p
```

最常用的是*p++

```
for(p=&a[0];p<&a[N];p++)  
{
```

```
    sum += *p;
```

```
}
```

改写成

```
P = &a[0];  
while(p<&a[N])
```

```
{
```

```
    sum += *p++;
```

```
}
```

指针和数组



指针的算术运算是数组和指针之间相互关联的一种方法，但这不是两者之间唯一的联系。

C中可以用数组的名字作为指向数组第一个元素的指针

```
int a[10];
*a=7;同样可以通过a+1 来修改a[1]; *(a+1)=12
a+i 等同于&a[i]; 且 *(a+i) = a[i];
```

```
for(p=a;p<a+N;p++)
{
    sum+=*p
}
```

虽然可以把数组名用作指针，但是不能给数组名赋新的值，`a++` //这样的操作不被允许

用指针作为数组名

```
#define N 100
...
int a[N], i, sum = 0, *p = a;
...
for (i = 0; i < N; i++)
    sum += p[i];
```

编译器把`p[i]`看作`*(p+i)`，这是指针算术运算非常正规的用法。

指针和数组

数组名在传递函数时，总是被视为指针。

在给函数传递普通变量时，变量值的会被复制；也任何对响应形式参数的改变都不会影响到变量。

但是，作为实际参数的数组可能被改变。

```
int a[10];
void chang_a(int a[],int n)
{
    int i;
    for(i=0;i<n;i++)
    {
        a[i]=0;
    }
}
```

如果需要可以把数组型形式参数声明为指针 `void chang_a(int *a,int n);`

对于形式参数而言，声明为数组跟声明为指针是一样；但是对变量而言，则不同；

`int a[10];`//会导致编译器10个整数的空间

和 `int *a;`//只会为一个指针变量分配空间

动态内存使用

C中语言的数据结构大小通常是固定的。例如数组，一旦程序完成编译的，数组元素的数量就是固定了，

C语言支持动态存储分配，那么在程序执行期间分配内存单元的能力。

虽然动态存储分配是用于所有类型的数据，但是主要用于字符串、数组和结构。

`malloc`函数-----分配内存块,但是不对内存块进行初始化

`calloc`函数-----分配内存块,并且对内存块进行清零

由于申请函数无法知道计划存贮在内存块中的数据是什么类型的，所以他不能返回int类型、char类型等普通类型的指针，而是返回`void *`类型的值，此类型的值是通用类型，本质上它只是内存地址。

当调用函数分配不到足够大的内存块的时候，改函数会返回空指针(`NULL`)---不指向任何地方的指针

注意，通过空指针访问内存是未定义的行为，程序可能会崩溃。

动态内存

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

char *p = malloc(n+1); // 分配N个字符的字符串的内存空间

在执行赋值操作时会把malloc函数返回的通用指针转化为char * 类型

通常情况下,可以把void *类型赋值给任何指针类型的变量,反之也可以。我们通常写成p = (char *)malloc(n+1);
为字符串分配内存空间时, 不要忘记包含空字符

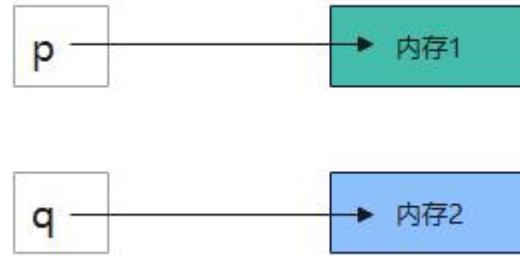
创建指针数组 char *p[5];

malloc函数所获得的内存块都来自一个成为堆的存储池, 因此这个池会被耗尽;

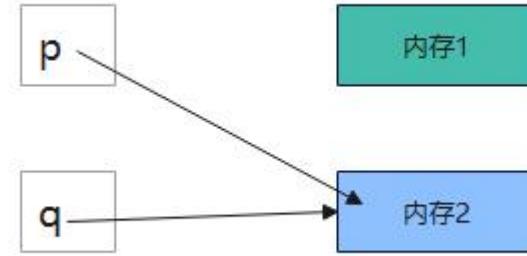
所以在申请的**内存块不使用的时候要释放** void free(void *ptr)

动态内存—内存泄漏

不再使用的内存要进行释放(free)



Thi
izzul.irfan@quectel.com
Thu Oct 03 09:21:00 CST 2024



假如p指向了一个内存块1, q指向了另一个内存块2;

```
char *p=malloc(20);  
char *q=malloc(20);
```

$p = q;$ //p和q 指向同一块内存，没有办法再去访问内存1，导致内存泄漏。

函数指针

指针可以指向各种类型的数据,包括变量、数组元素以及动态分配的内存块, 同样还允许指向函数。

函数占用内存单元, 所以每个函数都有地址, 就像每个变量都有地址一样。

指针作为函数的参数、指针作为函数的返回值

```
int hello(char *i); // 声明hello带有一个char *的形参, 且返回int类型的
```

```
int (*f)(int*); // 声明f 指向一个函数, 该函数有一个int的类型形参, 且返回值为int类型
```

```
f = hello; // 赋值
```

```
y=(*f)(&x); // y=f(&x); *f 表示f所指向的函数, 而&x是函数调用的实际参数
```

C中允许用f(&x)来调用f 指向的函数, 虽然f(&x)看上去自然, 但是(*f)(&x)的使用更易提醒读者f是指针不是函数名;

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

```
func fp = hello;
```

函数指针数组

```
void (*file_cmd[])(void)={new_cmd,open_cmd,close_cmd};
```

const类型指针

const标识符用来表示一个对象的不可变的性质，例如定义：

如果用const来修饰一个指针变量：

```
int a = 20;  
int b = 20;  
int * const p = &a;
```

这里的const用来修饰指针变量p，根据const的性质可以得出结论：p在定义为变量a
也就是说指针变量p中就只能存储变量a的地址0x11223344。如果在后面的代码中写p = &b;，编译时就会报错，因为p是不可
改变的，不能再被设置为变量b的地址。

但是，指针变量p所指向的那个变量a的值是可以改变的，即：*p = 21;这个语句是合法的，因为指针p的值没有改变(仍然是
变量c的地址0x11223344)，改变的是变量c中存储的值。

与下面的代码区分一下：

```
int a = 20;  
int b = 20;  
const int *p = &a;  
p = &b;
```



变长度的结构体

```
struct Packet
{
    int state;
    int len;
    char cData[0]; //这里的0长结构体就为变长结构体提供了非常好的支持
};
```

0长度的数组，也称作柔性数组。

在一个结构体的最后，声明一个长度为0的数组，就可以使得这个结构体是可变长的，对于编译器来说，此时长度为0的数组并不占用空间，因为数组名本身不占空间，它只是一个偏移量，数组名这个符号本身代表了一个不可修改的地址常量。
(数组名不是指针)

在结构体中，长度为0的数组必须在最后声明；

```
struct Packet
{
    int state;
    int len;
    char *p;
};
```

- 长度为0的数组并不占有内存空间，而指针方式需要占用内存空间。
- 对于长度为0数组，在申请内存空间时，采用一次性分配的原则进行；对于包含指针的结构体，在申请空间时需分别进行，释放时也需分别释放。
- 对于长度为0的数组的访问可采用数组方式进行

```
struct zero_buffer
{
    int len;
    char data[0];
};

if ((zbuffer = (struct zero_buffer *)malloc(sizeof(struct zero_buffer) + sizeof(char) * CURR_LENGTH)) != NULL)
{
    zbuffer->len = CURR_LENGTH;
    memcpy(zbuffer->data, "Hello World", CURR_LENGTH);

    printf("%d, %s\n", zbuffer->len, zbuffer->data);
}

free(zbuffer);
zbuffer = NULL;
```

课后作业



3.1

编写一个函数当传递两个变量的地址时，函数交换两个变量的值

3.2

编写函数，实现查找一个数组中的最大值和第二大的值，并保存在由形参指向的地址中

```
Void find_two_largest(int *a,int n,int *largest,int *second_lar);
```

3.3

编写函数实现把参数字符串中字符反向排列，注意使用指针而不是数组下标，不要使用任何C函数库中的用于操纵字符串的函数

```
void reserse_string(char *string);
```

3.4 编写函数copy_n

```
void copy_n(char dst[],char src[],int n);
```

这个函数用于把一个字符串从数组src复制到数组dst。

3.5

使用共用体判断一个机器的大小端模式



第四课 预处理、编译流程、内存段、栈帧

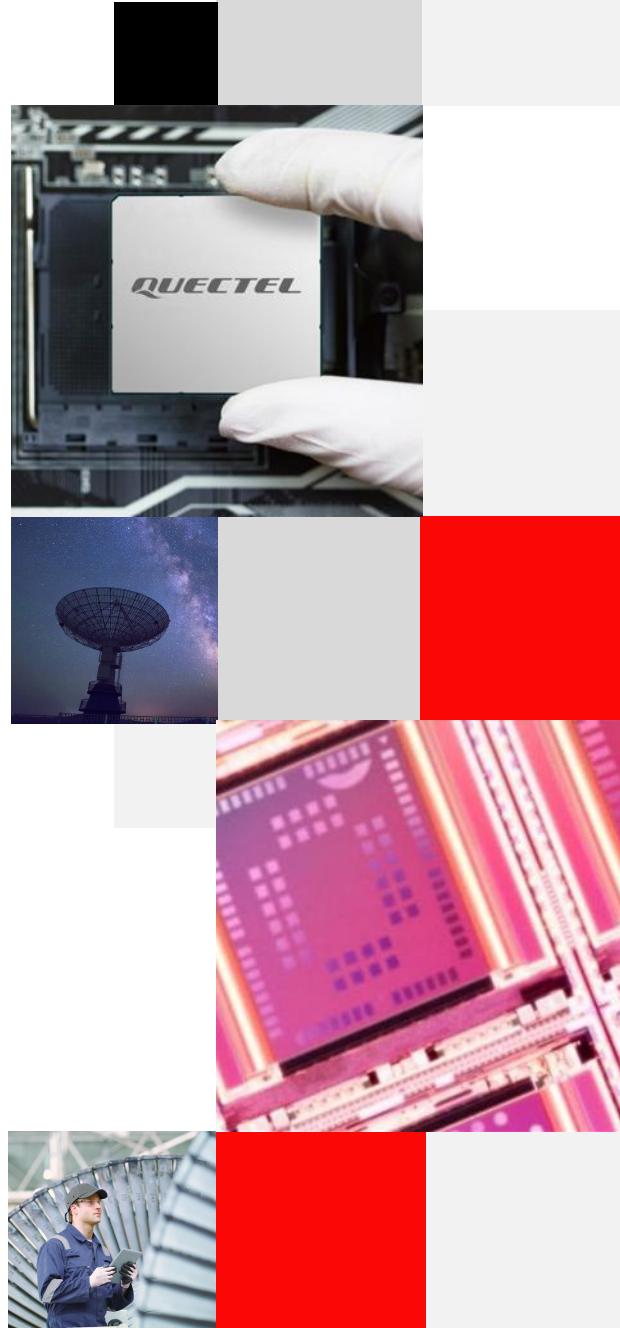
Build a Smarter World

www.quectel.com

izzul.irfan@quectel.com
Thu Oct 03 09:21:00 CST 2024

izzul.irfan@quectel.com
Thu Oct 03 09:21:00 CST 2024

izzul.irfan@quectel.com
Thu Oct 03 09:21:00 CST 2024



预处理器



预处理器的行为是由预处理指令(由#字符开头的一些命令)控制的。如#define #include
#define指令定一个宏----用来代表其他东西的一个名字
#include指令告诉预处理器打开一个特定的文件，将它的内容作为正在编译的文件的一部分

宏定义:#define #undef

文件包含:#include

条件编译:#if/#ifdef/#ifndef/#elif/#else/#endif

//插入任意数量的空格或水平制表符

```
# define N 1000
```

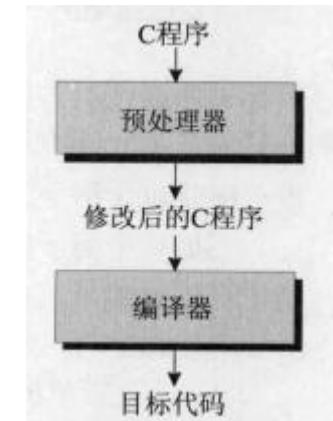
```
#define DISK ( hello* \
             world )
```

不要放置任何额外的符号

```
#define N = 100
```

```
Int a[N];//int a[N = 100]
```

```
#define MAX(x,y) ((x)>(y)?(x):(y))
```



预处理器



运算符和##运算符，在预处理阶段被执行

```
#define MY_PRINTF(n) printf(#n " = %d\n",n);
```

```
MY_PRINTF(i/j); //printf("i/j" " = %d\n",i/j);
```

C中相邻字符串字面量会被合并，因此

```
#define GENERIC_MAX(type) \
    type type##_max(type x, type y) \
    { \
        return x>y?x:y; \
    }
```

宏	含义
__DATE__	预处理的日期 ("Mmm dd yyyy"形式的字符串字面量, 如 Nov 23 2013)
__FILE__	表示当前源代码文件名的字符串字面量
__LINE__	表示当前源代码文件中行号的整型常量
__STDC__	设置为 1 时, 表明实现遵循 C 标准
__STDC_HOSTED__	本机环境设置为 1; 否则设置为 0
__STDC_VERSION__	支持 C99 标准, 设置为 199901L; 支持 C11 标准, 设置为 201112L
__TIME__	翻译代码的时间, 格式为 "hh:mm:ss"

预处理器

通过把宏参数列表中最后的参数写成省略号（即， 3个点...） 来实现这一功能。 这样， 预定义宏

`_ __VA_ARGS__` 可用在替换部分中， 表明省略号代表什么。 例如， 下面的定义： #define PR(...) printf(_ __VA_ARGS__)

```
#define PR(X, ...) printf("Message " #X ": " _ __VA_ARGS__)
```

记住， 省略号只能代替最后的宏参数：

```
#define WRONG(X, ..., Y) #X #_ __VA_ARGS__ #y //不能这样做
```

```
#include <stdio.h>
#include <math.h>
#define PR(X, ...) printf("Message " #X ": " _ __VA_ARGS__)
int main(void)
{
    double x = 48;
    double y;
    y = sqrt(x);
    PR(1, "x = %g\n", x);
    PR(2, "x = %.2f, y = %.4f\n", x, y);
}
return 0;
```

预处理器

一、`_func_`

二、`#if define(DEBUG)`

...

`#endif`

三、`#include` 命令是预处理命令的一种，预处理命令可以将别的源代码内容插入到所指定的位置；可以标识出只有在特定条件下才会被编译的某一段程序代码；可以定义类似标识符功能的宏，在编译时，预处理器会用别的文本取代该宏。

<code>#include <stdio.h></code>	←查找系统目录
<code>#include "quectel.h"</code>	←查找当前工作目录
<code>#include "/usr/biff/p.h"</code>	←查找/usr/biff目录

```
#ifndef __QUECTEL_H__  
#define __QUECTEL_H__  
#include <stdio.h>
```

```
#define QUECTEL_SW 10  
  
#endif
```

明示常量——例如，`stdio.h`中定义的EOF、NULL和BUFSIZE（标准I/O缓冲区大小）。

`#ifndef`指令通常用于防止多次包含一个文件

