C语言编程进阶

—— C语言进阶概述

Programming
Your Future

前言

- 本课程的目标是在学员掌握C语言基本语法知识、计算机相关基础知识并且具备基本编程能力的基础上，力图从内存以及编译的角度对一些C语法规则等进行深入的解释，能够理解C语言发生编译以及运行错误的深层次原因。为学员编写规范、高效、健壮的代码，设计较复杂的算法以及培养编程思想方面打下坚实基础。
- 本课程的学习方法，除了上课学习理论以及练习之外，课下还要多查找资料，多百度，弥补理论方面的不足；多编程、多练习，理论与实践相结合，知行合一；做项目，积累经验，学以致用（教一、做二、用三）；多总结、多交流。

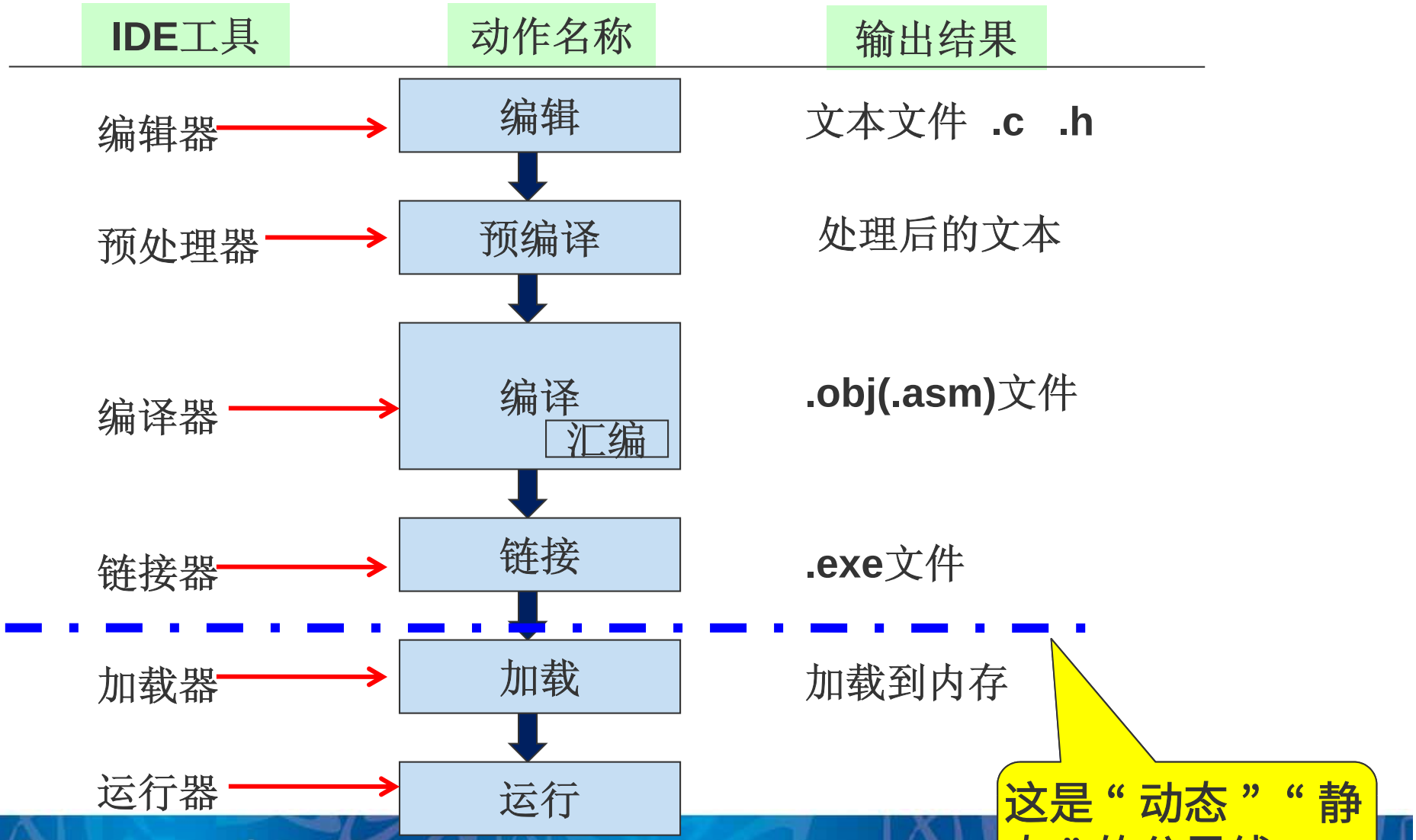
C 语言的历史

- **K. Thompson（UC Berkeley, 从1966开始在BTL工作）**

- **Dennis Ritchie(哈佛应用数学专业，从1968年开始在BTL工作)**

二位大师就是在这台DEC PDP-11中型机上发明了C语言。

C 程序的开发流程


IDE工具
动作名称
输出结果
编辑器
编辑
文本文件 .c .h
预处理器
预编译
处理后的文本
编译器
编译
汇编
.obj(.asm)文件
链接器
链接
.exe文件
加载器
加载
加载到内存
运行器
运行
这是“动态”“静态”的分界线

学习C语言的三个阶段

- 理解C语言的语法，写出至少让编译器通过且不警告的程序——这是大学
- 了解编译器将
  言；
- 形成一种自己