```
/*stdio.h*/  
#ifndef _STDIO_H  
#define _STDIO_H  
//省略了文件的内容  
#endif  
。
```

课后作业

5.1 编写一个宏NELEMS(a) 来计算一维数组a中的元素个数

5.2 有

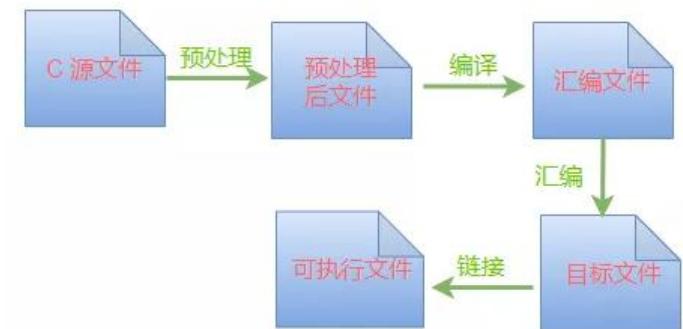
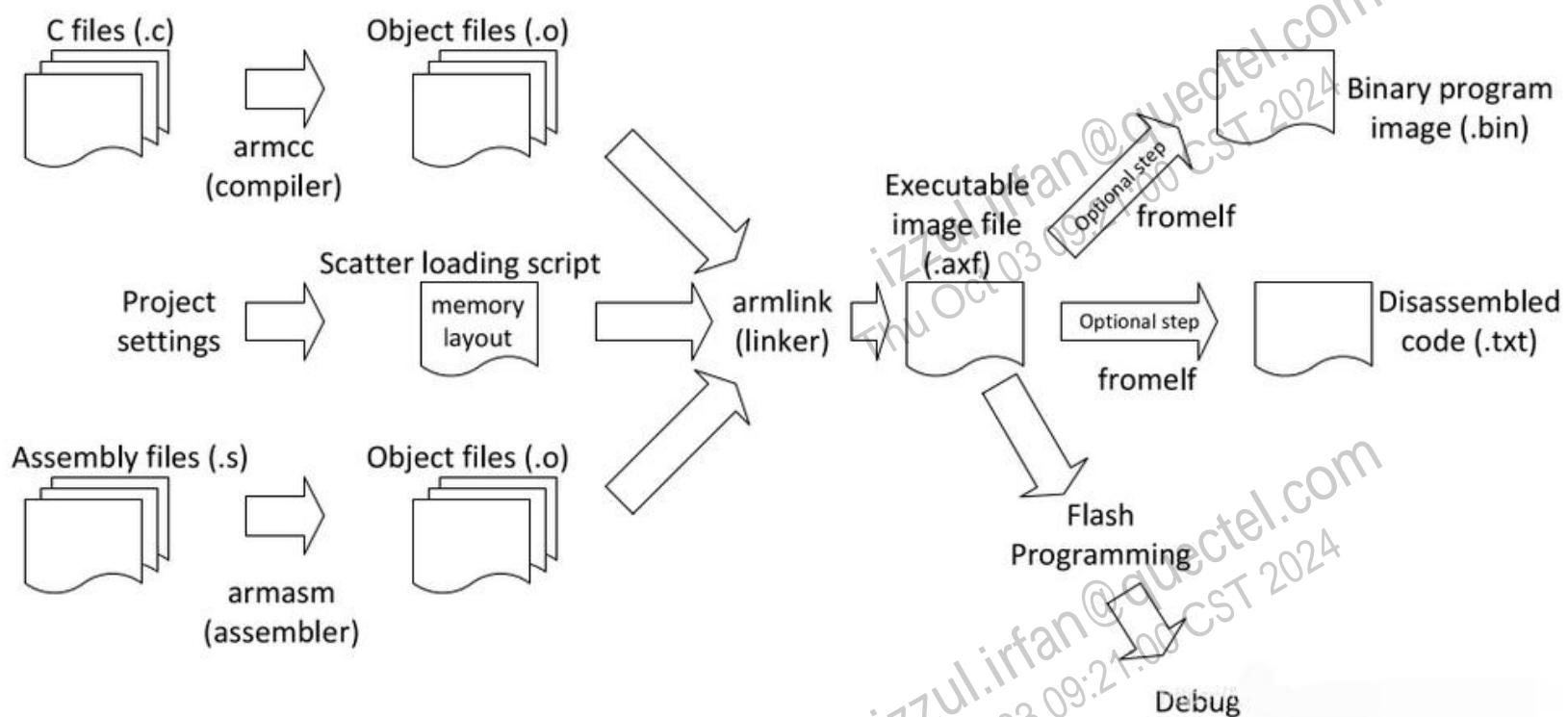
```
#define GENERIC_MAX(type) \
type type##_max(type x, type y) \
{ \
    \
    return x > y ? x : y; \
}
```

a) 写出GENERIC_MAX(long) 被预处理扩展后的形式。

b) 解释为什么GENERIC_MAX不能应用于unsigned long 这样的基本类型

编译原理

- 1、编译以单个c文件为单位;
- 2、预编译阶段h文件被展开在c文件;
- 3、编译阶段进行C语言词法、语法、声明等检查; 如报错缺少某变量的声明
- 4、链接阶段根据链接脚本进行变量内存地址的分配、定义的检查; 如报错某某函数未定义
- 5、编译结束确定了函数栈的变化。



编译原理

我们在C程序中使用变量来“代表”一个数据，使用函数名来“代表”一个函数，变量名和函数名是程序员使用的助记符。变量和函数最终是要放到内存中才能被CPU使用的，而内存中所有的信息(代码和数据)都是以二进制的形式来存储的，计算机根据就不会从格式上来区分哪些是代码、哪些是数据。**CPU在访问内存的时候需要的是地址，而不是变量名、函数名。**

问题来了：在程序代码中使用变量名来指代变量，而变量在内存中是根据地址来存放的，这二者之间如何映射(关联)起来的？

答案是：**编译器！** 编译器在编译文本格式的C程序文件时，会根据目标运行平台(就是编译出的二进制程序运行在哪里？是x86平台的电脑？还是ARM平台的开发板？)来安排程序中的各种地址，例如：加载到内存中的地址、代码段的入口地址等等，同时编译器也会**把程序中的所有变量名，转成该变量在内存中的存储地址。**

变量有2个重要属性：变量的类型和变量的值。

示例：代码中定义了一个变量