  阶段，才能写出不仅仅正确，还要简洁、清晰、强壮、合乎规范的程序。

一个程序员的成长阶段：
开始：把所有的功能都放在一个函数里；
进步：把不同功能的代码分别放在几个函数里；
成熟：最终学会了用几个不同的文件来构造程序。

另外，不要以编译器来理解语言，各家编译器都有自己的“方言”，要依靠语言标准。唯有真正了解了语言每个概念背后的原理机制，才会驾驭语言且立于不败之地。

C语言的基本元素-变量

变量：在程序运行过程中，其值可以改变的量叫变量。

变量在内存中占一定的存储单元，在存储单元中存放变量的值。

C语言中变量一定要先定义，再使用。且所有的变量定义要放在其作用域的第一条可执行语句之前。

变量定义的一般格式：

数据类型 变量1[= 初值]，变量2[= 初值]，…，变量n [= 初值];

决定了分配的字节数和取值范围。

```
int a = 10；//定义的同时初始化
```

```
int a；//定义时未初始化
a = 10；//赋值
```

变量的特点

申请空间并初始化变量：

int a = 5;

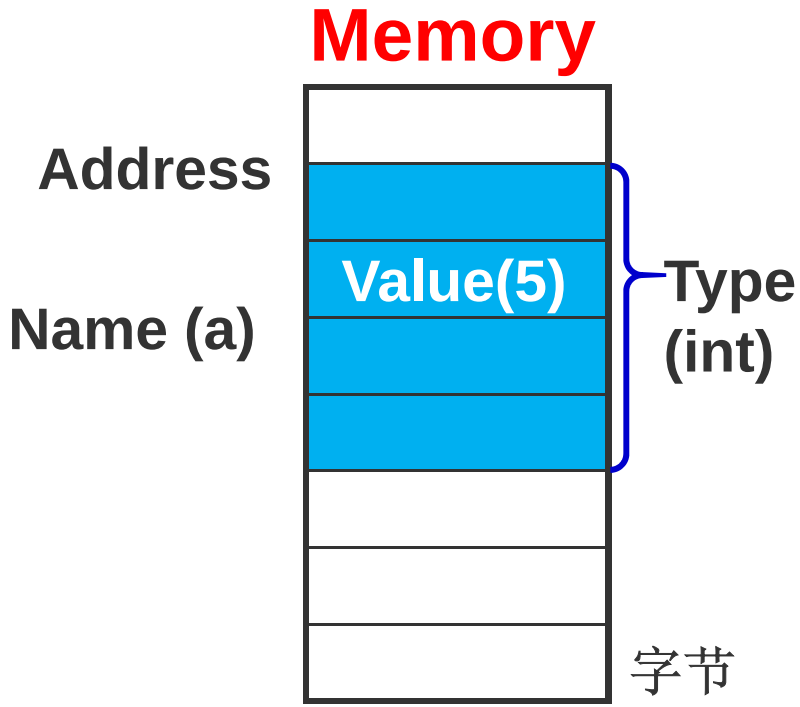
一个变量必然包含以下概念：

名字(变量名)

类型(※)

值(变量的内容)

地址(变量在内存中所处的位置)



变量的初值

```
#include <stdio.h>
int i;
void main( )
{
    static int j;
    int k;
    printf("%d,%d,%d.\n",i, j, k);
}
```

这运行结果说明了什么？

0, 0, -485173492

结论：
当定义变量没有初始化它时，
全局变量、静态变量则由系统初始化为默认值。
而局部变量则不会被初始化，它会是任意值。

变量定义和变量声明

变量声明：仅告诉编译器变量的类型、存储类型，不分配存储空间。

如　**extern int i；**

变量定义：告诉编译器在此处分配存储空间且创建变量。

如　**int i= 1；**

程序中，一个变量只能定义一次，但变量声明可以有多次。

可不可以在头文件中定义变量？

奇怪的运行结果

判读这段代码的运行结果，分析其原因：

```
void main( )
{
        int i = -1;
        unsigned int ui ;
        ui = i;
        printf("%d   %u\n",i,ui);
}
```

发生了隐式类型转换

竟然是：

-1　4294967295

为什么会是这样？

负数的存储形式

C语言对于负数是以“ 补码 ”形式表示的。所谓“ 补码 ”即将一个负数去掉符号后“ 求反加1 ”。

如， int i = 1; 在内存中存储为：

0000 0000 0000 0000 0000 0000 0000 0001	1的原码

int x = -1; 在内存中存储为：

1111 1111 1111 1111 1111 1111 1111 1111	-1的补码

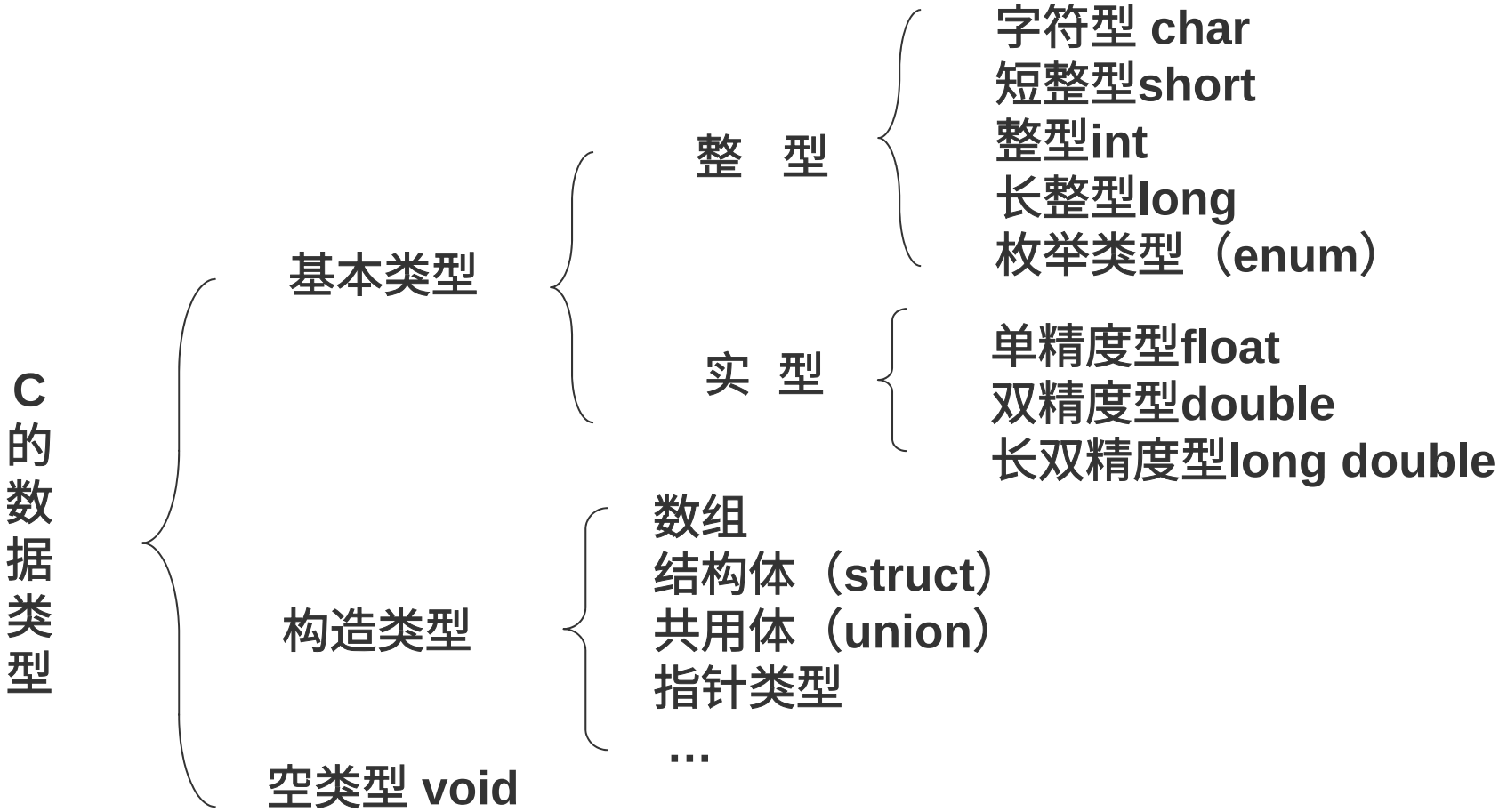
当 unsigned int ui = x; 后，ui的内存视图为：

1111 1111 1111 1111 1111 1111 1111 1111	被视作原码

X的值原封不动的给了ui。

结论：
不同类型的变量赋值时会发生隐式类型转换。这是应当避免的。

数据类型的分类



类型重定义 typedef　　不是独立类型.

数据类型的意义

定义了数据占用的内存空间大小；
定义了数据的取值范围；
定义了数据在内存中的存储格式；
决定了数据的运算行为；

怎么来理解“ 决定

```