```
int a = 20;
```

类型是int型，值是20。这个变量在内存中的存储模型为：



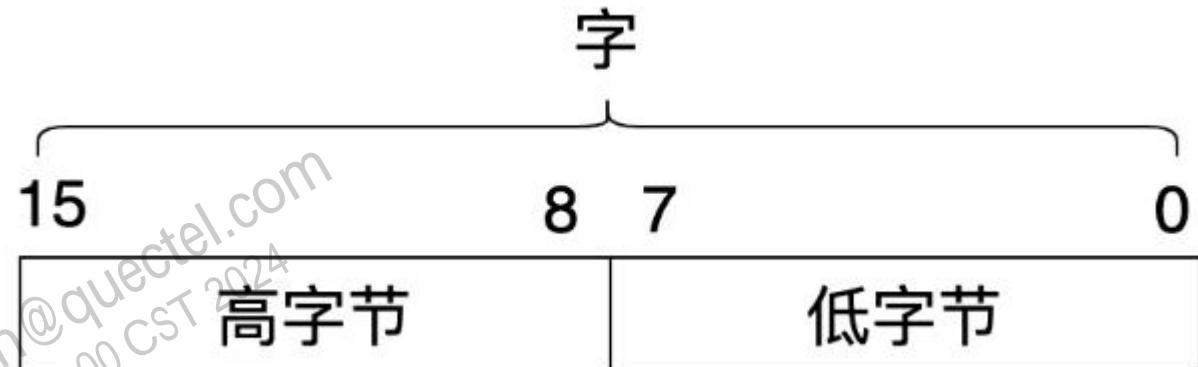
编译原理

在 CPU 内部，一些都是代表 0 或 1 的电信号，这些二进制数字的一组电信号出现在处理器内部线路上，它们是一排高低电平的组合，代表着二进制数中的每一位。

在处理器内部，必须用一个称为寄存器的电路把这些数据锁存起来。

因此，寄存器本质上也属于存储器的一种。只不过它们位于处理器的内部，CPU 访问寄存器比访问内存的速度更快。

处理器总是很忙的，在它操作的过程中，所有数据在寄存器里面只能是临时存在一小会，然后再被送往别处，这就是为什么它被叫做“寄存器”。



编译原理

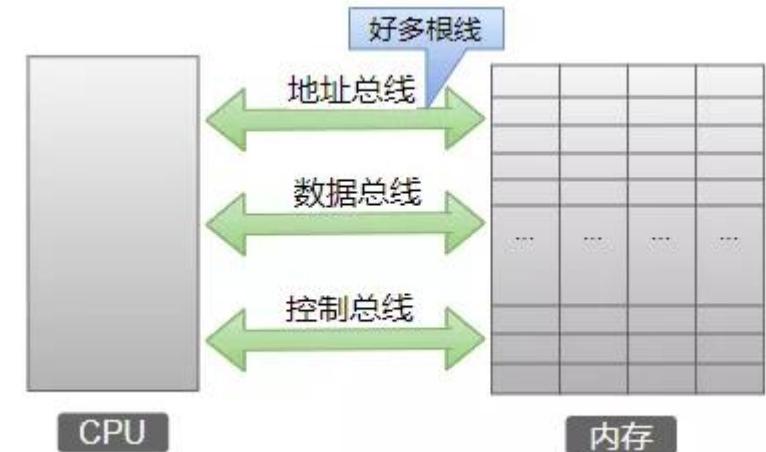
我们平时所说的计算机是32位、64位，指的是计算机的CPU中寄存器的最大存储长度，如果寄存器中最大存储32bit的数据，就称之为32位系统。

在计算机中，数据一般都是在硬盘、内存和寄存器之间进行来回存取。

CPU通过3种总线把各组成部分联系在一起：地址总线、数据总线和控制总线。地址总线的宽度决定了CPU的寻址能力，也就是CPU能达到的最大地址范围。

内存是通过地址来管理的，那么CPU想从内存中的某个地址空间上存取一个数据，那么CPU就需要在地址总线上输出这个存储单元的地址。假如地址总线的宽度是8位，能表示的最大地址空间就是256个字节，能找到内存中最大的存储单元是255这个格子(从0开始)。即使内存条的实际空间是2G字节，CPU也没法使用后面的内存地址空间。如果地址总线的宽度是32位，那么能表示的最大地址就是2的32次方，也就是4G字节的空间。

【注意】：这里只是描述地址总线的概念，实际的计算机中地址计算方式要复杂的多，比如：虚拟内存中采用分段、分页、偏移量来定位实际的物理内存，在分页中还有大页、小页之分，感兴趣的同学可以自己查一下相关资料。

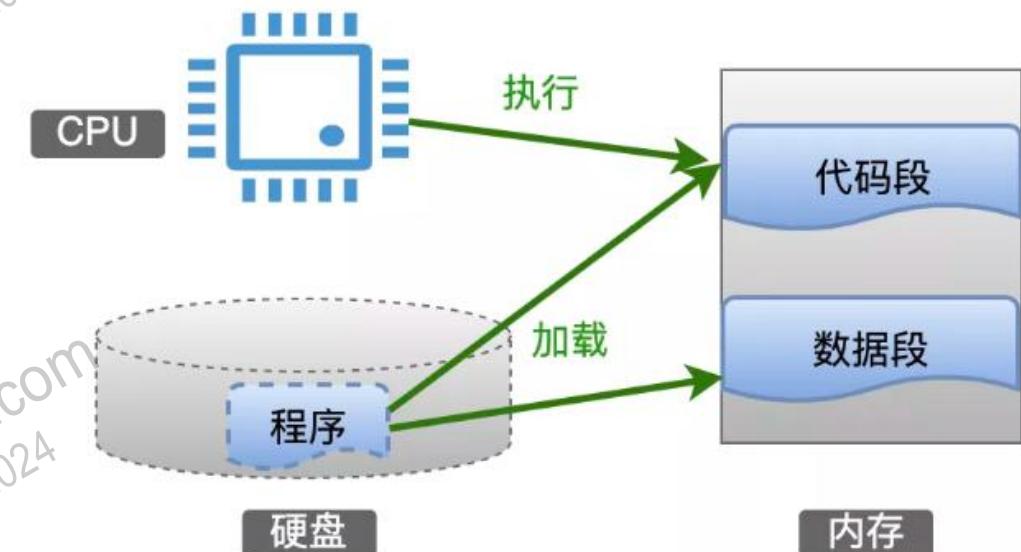


编译原理

我们编写一个程序源文件之后，编译得到的二进制可执行文件存放在电脑的硬盘上，此时它是一个静态的文件，一般称之为程序。

当这个程序被启动的时候，操作系统将会做下面几件事情：

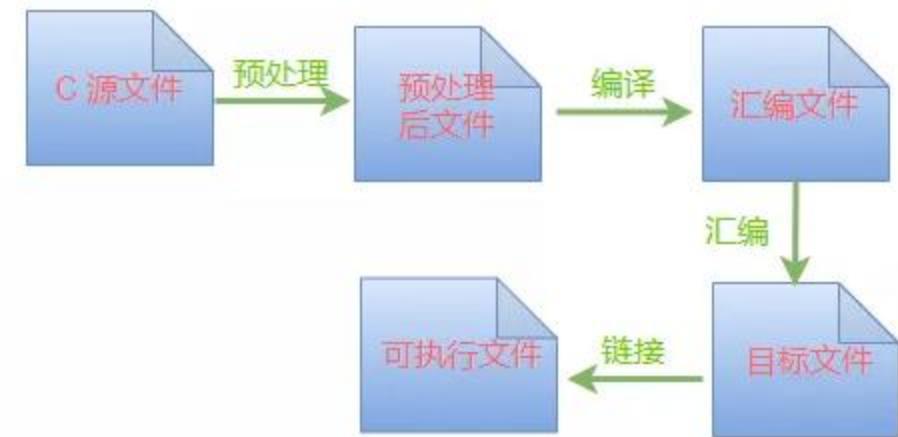
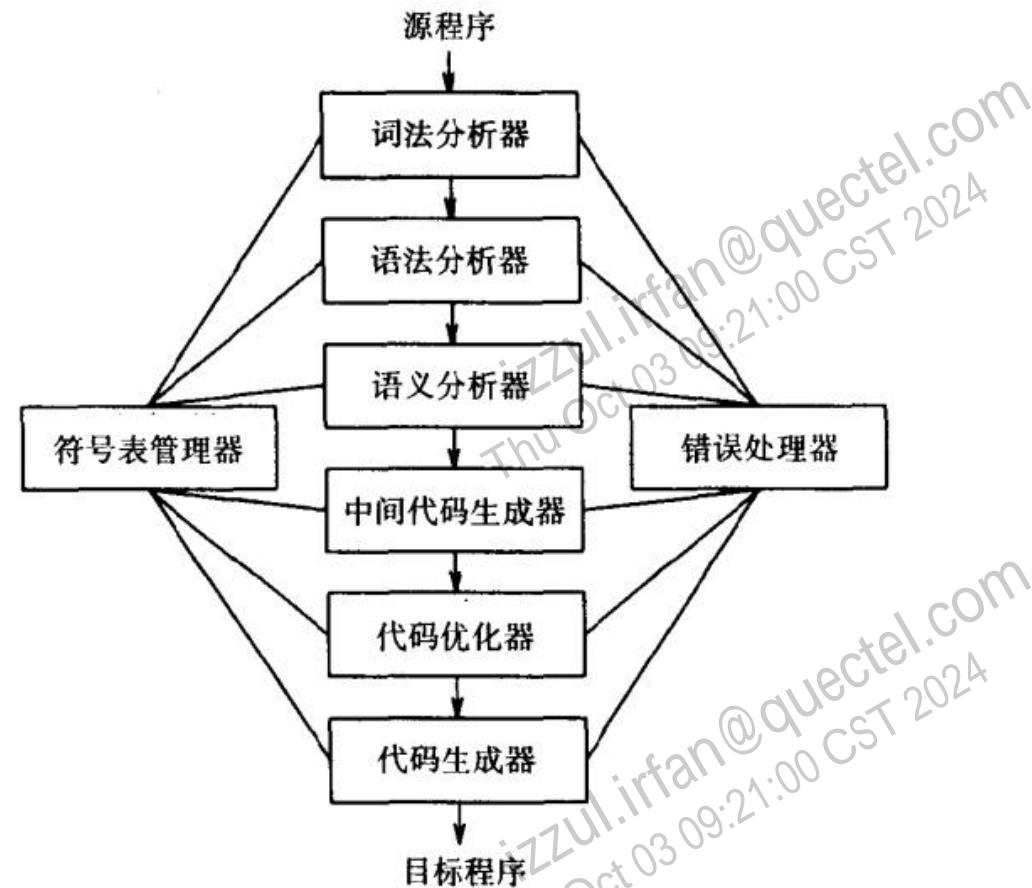
1. 把程序的内容(代码段、数据段)从硬盘复制到内存中；(XIP例外)
2. 创建一个数据结构PCB(进程控制块)，来描述这个程序的各种信息(例如：使用的资源，打开的文件描述符...);
3. 在代码段中定位到入口函数的地址，让CPU从这个地址开始执行



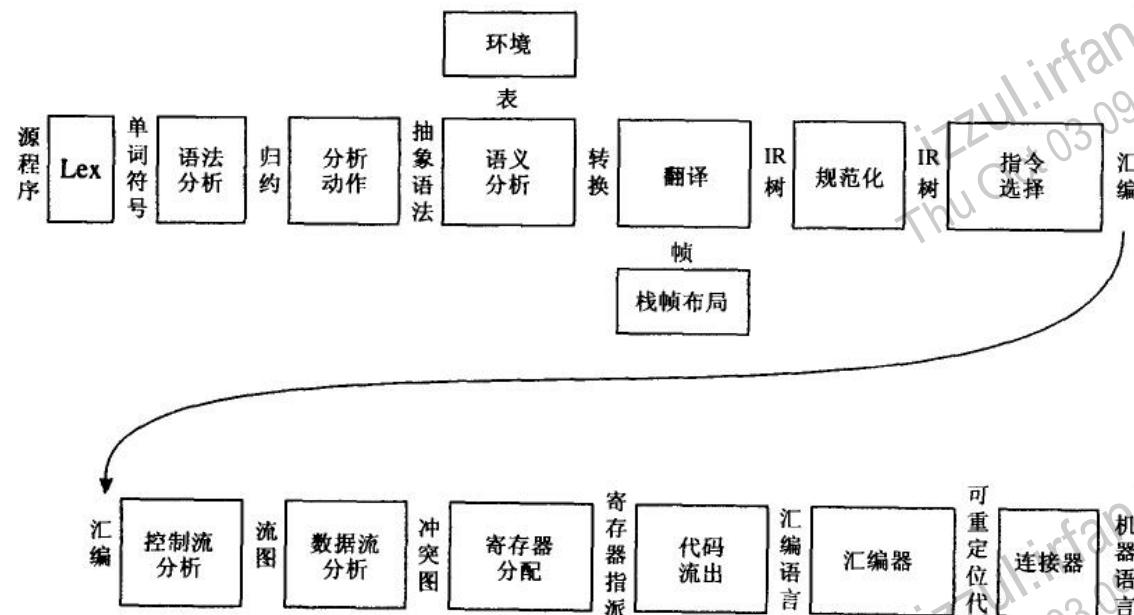
编译原理

.c---(预处理)-->.i---(编译)-->*.s---(汇编)-->*.o---(连接)--->可行性文件

之前章节讲了预处理阶段，这里将主要介绍编译阶段。



编译原理



阶段	描述
词法分析	将源文件分解为一个个独立的单词符号
语法分析	分析程序的短语结构
语义动作	建立每个短语对应的抽象语法树
语义分析	确定每个短语的含义，建立变量和其声明的关联，检查表达式的类型，翻译每个短语
栈帧布局	按机器要求的方式将变量、函数参数等分配于活跃记录（即栈帧）内
翻译	生成中间表示树（IR树），这是一种与任何特定程序设计语言和目标机体系结构无关的表示
规范化	提取表达式中的副作用，整理条件分支，以方便下一阶段的处理
指令选择	将IR树结点组合成与目标机指令相对应的块
控制流分析	分析指令的顺序并建立控制流图，此图表示程序执行时可能流经的所有控制流
数据流分析	收集程序变量的数据流信息，例如，活跃分析（liveness analysis）计算每一个变量仍需使用其值的地点（即它的活跃点）
寄存器分配	为程序中的每一个变量和临时数据选择一个寄存器，不在同一点活跃的两个变量可以共享同一个寄存器
代码流出	用机器寄存器替代每一条机器指令中出现的临时变量名

编译原理



词法分析:就是要把这一串连续的数字根据C语言的词法规则切割成----分立的token(标识符、数字、符号)并鉴别属性;

int a = 10 ; //分析称 “int”,“a”,“=”,“10”,“;”

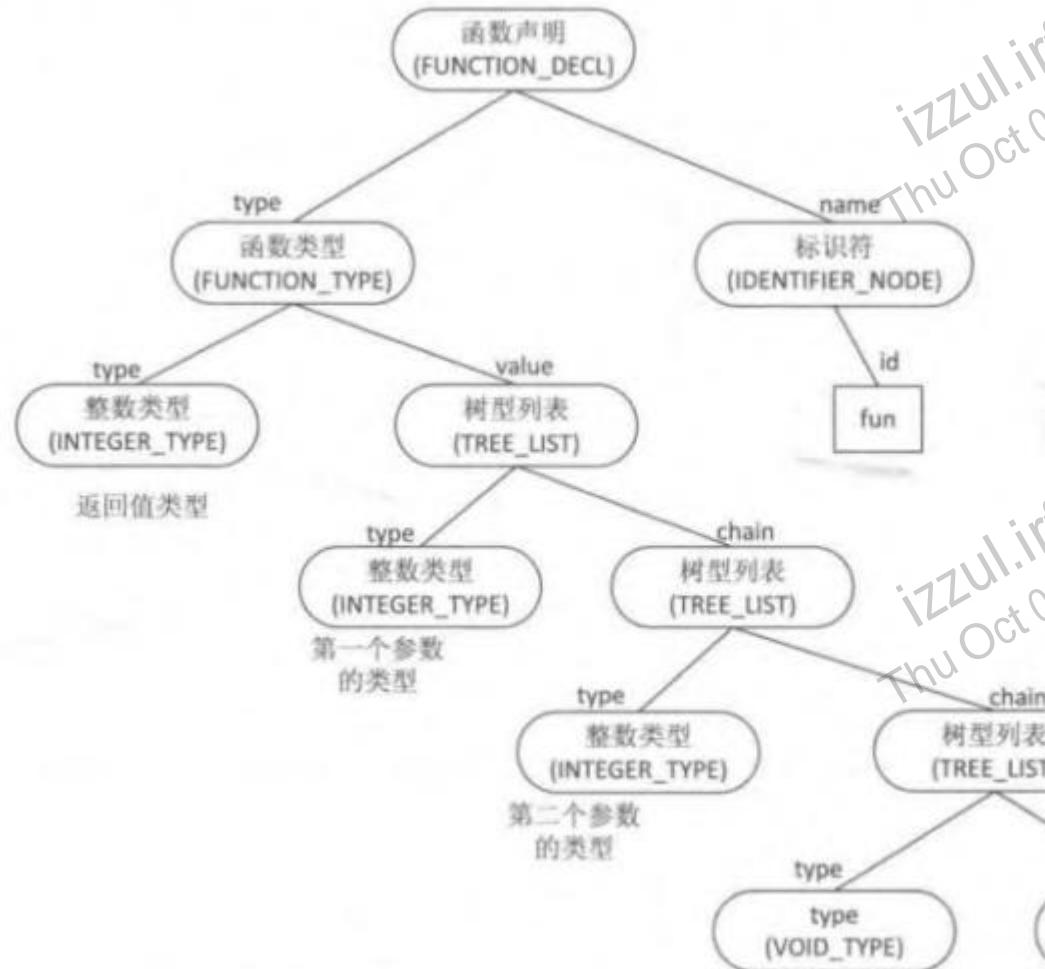
语法分析:就是从连续的token中识别出标识符、关键字、数字、运算符并存储为符号(token)流，从token流中识别出符合C语言语法的语句

Int	fun	(int a,int b)	;
返回值类型	函数名	(参数声明表)	;

函数声明模板

这段token流最终与C语言规定的函数声明模板相匹配,编译器就认为此语句为函数声明语句。并以语法树的形式在内存中记录下来。

编译原理



语义分析：检测源程序的语义错误并收集代码生成阶段要用到的类型信息。将变量的定义与他们各个使用联系起来，检查每一个表达式是否有正确的类型。
这个阶段还要进行符号表的管理。

编译原理

从语法树到中间代码

由于计算机存在多种架构的CPU硬件,如X86、ARM平台, 考虑到程序在不同CPU之间的可移植性, 先转换成一个通用的、抽象的CPU 指令代码, 这就是中间代码的设计思想。然后再根据具体的CPU 指令集落实到具体的CPU目标代码。

选定具体的CPU、操作系统后, 中间代码可以转换为目标代码----汇编代码。然后由汇编器依照选定操作系统的目
标文件, 将*.s文件转换为具体的目标文件, 对于linux而言是*.o文件, windows是*.obj。目标文件中已经是CPU的机器指令了

最后连接器把一个或多个目标文件(库文件本质上也是目标文件)链接成符合选择定操作系统指定的可执行文件

.bin格式文件：

Bin文件是最纯粹的二进制机器代码，或者说是“顺序格式”。按照assembly code顺序翻译成binary machine code，内部没有地址标记。Bin是直接的内存映象表示，二进制文件大小即为文件所包含的数据的实际大小。BIN文件就是直接的二进制文件，一般用编程器烧写时从00开始，而如果下载运行，则下载到编译时的地址即可。可以直接在裸机上运行。

.hex格式文件：

Intel hex 文件常用来保存单片机或其他处理器的目标程序代码。它保存物理程序存储区中的目标代码映象。一般的编程器都支持这种格式。就是机器代码的十六进制形式，并且是用一定文件格式的ASCII码来表示。

HEX文件由记录（RECORD）组成。在HEX文件里面，每一行代表一个记录。每条记录都由一个冒号“：“打头，其格式如下：

:BBAAAATTHHHH...HHHHCC

BB:字节个数。

AAAA:数据记录的开始地址,高位在前,低位在后。

TT: Type

00数据记录，用来记录数据。

01记录结束，放在文件末尾，用来标识文件结束。

02用来标识扩展段地址的记录

04扩展地址记录(表示32位地址的前缀)

HHHH:一个字(Word)的数据记录,高字节在前,低字节在后。TT之后共有 BB/2 个字的数据。

CC: 占据一个Byte的CheckSum

编译原理

elf格式文件：

ELF (Executable and linking format) 文件是x86 Linux系统下的一种常用目标文件(objectfile)格式，有三种主要类型：

(1) 适于连接的可重定位文件(relocatable file)，可与其它目标文件一起创建可执行文件和共享目标文件。

*.o文件就是可重定位文件,由c 源文件经过预编译、编译、汇编步骤生成

(2) 适于执行的可执行文件(executable file)，用于提供程序的进程映像，加载到内存执行。

Linux OS下，通常gcc -o test test.c，生成的test文件就是ELF格式的，在Linux Shell下输入./test 就可以执行。Windows 下的 .exe

(3) 共享目标文件(shared object file),连接器可将它与其它可重定位文件和共享目标文件连接成其它的目标文件，动态连接器又可将它与可执行文件和其它共享目标文件结合起来创建一个进程映像。

简单的例子，就是.lib文件

axf格式文件：

和elf文件类似；但是axf是有arm 开发的armcc编译器生成的； gcc编译器生成的一般称为elf；两者很像，但是格式上有区别，此处不进一步讨论。

map文件

map 文件对应的中文名应该是映射文件，用来展示（映射）项目构建的链接阶段的细节。通常包含程序的全局符号、交叉引用和内存映射等等信息。目前，大多数编译套件（主要是其中的链接器）都可以生成 map 文件。常见的 GCC、VC、IAR 都可以输出 map 文件（PC(x86)平台的 map 文件与 ARM 平台的差别较大）。

编译原理



ARM架构体系主流的编译器

比较	ARMCC	IAR	GCC for ARM	LLVM (clang)
命令行工具	随IDE发布，也独立提供	仅随其IDE发布，不独立提供	独立提供	只有命令行工具
开发商	ARM	IAR	ARM、Linaro、Mentor	LLVM
支持的平台	Windows、Linux	Windows	Windows、Linux、Mac (部分)	Windows、Linux、Mac
配套IDE	Keil MDK、ARM Development Studio 5、ADS	IAR EMBEDDED WORKBENCH FOR ARM	除以上两者外的其他支持ARM的IDE，例如：eclipse、Visual Studio	除以上两者外的其他支持ARM的IDE，例如：eclipse、Visual Studio

编译原理

GCC for ARM

GCC 全称为 GNU Compiler Collection。GCC 是几种主要编程语言的编译器的集成分发。这些语言目前包括 C, C++, Objective-C, Objective-C++, Fortran, Ada, Go 和 BRIG (HSAIL)。

GCC 原名为 GNU C 语言编译器 (GNU C Compiler)，因为它原本只能处理 C 语言。GCC 很快地扩展，变得可处理 C++。后来又扩展能够支持更多编程语言，如 Fortran、Pascal、Objective-C、Java、Ada、Go 以及各类处理器架构上的汇编语言等，所以改名 GNU 编译器套件 (GNU Compiler Collection)。更名之后，原来的针对于 C 语言的编译器名字还叫 gcc，针对 C++ 的编译器叫做 g++。

GCC for ARM 则是基于 GCC 开发的，用来编译生成 ARM 内核可执行文件的编译套件，俗称 ARM 交叉编译套件。相比于以上两个巨贵的编译器，GCC for ARM 因为是基于开源的 GCC 的，因此是免费的。目前主要由三大主流工具商提供，第一是 ARM，第二是 Codesourcery，第三是 Linora。目前我们用的针对 ARM 芯片的集成开发环境 (IDE)，除了 IAR 和 ARM 自己的 Keil、DS，大多都是使用 GCC for ARM 的编译器！

首先，看看 ARM 交叉编译工具链的命名规则：arch [-vendor] [-os] [-(gnu)eabi] [-gcc]

arch: 体系架构，如 ARM, MIPS

vendor: 工具链提供商，没有 vendor 时，用 none 代替；

os: 目标操作系统，没有 os 支持时，也用 none 代替

eabi: 嵌入式应用二进制接口 (Embedded Application Binary Interface)

```
#-----  
# Configure compiling utilities  
#-----  
CC:=$(GCC_INSTALL_PATH)\bin\arm-none-eabi-gcc.exe  
CPP:=$(GCC_INSTALL_PATH)\bin\arm-none-eabi-g++.exe  
LD:=$(GCC_INSTALL_PATH)\bin\arm-none-eabi-ld.exe  
AR:=$(GCC_INSTALL_PATH)\bin\arm-none-eabi-ar.exe  
OBJCOPY:=$(GCC_INSTALL_PATH)\bin\arm-none-eabi-objcopy.exe  
MAKE:=$(COMPILE_TOOL_PATH)\gnumake\gnumake.exe  
CRC_BIN:=$(TOP_DIR)\quectel_build\bin\crc_bin.exe
```

编译原理

Map文件的作用

不用的编译器生成的map文件格式可能有差异，但是调试属性大致相同。

ARMCC编译器下armlink生成map文件

1. 分析问题
2. 优化程序
3. 学习了解连接过程

Image Symbol Table

镜像符号映射表。就是每个符号（这里的符号可以指模块、变量、函数）实际的地址等信息。

Memory Map of the image

该映射包含镜像文件中每个加载域，执行域和输入节（包括连接器生成的输入节）的地址和大小。

Execution Region DDR_NONCACHE_REGION_ACIPC_PS_DL_ENDMARKER (Base: 0x7ef4ffffc, Size: 0x00000004, Max: 0x00000004, ABSOLUTE)

Base Addr	Size	Type	Attr	Idx	E Section Name	Object
-----------	------	------	------	-----	----------------	--------

0x7ef4ffffc	0x00000004	Zero	RW	8	DDR_NONCACHE_REGION_ACIPC_PS_DL_ENDMARKER.bss	anon\$\$.obj.o
-------------	------------	------	----	---	---	----------------

```

> Section Cross References ... armlink --xref
=====
> Removing Unused input sections from the image... armlink --info unused
581 unused section(s) (total 29938 bytes) removed from the image.

=====
> Image Symbol Table ... armlink --symbols
=====

> Memory Map of the image ... armlink --map
=====

> Image component sizes ... armlink --info sizes
注意：--info sizes 实际等效于 --info sizes,totals，所以 --info totals 无效
=====
> armlink --info totals
      Code (inc. data)   RO Data   RW Data   ZI Data   Debug   Grand Totals
      45526       2968       1130       4268       5572    1047714   ELF Image Totals (compressed)
      45526       2968       1130        868       5572    1047714
      45526       2968       1130        868          0          0   ROM Totals
=====
注意：单独 --info totals 还会有些内容
=====
      Total RO Size (Code + RO Data) ..... 46656 ( 45.56kB)
      Total RW Size (RW Data + ZI Data) ..... 9840 ( 9.61kB)
      Total ROM Size (Code + RO Data + RW Data) ..... 47524 ( 46.41kB)
=====
```

编译原理

.data: 部分保存有助于程序内存映像的已初始化数据

.text: 本节包含程序的文本或可执行指令

.bss: 本节保存有助于程序内存映像的未初始化数据

Image component sizes

该部分列出了组成镜像的各部分内容的大小等详细信息。主要有三部分组成：用户文件大小信息、库文件大小信息、汇总信息。

Code (inc. data): 这对应两列数据，分别表示代码占用的字节数和代码中内联数据占用的字节数。inc. data 是内联数据 (inline data) 的缩写。内联数据包括文字池和短字符串等。RO Data: 模块中 RO 数据占用的字节数。这是除去 Code (inc. data) 列中 inc. data 数据外的只读数据的字节数。例如，我们定义的 const 数组！

RW Data: 模块中 RW 数据占用的字节数。

ZI Data: 模块中 ZI 数据占用的字节数。

Debug: 模块中调试数据占用的字节数。例如，调试用的输入节以及符号和字符串表。

Object Name: 对象文件的名字。

编译原理

Image component sizes						
Code (inc. data)	RO Data	RW Data	ZI Data	Debug	Object Name	
212	58	0	12	0	2124	GKITick.o
74962	22016	54503	1036	0	3026739	diagDB.o
28	4	44	0	0	1530	dsp_filters.o
298	150	0	0	0	85070	hal_init.o
1600	168	0	8	0	192378	ps_init.o
2708	332	316	5	202	34033	ps_nvram.o
448	14	0	0	0	1919	tavor_packages.o
1122	50	0	64	452	45289	usbmgrtppal.o
Object Totals						
85736	24512	54864	1128	6203100	3389082	(incl. Generated)
4306	1720	0	0	6202444	0	(incl. Padding)
52	0	1	3	2	0	
Code (inc. data)	RO Data	RW Data	ZI Data	Debug	Library Member Name	
8	4	0	31	0	985	stubs.o
720	0	0	0	0	144	C45.o
4016	14	967	0	0	732	Dictionary.o
630	0	0	0	0	176	FootAdjust.o
972	6	487	0	0	304	FootRule.o
3566	42	113	0	0	1112	FrontDefine.o
126	0	0	0	0	68	IsASCIITable.o
120	4	266	0	0	132	IsCECommon.o

Code (inc. data)	RO Data	RW Data	ZI Data	Debug	
3919588	472644	1433360	114244	8567112	54644298 Grand Totals
3919588	472644	1433360	114244	8567112	54644298 ELF Image Totals
3919588	472644	1433360	114244	0	0 ROM Totals
<hr/>					
Total RO Size (Code + RO Data)					5352948 (5227.49kB)
Total RW Size (RW Data + ZI Data)					8681356 (8477.89kB)
Total ROM Size (Code + RO Data + RW Data)					5467192 (5339.05kB)

ROM Totals: 显示包含镜像所需的 ROM 的最小大小。这其中不包括未存储在 ROM 中的 ZI 数据和调试信息

Total RO Size: 就是我们的可执行程序中常量数据（代码和只读数据）的大小。

Total RW Size: 就是我们的可执行程序中需要占用的内存的大小。（已经初始化的全局变量）

Total ROM Size: 就是我们的可执行程序本身的大小。这个大小就等于我们的可执行文件的大小。（bin文件大小）

编译原理----C程序在ARM芯片上运行



ARM芯片的寄存器

1. r0-r3 用作传入函数参数，传出函数返回值。在子程序调用之间，可以将 r0-r3 用于任何用途。被调用函数在返回之前不必恢复 r0-r3。---如果调用函数需要再次使用 r0-r3 的内容，则它必须保留这些内容。
2. r4-r11 被用来存放函数的局部变量。如果被调用函数使用了这些寄存器，它在返回之前必须恢复这些寄存器的值。r11 是栈帧指针 fp(保存栈底)。
3. r12 是内部调用暂时寄存器 ip。它在过程链接胶合代码（例如，交互操作胶合代码）中用于此角色。在过程调用之间，可以将它用于任何用途。被调用函数在返回之前不必恢复 r12。
4. 寄存器 r13 是栈指针 sp(栈顶)。它不能用于任何其它用途。sp 中存放的值在退出被调用函数时必须与进入时的值相同。
5. 寄存器 r14 是链接寄存器 lr。如果您保存了返回地址，则可以在调用之间将 r14 用于其它用途，程序返回时要恢复
6. 寄存器 r15 是程序计数器 pc。它不能用于任何其它用途。

编译原理----C程序在ARM芯片上运行



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

```
int m = 8;
int fun(int a,int b)
{
    int c = 0;
    c = a + b;
    return c;
}
int main()
{
    int i = 4;
    int j = 5;
    m = fun(i, j);
    return 0;
}
```

```
00010400 <main>:
10440: e92d4800      push   {fp, lr}
10444: e28db004      add    fp, sp, #4
10448: e24dd008      sub    sp, sp, #8
1044c: e3a03004      mov    r3, #4
10450: e50b300c      str    r3, [fp, #-12]
10454: e3a03005      mov    r3, #5
10458: e50b3008      str    r3, [fp, #-8]
1045c: e51b1008      ldr    r1, [fp, #-8]
10460: e51b000c      ldr    r0, [fp, #-12]
10464: ebffffe5      bl    10400 <fun>
10468: e1a02000     mov    r2, r0
1046c: e59f3010      ldr    r3, [pc, #16] ; 10484 <main+0x44>
10470: e5832000      str    r2, [r3]
10474: e3a03000      mov    r3, #0
10478: e1a00003     mov    r0, r3
1047c: e24bd004      sub    sp, fp, #4
10480: e8bd8800      pop    {fp, pc}
10484: 00021024      andeq r1, r2, r4, lsr #32
00010400 <fun>:
10400: e52db004      push   {fp}
10404: e28db000      add    fp, sp, #0
10408: e24dd014      sub    sp, sp, #20
1040c: e50b0010      str    r0, [fp, #-16]
10410: e50b1014      str    r1, [fp, #-20] ; 0xfffffffffec
10414: e3a03000      mov    r3, #0
10418: e50b3008      str    r3, [fp, #-8]
1041c: e51b2010      ldr    r2, [fp, #-16]
10420: e51b3014      ldr    r3, [fp, #-20] ; 0xfffffffffec
10424: e0823003     add    r3, r2, r3
10428: e50b3008      str    r3, [fp, #-8]
1042c: e51b3008      ldr    r3, [fp, #-8]
10430: e1a00003     mov    r0, r3
10434: e24bd000      sub    sp, fp, #0
10438: e49db004      pop    {fp} ; (ldr fp, [sp], #4)
1043c: e12ffff1e      bx    lr
```

编译原理----C程序在ARM芯片上运行

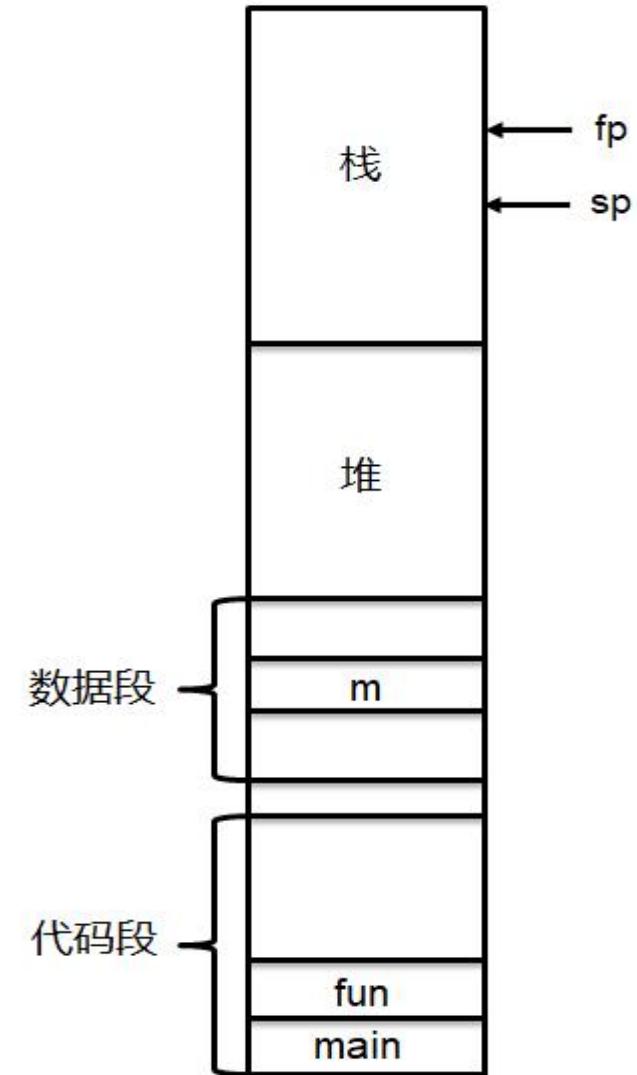


右图为一块内存存储介质，如PSRAM，我们可以将一块内存划分成几个区域。

代码段:存放代码(text段);

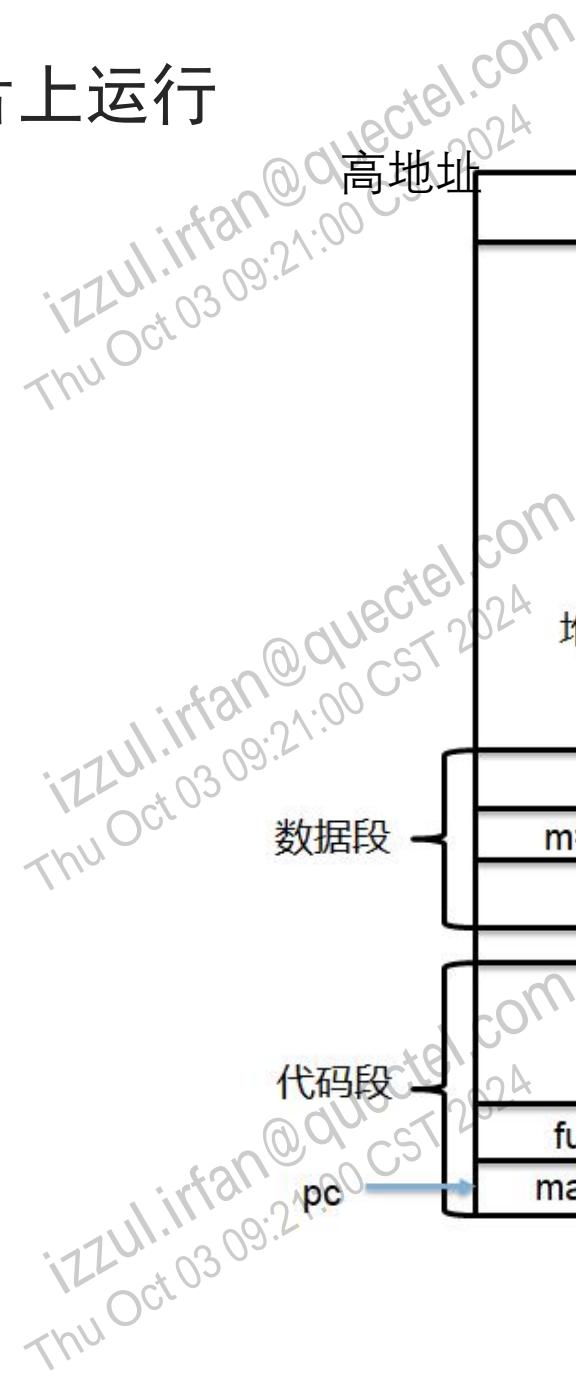
数据段:ZI段和RO data

堆: Malloc分配的内存地址就在堆中。



编译原理----C程序在ARM芯片上运行

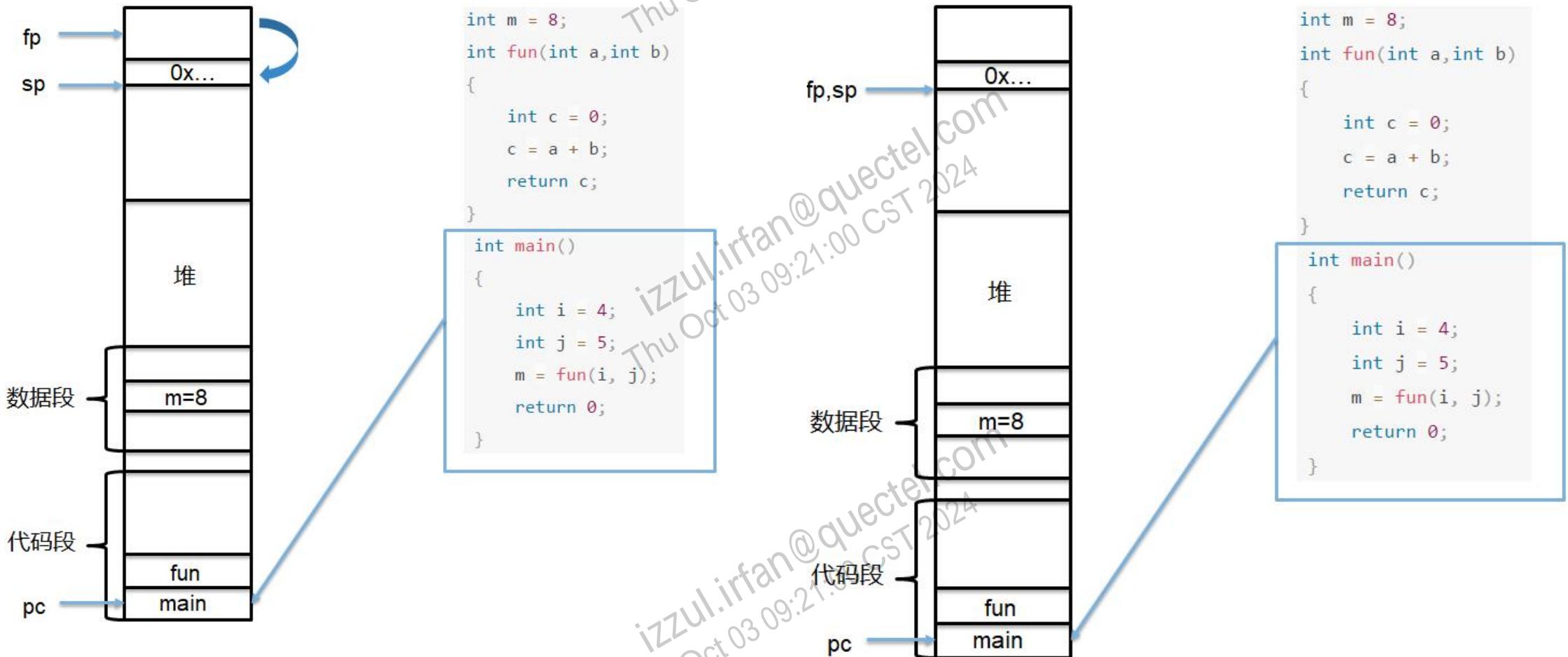
1、在程序运行前，先初始化全局变量m



```
int m = 8;
int fun(int a,int b)
{
    int c = 0;
    c = a + b;
    return c;
}
int main()
{
    int i = 4;
    int j = 5;
    m = fun(i, j);
    return 0;
}
```

编译原理----C程序在ARM芯片上运行

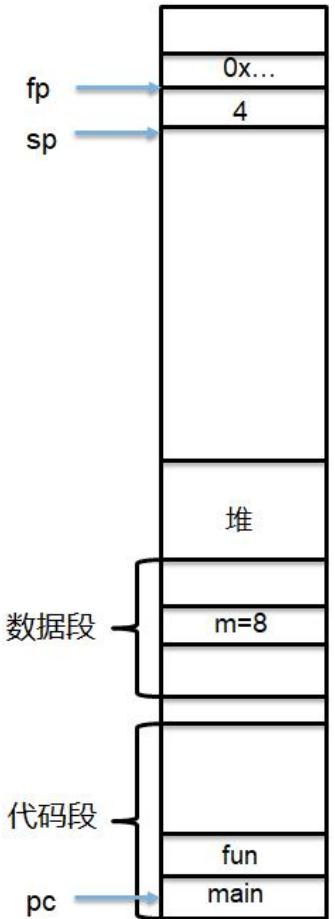
- 2、保存进入main之前的栈底, fp-sp之间是当前函数栈
- 3、准备好main函数的栈



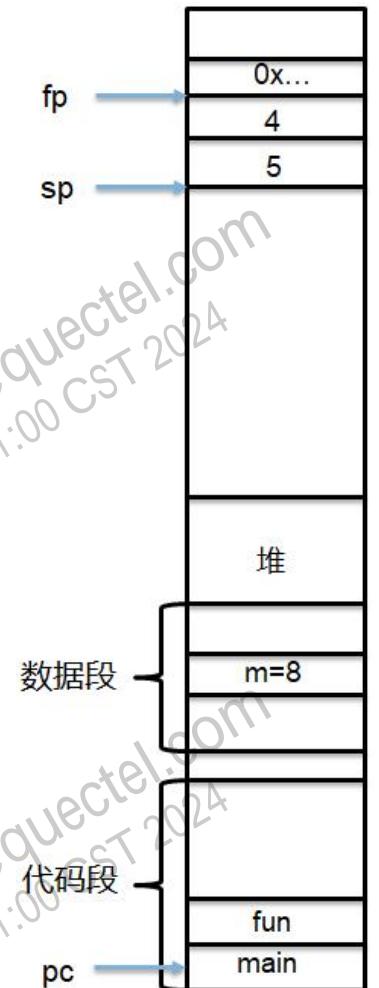
编译原理----C程序在ARM芯片上运行

4、局部变量i入栈

5、局部变量j入栈



```
int m = 8;
int fun(int a,int b)
{
    int c = 0;
    c = a + b;
    return c;
}
int main()
{
    int i = 4;
    int j = 5;
    m = fun(i, j);
    return 0;
}
```

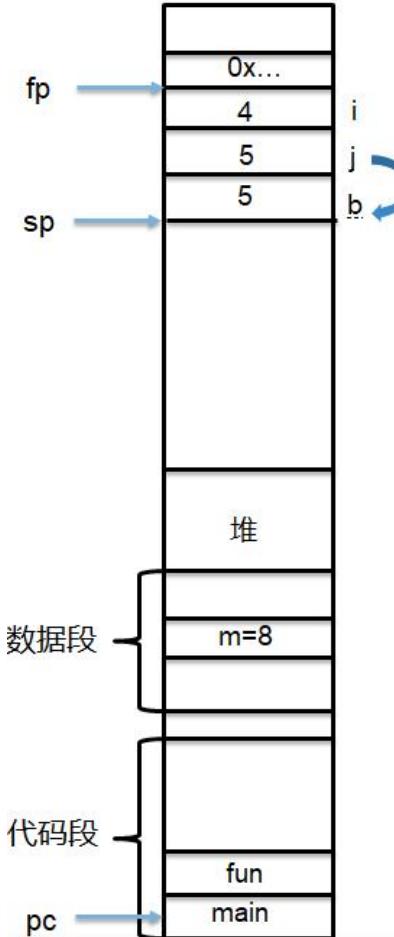


```
int m = 8;
int fun(int a,int b)
{
    int c = 0;
    c = a + b;
    return c;
}
int main()
{
    int i = 4;
    int j = 5;
    m = fun(i, j);
    return 0;
}
```

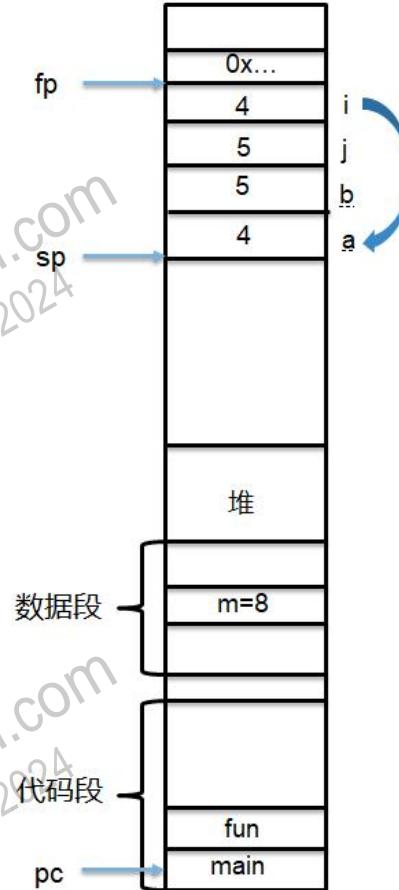
编译原理----C程序在ARM芯片上运行

6、准备函数fun的调用, 形参反向入栈 先形参b入栈

7、形参a入栈



```
int m = 8;
int fun(int a,int b)
{
    int c = 0;
    c = a + b;
    return c;
}
int main()
{
    int i = 4;
    int j = 5;
    m = fun(i, j);
    return 0;
}
```

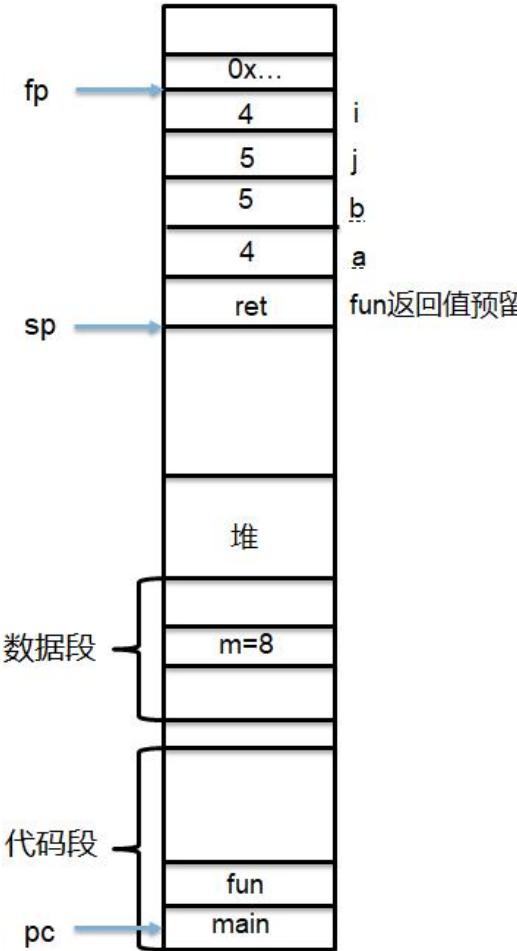


```
int m = 8;
int fun(int a,int b)
{
    int c = 0;
    c = a + b;
    return c;
}
int main()
{
    int i = 4;
    int j = 5;
    m = fun(i, j);
    return 0;
}
```

编译原理----C程序在ARM芯片上运行

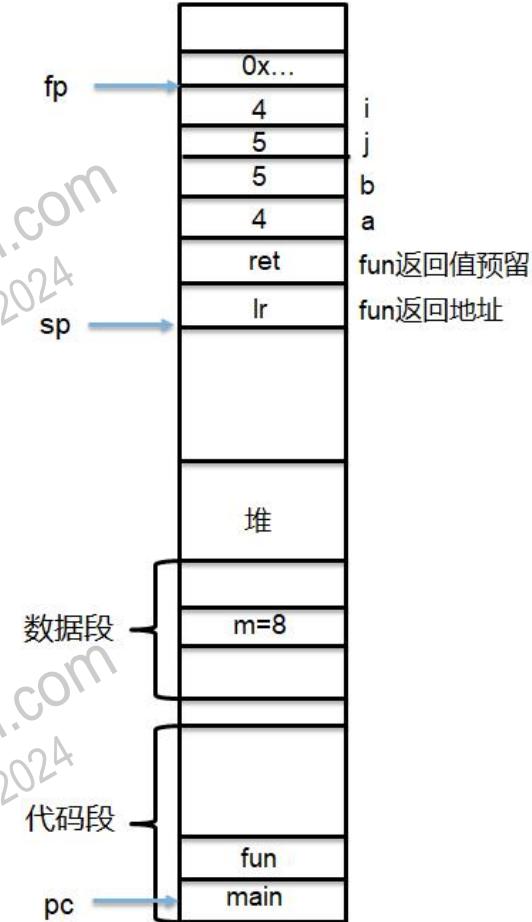
8、留空一个地址作为fun返回值, 待后面返回时填入

9、fun返回地址入栈, 通常是main函数当前pc指针的下一个



```

int m = 8;
int fun(int a,int b)
{
    int c = 0;
    c = a + b;
    return c;
}
int main()
{
    int i = 4;
    int j = 5;
    → m = fun(i, j);
    return 0;
}
  
```



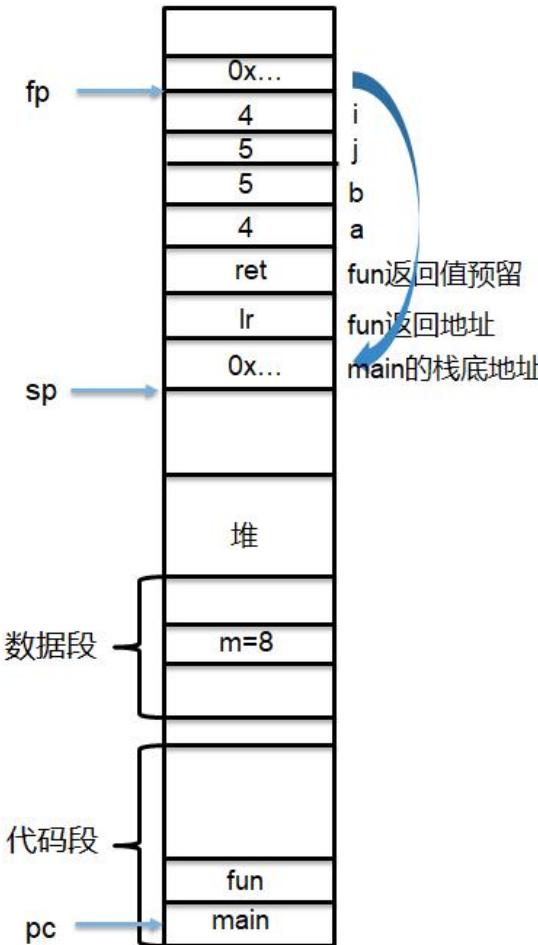
```

int m = 8;
int fun(int a,int b)
{
    int c = 0;
    c = a + b;
    return c;
}
int main()
{
    int i = 4;
    int j = 5;
    → m = fun(i, j);
    return 0;
}
  
```

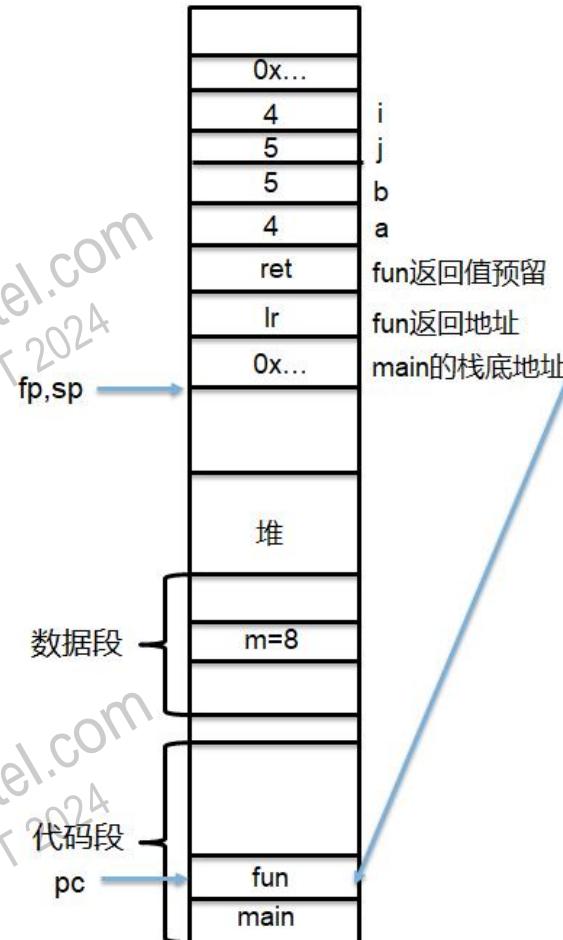
编译原理----C程序在ARM芯片上运行

10、main函数的栈底地址入栈

11、pc指针跳转fun代码



```
int m = 8;
int fun(int a,int b)
{
    int c = 0;
    c = a + b;
    return c;
}
int main()
{
    int i = 4;
    int j = 5;
    → m = fun(i, j);
    return 0;
}
```

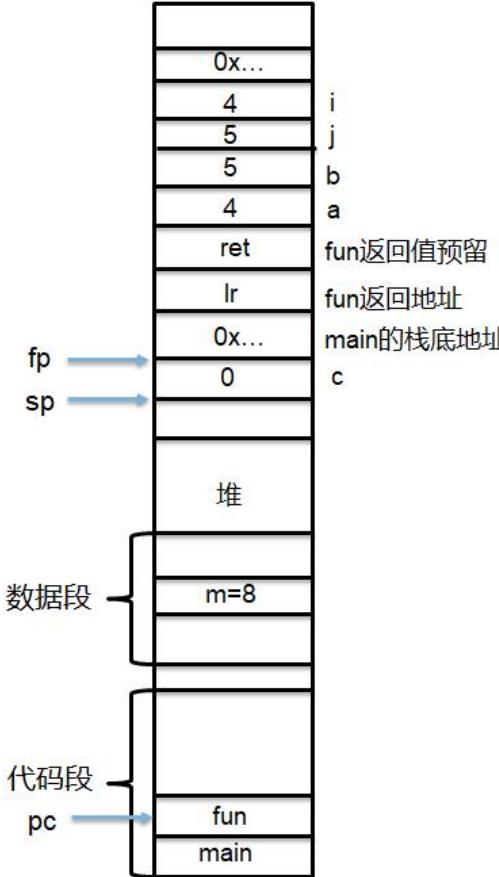


```
int m = 8;
int fun(int a,int b)
{
    int c = 0;
    c = a + b;
    return c;
}
int main()
{
    int i = 4;
    int j = 5;
    m = fun(i, j);
    return 0;
}
```

编译原理----C程序在ARM芯片上运行

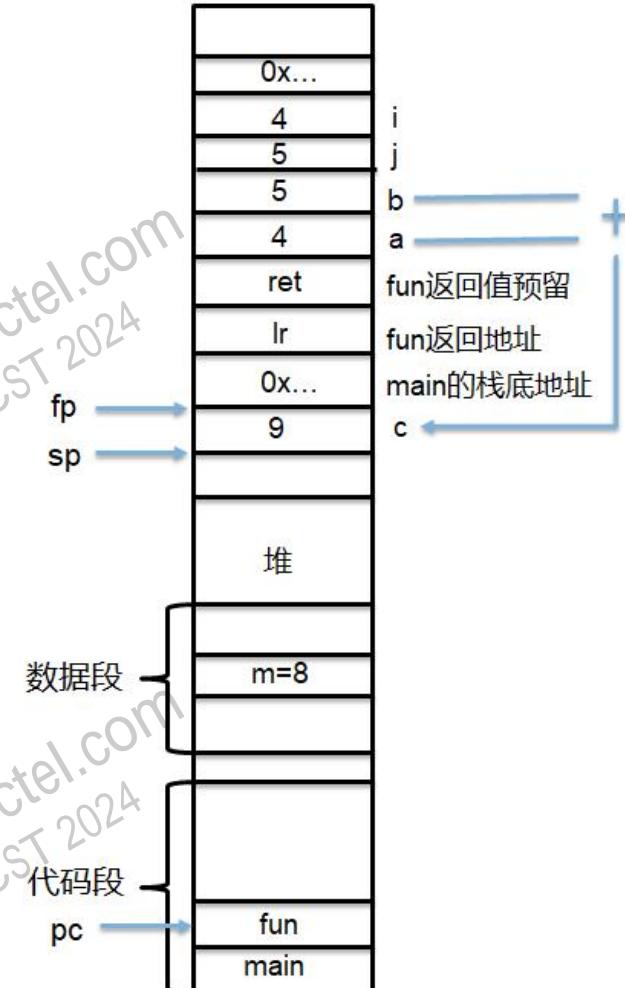
12、C入栈

13、可以看到函数fun的数据形参a,b 在上一层函数的栈中.一部分在自己的栈上.此步取值到加法器中进行加法运算,再赋值给c



```

int m = 8;
int fun(int a,int b)
{
    int c = 0;
    c = a + b;
    return c;
}
int main()
{
    int i = 4;
    int j = 5;
    m = fun(i, j);
    return 0;
}
  
```



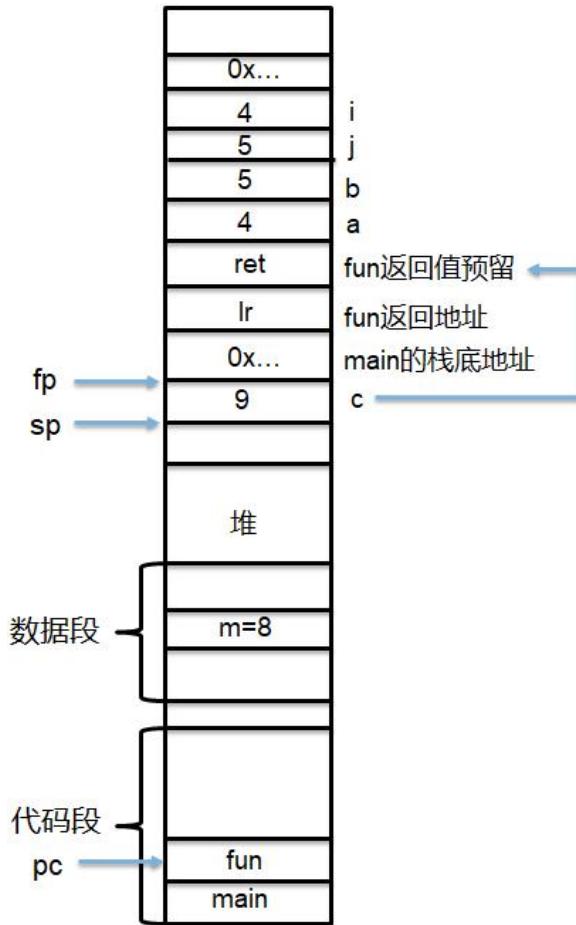
```

int m = 8;
int fun(int a,int b)
{
    int c = 0;
    c = a + b;
    return c;
}
int main()
{
    int i = 4;
    int j = 5;
    m = fun(i, j);
    return 0;
}
  
```

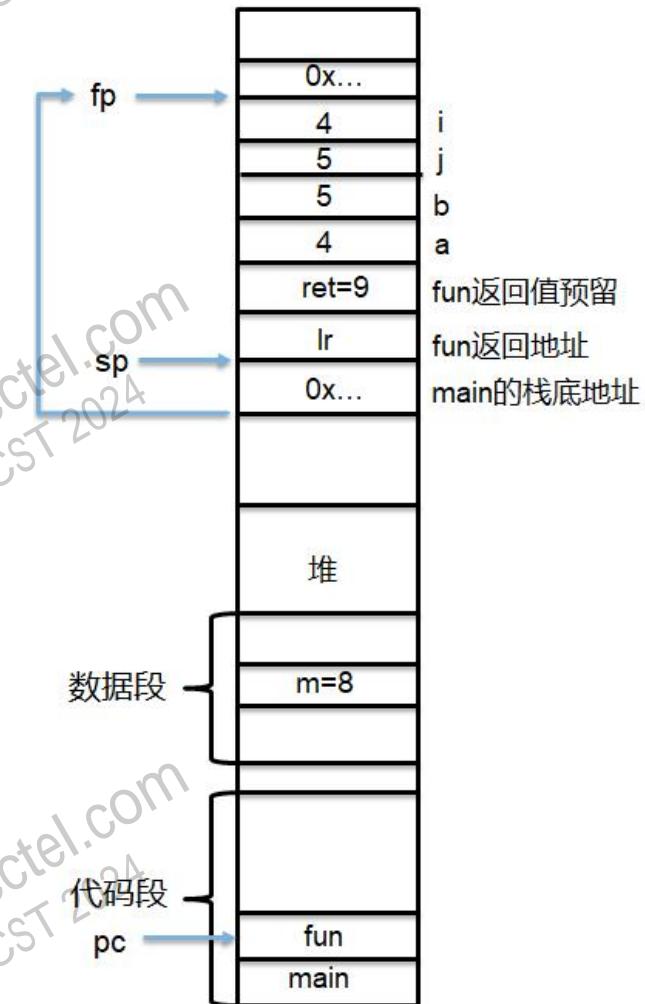
编译原理----C程序在ARM芯片上运行

14、c赋给返回值,填入上面的留空位置

15、栈底恢复上一层



```
int m = 8;
int fun(int a,int b)
{
    int c = 0;
    c = a + b;
    → return c;
}
int main()
{
    int i = 4;
    int j = 5;
    m = fun(i, j);
    return 0;
}
```

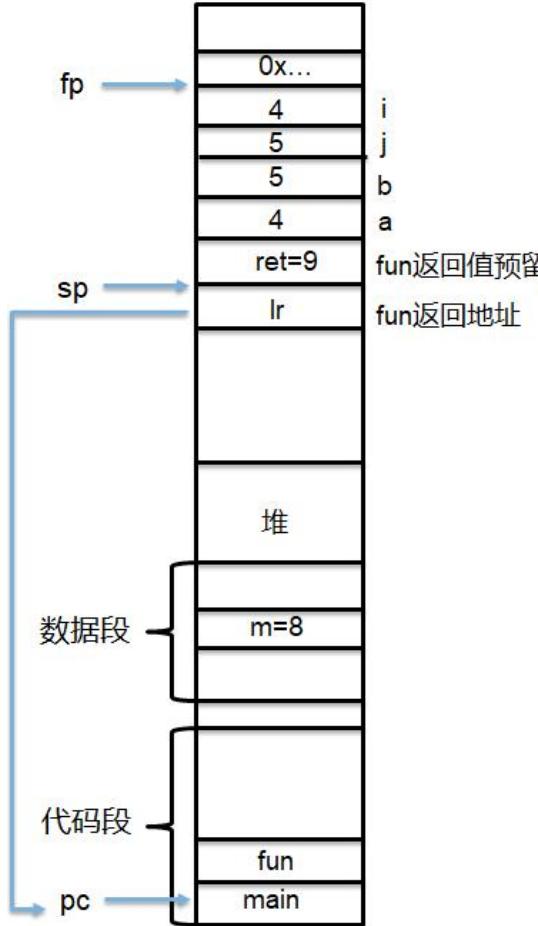


```
int m = 8;
int fun(int a,int b)
{
    int c = 0;
    c = a + b;
    → return c;
}
int main()
{
    int i = 4;
    int j = 5;
    m = fun(i, j);
    return 0;
}
```

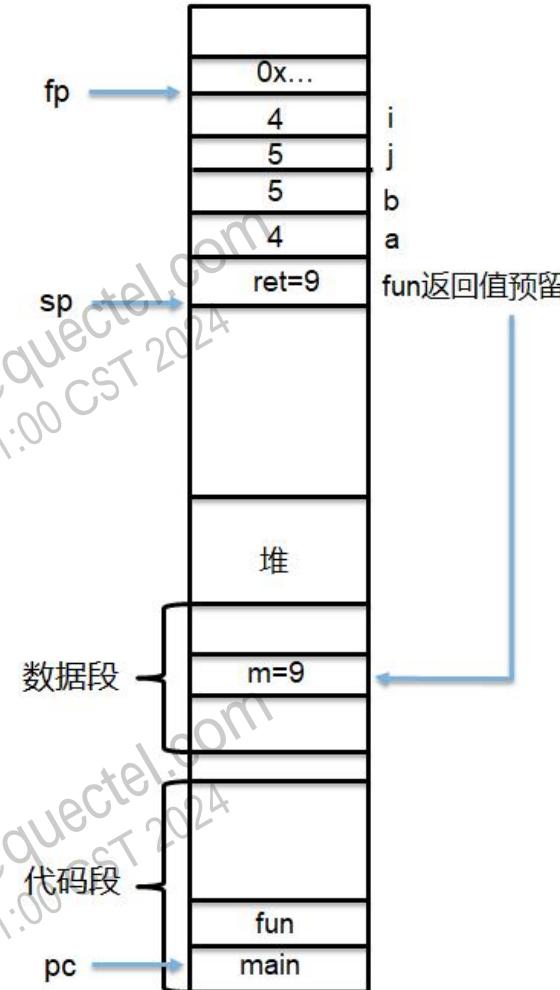
编译原理----C程序在ARM芯片上运行

16、lr赋值给PC，实现指针跳转

17、返回值赋值给全局变量m



```
int m = 8;
int fun(int a,int b)
{
    int c = 0;
    c = a + b;
    return c;
}
int main()
{
    int i = 4;
    int j = 5;
    m = fun(i, j);
    return 0;
}
```

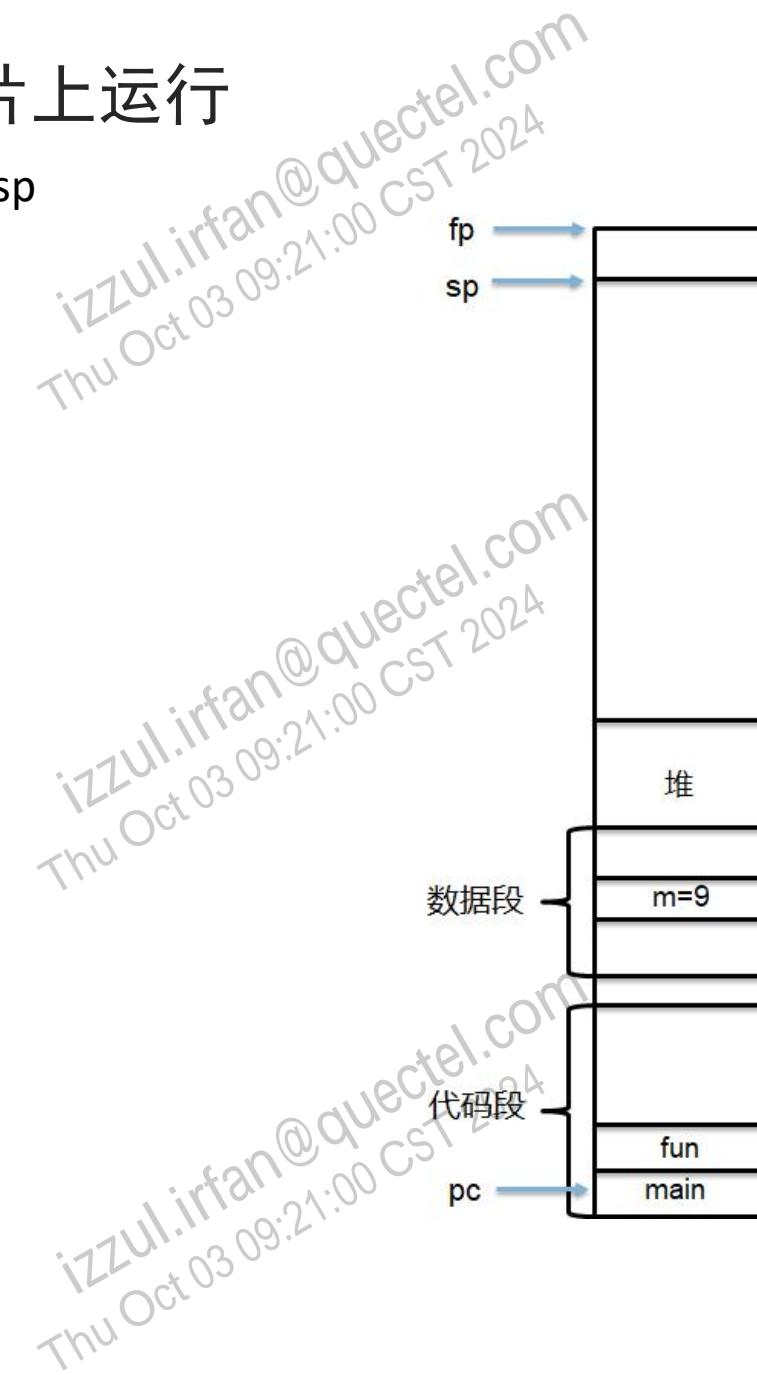
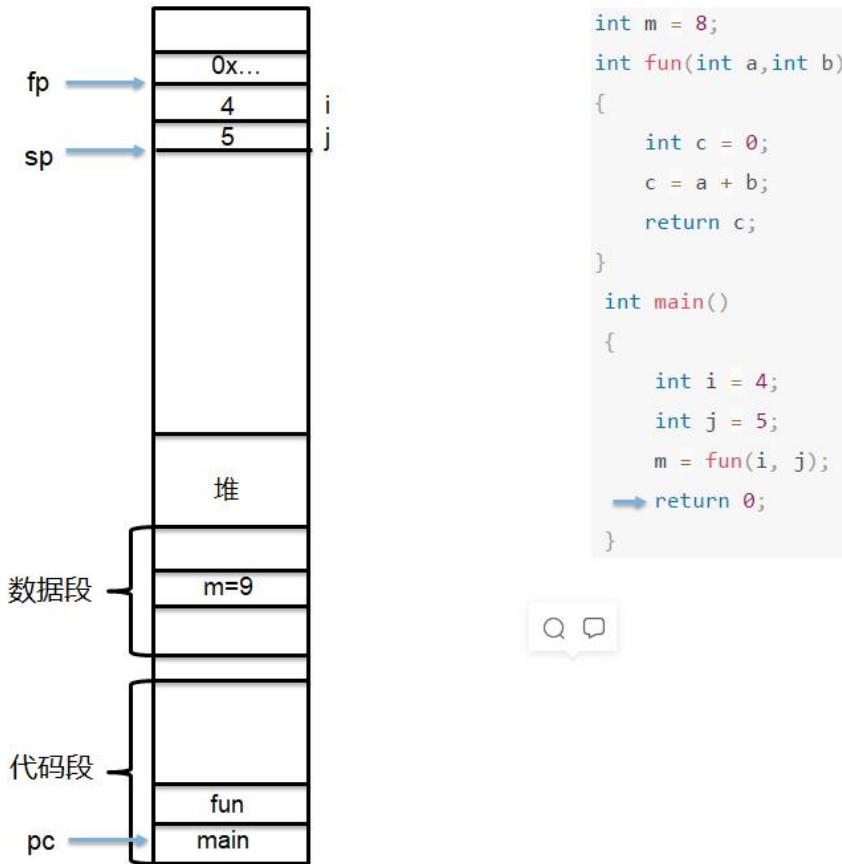


```
int m = 8;
int fun(int a,int b)
{
    int c = 0;
    c = a + b;
    return c;
}
int main()
{
    int i = 4;
    int j = 5;
    m = fun(i, j);
    return 0;
}
```

编译原理----C程序在ARM芯片上运行

18、前面函数调用的形参已经无用,回滚sp

19、函数返回,清理main的栈空间



```

int m = 8;
int fun(int a,int b)
{
    int c = 0;
    c = a + b;
    return c;
}
int main()
{
    int i = 4;
    int j = 5;
    m = fun(i, j);
    return 0;
}

```



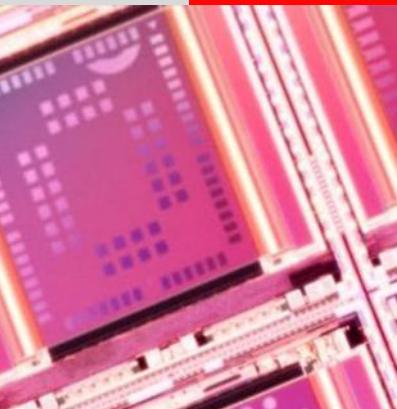
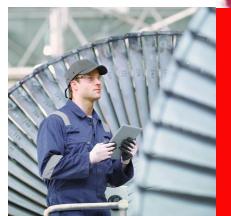
第五课 链表、栈、队列、树

Build a Smarter World

www.quectel.com

izzul.irfan@quectel.com
Thu Oct 03 09:21:00 CST 2024

izzul.irfan@quectel.com
Thu Oct 03 09:21:00 CST 2024



数据结构

数据结构:简单的解释,是相互之间存在一种或多种特定关系的数据元素的集合。

数据元素都不是孤立存在的, 而是在他们之间存在着某种关系, 这种数据元素相互之间的关系成为结构

有以下四种类型:

- 1.集合: 数据结构中的元素之间除了“同属一个集合”的相互关系外, 别无其他关系; 如, c中数组
- 2.线性结构: 数据结构中的元素存在一对一的相互关系;
- 3.树形结构: 数据结构中的元素存在一对多的相互关系;
- 4.图形结构: 数据结构中的元素存在多对多的相互关系;

数据结构

数组、结构体

数组是可以再内存中连续存储多个元素的结构，在内存中的分配也是连续的，数组中的元素通过数组下标进行访问，数组下标从0开始。

优点：

- 1、按照索引查询元素速度快
- 2、按照索引遍历数组方便

缺点：

- 1、数组的大小固定后就无法扩容了
- 2、数组只能存储一种类型的数据
- 3、添加，删除的操作慢，因为要移动其他的元素。

适用场景：

频繁查询，对存储空间要求不大，很少增加和删除的情况。

数据结构



链表

链表是物理存储单元上非连续的、非顺序的存储结构，数据元素的逻辑顺序是通过链表的指针地址实现，每个元素包含两个结点，一个是存储元素的数据域(内存空间)，另一个是指向下一个结点地址的指针域。根据指针的指向，链表能形成不同的结构，例如单链表，双向链表，循环链表等。

链表的优点：

链表是很常用的一种数据结构，不需要初始化容量，可以任意加减元素；

添加或者删除元素时只需要改变前后两个元素结点的指针域指向地址即可，所以添加，删除很快；

缺点：

因为含有大量的指针域，占用空间较大；

查找元素需要遍历链表来查找，非常耗时。

适用场景：

数据量较小，需要频繁增加，删除操作的场景

数据结构



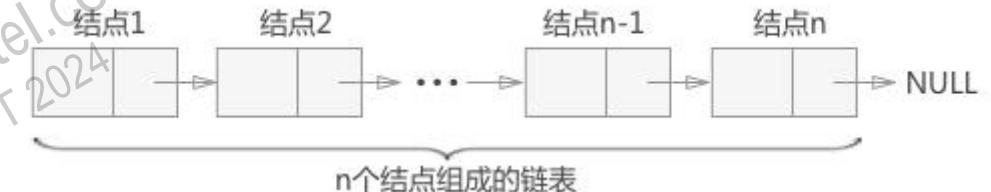
链表

逻辑结构上一个挨一个的数据，在实际存储时，并没有像顺序表(数组)那样也相互紧挨着。恰恰相反，数据随机分布在内存中的各个位置，这种存储结构称为线性表的链式存储。

由于分散存储，为了能够体现出数据元素之间的逻辑关系，每个数据元素在存储的同时，要配备一个指针，用于指向它的直接后继元素，即每一个数据元素都指向下一个数据元素（最后一个指向NULL(空)）。

1. 每个元素本身由两部分组成：本身的信息，称为“数据域”；
2. 指向直接后继的指针，称为“指针域”。

```
struct link{  
    int data;      //定义数据域  
    struct link *next; //定义指针域，存储直接后继的节点信息  
};
```



数据结构

链表，创建，删除，遍历，查找，删除节点，增加节点

```
struct link *create(int n){
    struct link *headnode,*node;
    headnode = (struct link *)malloc(sizeof(struct link)); //为头节点申请内存
    headnode->next = NULL; //让头节点的指针域置空
    for(int i=0;i<n;i++){
        node = (struct link *)malloc(sizeof(struct link)); //给新建节点申请内存
        scanf("%d",&node->data); //新建节点数据域传值
        node->next = headnode->next; //新建节点的数据域指向头节点 == 创建尾节点
        headnode->next = node; //将新建节点数据域传给头节点
    }
    return headnode;
}
```

```
void deleteNode(struct Stu *head,int n){ //删除n处的节点
    struct Stu *p = head,*pr = head;
    int i=0;
    while(i<n&&p!=NULL){ //到达指定节点，此时p指向指定节点，pr指向上一节点
        pr = p; //将p的地址赋值给pr
        p = p->next; //p指向下一节点
        i++;
    }
    if(p!=NULL){ //当p不为空时，即p不能指向尾节点之后的节点
        pr->next = p->next;
        free(p);
    } else{
        printf("节点不存在! \n");
    }
}
```

```
void insertNode(struct Stu *head,int n){ //插入节点
    struct Stu *p = head,*pr;
    pr = (struct Stu*)malloc(sizeof(struct Stu)); //让pr指向新建节点申请的内存
    printf("input data:\n");
    scanf("%d %s",&pr->id,pr->name);
    int i = 0;
    //当插入位置是尾节点时，只要在尾节点后再插入一个节点，并让尾节点的指针
    //域指向新建节点，新建节点的指针域置空
    while(i<n&&p!=NULL){ //使p指向将要插入节点的位置
        p = p->next;
        i++;
    }
    if(p!=NULL){ //如果p没越界
        pr->next = p->next; //将新建节点的地址指向将要插入节点的后
        p->next = pr; //使插入节点指向新建节点
    } else{
        printf("节点不存在! \n");
    }
}
```

- while($p \rightarrow next \neq NULL$) 循环结束时，此时p的位置是尾节点的位置，但如果用于输出函数的判断条件，则尾节点的数据不会输出。
- while($p \neq NULL$) 循环结束时，此时p指向的位置为尾节点的下一个节点，因为没有申请内存空间，所以是一个未知的区域。

数据结构

链表，创建，删除，遍历，查找，删除节点，增加节点

```
void change(struct Stu *head,int n){  
    struct Stu *p = head;  
    int i = 0;  
    while(i<n && p!=NULL){ //使p指向需修改节点  
        p = p->next;  
        i++;  
    }  
    if(p != NULL){  
        printf("请输入修改之后的值:\n");  
        scanf("%d %s",&p->id,p->name);  
    }else{  
        printf("节点不存在! \n");  
    }  
}
```

链表逆序

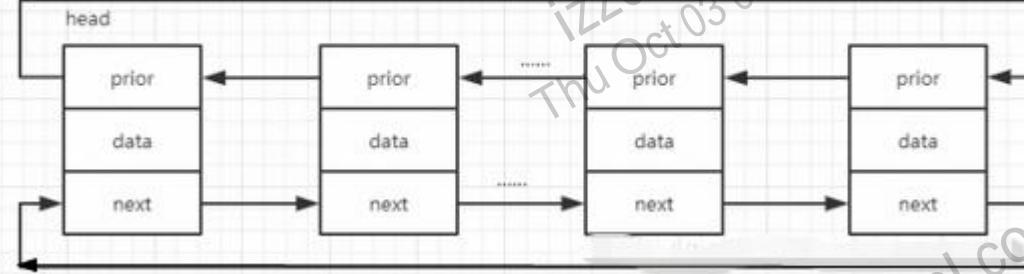
```
STU *link_reversed_order(STU *head)  
{  
    STU *pf = NULL, *pb = NULL, *tmp = NULL;  
    pf = head; //将头节点的地址赋值给pf  
    if(head == NULL) { //如果链表为空  
        printf("链表为空, 不需要逆序! \n");  
        return head;  
    } else if(head->next == NULL) { //如果只有一个节点  
        printf("链表只有一个节点, 不需要逆序! \n");  
        return head;  
    } else {  
        pb = pf->next; //pb指向pf的下一个节点  
        head->next = NULL; //头节点的指针域置空(变为尾节点)  
        while(pb != NULL) //当pb不为空时  
        {  
            tmp = pb; //将pb的地址赋值给temp  
            pb = pb->next; //pb指向下一个节点  
            tmp->next = pf; //pb的上一个节点的指针域指向pf  
            pf = tmp; //让pf指向tmp  
        }  
        head = pf;  
        return head;  
    }  
}
```

数据结构

链表

- 1、双向链表
- 2、环形链表

双向循环链表示意图



```
struct doubleCircularLinkedList
{
    struct doubleCircularLinkedList* prior;//结点的前驱指针
    int data;//结点的数据项
    struct doubleCircularLinkedList* next;//结点的后继指针
};
```

```
struct doubleCircularLinkedList* createList()
{
    //创建一个头结点，数据差异化当作表头
    struct doubleCircularLinkedList* headNode = (struct doubleCircularLinkedList*)malloc(sizeof(struct
doubleCircularLinkedList));
    //循环链表，所以初始化头指针，尾指针都是指向自身的，data数据域不做初始化
    headNode->prior = headNode;//头结点指向自身
    headNode->next = headNode;//尾结点指向自身
    return headNode;
}
```

数据结构

链表

- 1、双向链表
- 2、环形链表

```
struct doubleCircularLinkedList* createNode(int data)
{
    //动态申请内存malloc+free c语言的特点
    struct doubleCircularLinkedList* newNode = (struct
doubleCircularLinkedList*)malloc(sizeof(struct doubleCircularLinkedList));
    //创建结点过程相当于初始化过程
    newNode->data = data;//传入data数值初始化数据域
    newNode->prior = NULL;//初始化头结点为null
    newNode->next = NULL;//初始化尾结点为null
    return newNode;
}
//表头插入
void insertNodeByHead(struct doubleCircularLinkedList* headNode,int data)
{
    //创建一个新的结点，调用创建新结点的函数
    struct doubleCircularLinkedList* newNode = createNode(data);
    //修改四个指针变量
    newNode->prior = headNode;
    newNode->next = headNode->next;
    headNode->next->prior=newNode;
    headNode->next = newNode;
}
```

```
//表尾插入
void insertNodeByNext(struct doubleCircularLinkedList* headNode,int data)
{
    struct doubleCircularLinkedList* newNode = createNode(data);
    //首先找到最后一个结点的位置
    struct doubleCircularLinkedList* lastNode = headNode;
    while(lastNode->next != headNode)
    {
        lastNode = lastNode->next;
    }
    //找到之后调整四个指针
    headNode->prior = newNode;
    newNode->next = headNode;
    lastNode->next = newNode;
    newNode->prior = lastNode;
}
```

数据结构

链表

- 1、双向链表
- 2、环形链表

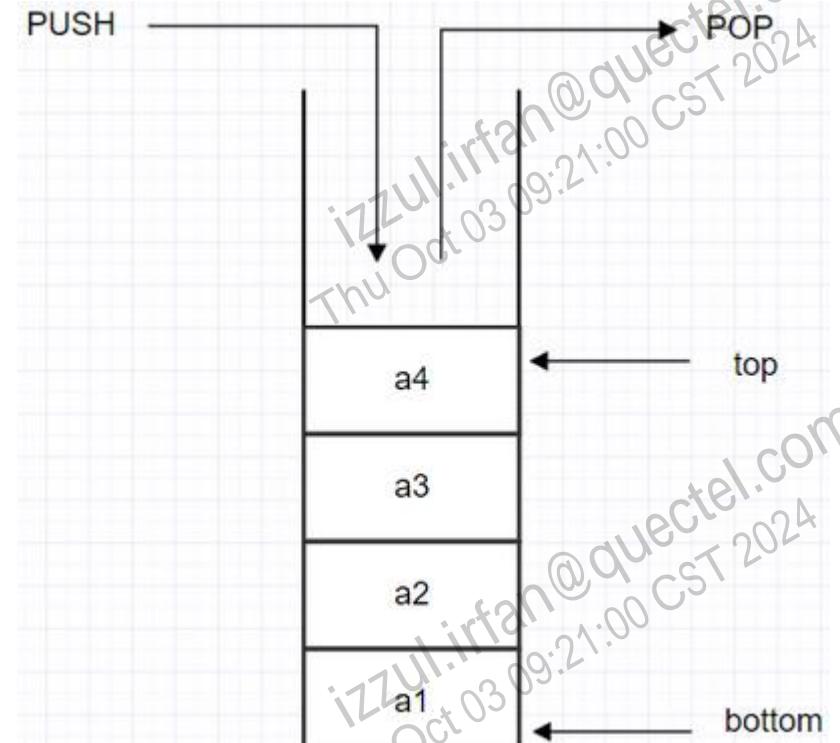
```
void SpecifyLocationToDelete(struct doubleCircularLinkedList* headNode,int posData)
{
    struct doubleCircularLinkedList* posNode = headNode->next;//指定结点指针
    struct doubleCircularLinkedList* posNodeprior = headNode;//指定结点前一个
    结点的指针
    //找到指定位置
    while(posNode->data != posData)
    {
        posNodeprior = posNode;
        posNode = posNodeprior->next;
        //如果没有找到特殊处理
        if(posNode->next == headNode)
        {
            printf("不存在指定位置, 无法删除! \n");
            return;
        }
    }
    posNodeprior->next = posNode->next;
    posNode->next->prior=posNodeprior;
    free(posNode); //删除之后, 释放空间
}
```

```
//查找指定元素
void modifySpecifiedElement(struct doubleCircularLinkedList* headNode,int posData,int elem)
{
    struct doubleCircularLinkedList* posNode = headNode->next;//指定结点指针
    struct doubleCircularLinkedList* posNodeprior = headNode;//指定结点前一个结点的
    指针
    //找到指定位置
    while(posNode->data != posData)
    {
        posNodeprior = posNode;
        posNode = posNodeprior->next;
        //如果没有找到特殊处理
        if(posNode->next == headNode)
        {
            printf("不存在元素! \n");
        }
    }
    posNode->data = elem;
    printf("修改成功! \n");
}
```

数据结构

栈----栈是一种后进先出的线性表，是最基本的一种数据结构，在许多地方都有应用

栈是限制插入和删除只能在一个位置上进行的线性表。其中，允许插入和删除的一端位于表的末端，叫做栈顶（top），不允许插入和删除的另一端叫做栈底（bottom）。对栈的基本操作有 **PUSH**（压栈）和 **POP**（出栈），前者相当于表的插入操作（向栈顶插入一个元素），后者则是删除操作（删除一个栈顶元素）。栈是一种后进先出（LIFO）的数据结构，最先被删除的是最近压栈的元素。栈就像是一个箱子，往里面放入一个小盒子就相当于压栈操作，往里面取出一个小盒子就是出栈操作，取盒子的时候，最后放进去的盒子会最先被取出来，最先放进去的盒子会最后被取出来，这即是后入先出。下面是一个栈的示意图：



数据结构

```
//定义节点
typedef struct Node{
    Elementtype Element;
    struct Node *next;
}NODE,*PNODE;

//定义栈的结构体
typedef struct Stack{
    PNODE PTOP;
    PNODE PBOOTOM;
}STACK,* PSTACK;
```

```
void InitStack(PSTACK Stack)
{
    PNODE PNew = (PNODE)malloc(sizeof(NODE));
    if(PNew == NULL){
        printf("新结点空间的分配失败! ");
        exit(-1);
    }

    Stack->PTOP = PNew;
    Stack->PBOOTOM = PNew;
    PNew->next = NULL;
    printf("栈创建成功! \n");
}
```

数据结构

```
void PushStack(PSTACK Stack,int val)
{
    PNODE P = (PNODE)malloc(sizeof(NODE));
    if(P == NULL){
        printf("新结点空间的分配失败! ");
        exit(-1);
    }
    P->Element=val; //将值赋给结点的赋值域
    P->next = Stack->PTOP; //使新建的结点指向下一个结点
    Stack->PTOP = P; //更新栈顶元素，使其指向新的结点
}
```

```
void PopStack(PSTACK Stack,int *val)
{
    if(Stack->PBOOTOM == Stack->PTOP)
    {
        printf("栈已空");
    }
    PNODE P = Stack->PTOP;
    *val = P->Element;
    Stack->PTOP = P->next;
    free(P);
    P = NULL;
    printf("%d 已出栈!\n",*val);
}
```

数据结构

队列，和栈一样，也是一种对数据的“存”和“取”有严格要求的线性存储结构。作业：

与栈结构不同的是，队列的两端都“开口”，要求数据只能从一端进，从另一端出，如图 1 所示：



图 1 队列存储结构

通常，称进数据的一端为“队尾”，出数据的一端为“队头”，数据元素进队列的过程称为“入队”，出队列的过程称为“出队”。

不仅如此，队列中数据的进出要遵循“先进先出”的原则，即最先进队列的数据元素，同样要最先出队列。拿图 1 中的队列来说，从数据在队列中的存储状态可以分析出，元素 1 最先进队，其次是元素 2，最后是元素 3。此时如果将元素 3 出队，根据队列“先进先出”的特点，元素 1 要先出队列，元素 2 再出队列，最后才轮到元素 3 出队列。

栈和队列不要混淆，栈结构是一端封口，特点是“先进后出”；而队列的两端全是开口，特点是“先进先出”。因此，数据从表的一端进，从另一端出，且遵循“先进先出”原则的线性存储结构就是队列。

数据结构

队列,

```
#define max 5//表示顺序表申请的空间大小
int enQueue(int *a,int front,int rear,int data){
    //添加判断语句, 如果rear超过max, 则直接将其从a[0]重新开始存储, 如果rear+1和front重合, 则表示数组已满
    if ((rear+1)%max==front) {
        printf("空间已满");
        return rear;
    }
    a[rear%max]=data;
    rear++;
    return rear;
}
int deQueue(int *a,int front,int rear){
    //如果front==rear, 表示队列为空
    if(front==rear%max) {
        printf("队列为空");
        return front;
    }
    printf("%d ",a[front]);
    //front不再直接 +1, 而是+1后同max进行比较, 如果=max, 则直接跳转到 a[0]
    front=(front+1)%max;
    return front;
}
```

思考?

链表实现队列

数据结构



数组排序----选择排序

选择排序(Selection-sort)是一种简单直观的排序算法。它的工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

```
void selectionSort(int arr[],int n) {  
    int len = n;  
    int minIndex, temp;  
    for (int i = 0; i < len - 1; i++) {  
        minIndex = i;  
        for (var j = i + 1; j < len; j++) {  
            if (arr[j] < arr[minIndex]) { // 寻找最小的数  
                minIndex = j;           // 将最小数的索引保存  
            }  
        }  
        temp = arr[i];  
        arr[i] = arr[minIndex];  
        arr[minIndex] = temp;  
    }  
}
```

表现最稳定的排序算法之一，因为无论什么数据进去都是 $O(n^2)$ 的时间复杂度，所以用到它的时候，数据规模越小越好。唯一的好处可能就是不占用额外的内存空间了吧。理论上讲，选择排序可能也是平时排序一般人想到的最多的排序方法了吧。

数组排序----冒泡排序

冒泡排序是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果它们的顺序错误就把它们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

```
void bubbleSort(int arr[],int n) {  
    int len = n;  
    for (int i = 0; i < len - 1; i++) {  
        for (int j = 0; j < len - 1 - i; j++) {  
            if (arr[j] > arr[j+1]) {      // 相邻元素两两对比  
                int temp = arr[j+1];    // 元素交换  
                arr[j+1] = arr[j];  
                arr[j] = temp;  
            }  
        }  
    }  
}
```

数组排序----插入排序

插入排序 (Insertion-Sort) 的算法描述是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。

```
void insertionSort(int arr[],int n)
    int len = n;
    int preIndex, current;
    for (int i = 1; i < len; i++) {
        preIndex = i - 1;
        current = arr[i];
        while (preIndex >= 0 && arr[preIndex] > current) {
            arr[preIndex + 1] = arr[preIndex];
            preIndex--;
        }
        arr[preIndex + 1] = current;
    }
}
```

算法分析

插入排序在实现上，通常采用in-place排序（即只需要到 $O(1)$ 的额外空间的排序），因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

数据结构



数组排序----快速排序

快速排序是对冒泡法排序的一种改进。

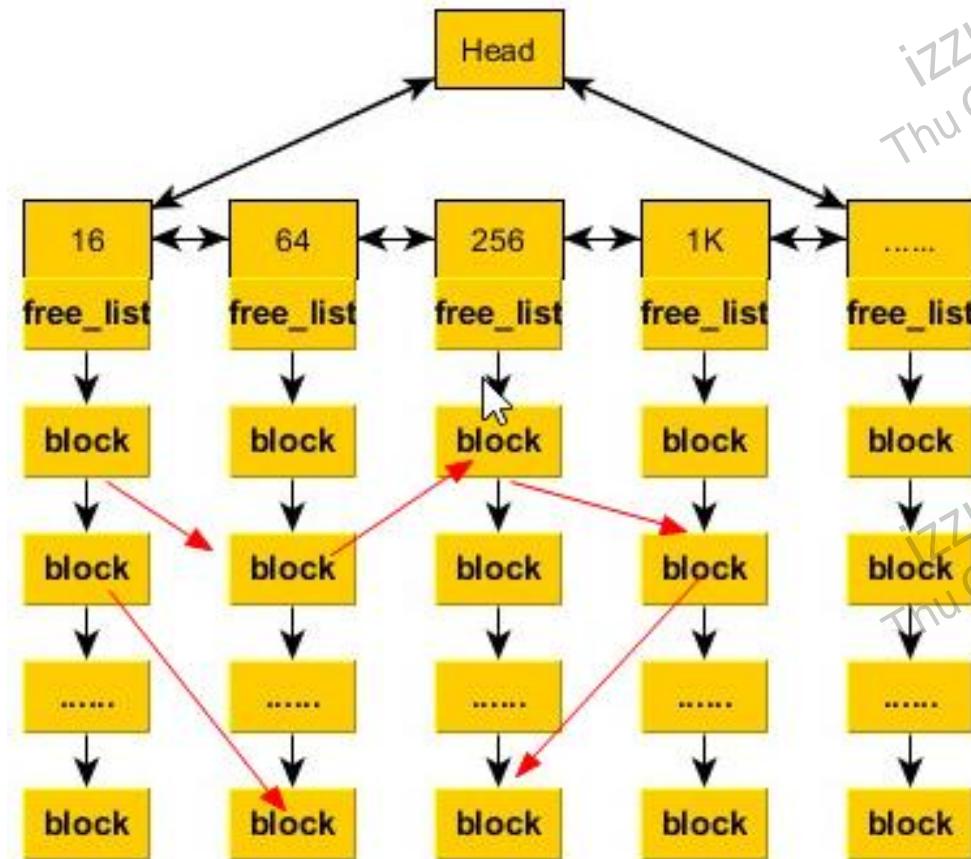
快速排序算法的基本思想：定义一个变量将数组分为左右两个部分，左边的数据都比该变量小，右边的数据都比该变量大，然后将所分得的两部分数据进行同样的划分，重复执行以上的划分操作，直到所有要进行排序的数据变为有序为止。

```
//对数组进行划分
int partition(int arr[], int low, int high){
    int key;
    key = arr[low];
    while(low<high){
        while(low < high && arr[high]>= key ) high--;
        if(low<high) arr[low++] = arr[high];
        while( low<high && arr[low]<=key ) low++;
        if(low<high) arr[high--] = arr[low];
    }
    arr[low] = key;
    return low;
}
```

```
/*
 * 快速排序数组
 * @param int arr[] 要排序的数组
 * @param int start 序列的起始元素
 * @param int end 序列的最后一个元素
 */
void quick_sort(int arr[], int start, int end){
    int pos;
    if (start<end){
        pos = partition(arr, start, end);
        quick_sort(arr,start,pos-1);
        quick_sort(arr,pos+1,end);
    }
    return;
}
```

数据结构

链表实现内存管理，实现自己的malloc和free



课后作业

7.1、函数递归与栈

采用非递归的方式实现二叉树的先序遍历

7.2、采用栈的方式实现10进制数转化为2进制，并使用putchar()显示出来
十进制和其他进制d转换原理：

$$N = (N/d) * d + N \% d;$$

$$(1348)_{10} = (2504)_8$$



答疑



笔试



移联万物，志高行远

- 拥有行业最丰富、完整的产品线，一站式满足各种场景、各个地区的需求
- 采用全自动生产线、测试线，保障产品质量始终如一，同时具有超高性价比
- 建立了业内规模最大的研发团队，为客户提供及时、专业、贴心的技术支持服务
- 持续研发新技术、新产品，率先发布5G、NB-IoT、C-V2X等产品

Thank You

Build a Smarter World

全国热线：400 960 7678

上海市闵行区田林路1016号科技绿洲3期（B区）5号楼 200233
联系电话：+86 21 5108 6236 电子邮件：info@quectel.com
技术支持：support@quectel.com



移远微信公众号