void  main()
{
    int  a = -10;
    unsigned b = 
    if( a>b)
        printf("%d
    else
        printf("%d
}
```

你认为结果如何？

怎么来理解“ 为编译器提供了检查依据 ”？

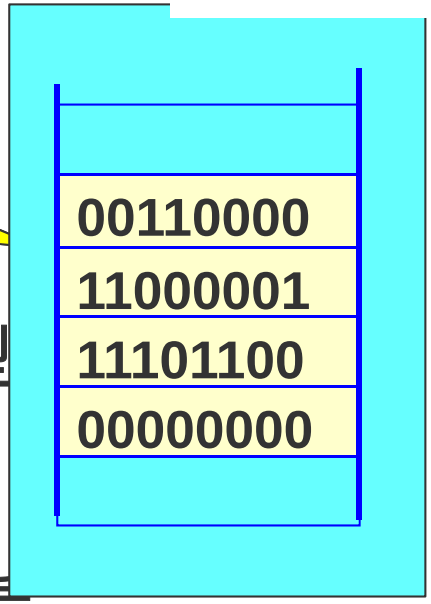
```
void  main()
{
    float  a = -10.3f;
    float  b = 20.2f;
    printf("%d\n", a%b );
    //…
}
```

结果是编译出错！ 因为C语言要求求模运算的两个操作数都必须是整型。发现 a和b是浮点型，不符合类型要求，报错。

同一块内存的不同视角

```
void ma
{
    float f = 3.14f;  //对于一块内存，按浮点型
    int * ip = (int *)(& f);
    char * cp = (char *)(& f);
    printf("the float : %f\n", f); //以浮点视角看
    printf("the int : %d\n", *ip); //以整型视角看
    printf(“the char : %s\n”, cp); //以字符型视角看
```

一块内存，4个字节中存有二进制数据



这个例子说明了类型是解释一串二进制码到底表示何种含义的关键，只有解释才会使之有意义。可见“类型决定了数据在内存中的存储格式”。

运行结果：

the float : 3.140000

the int : 1078523331

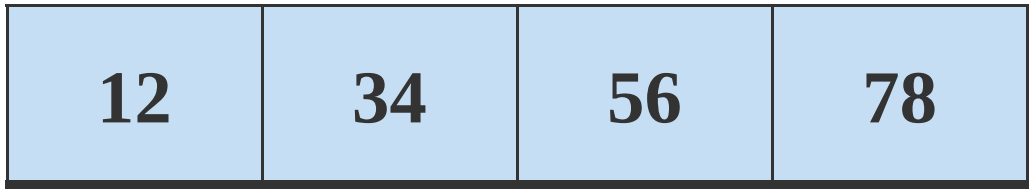
the char : 惋H@?

数据的端序

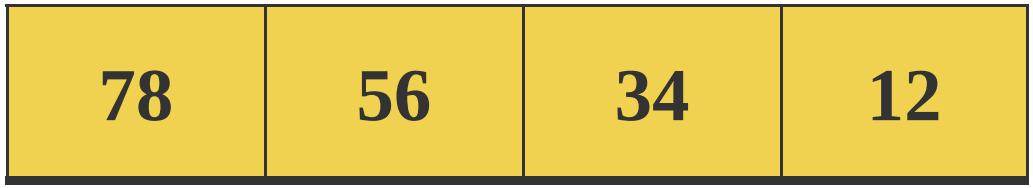
多字节数据在内存中究竟是如何存放的？

如 ：int b = 0x12345678 ;
会有两种存放可能：

小端对齐



大端对齐



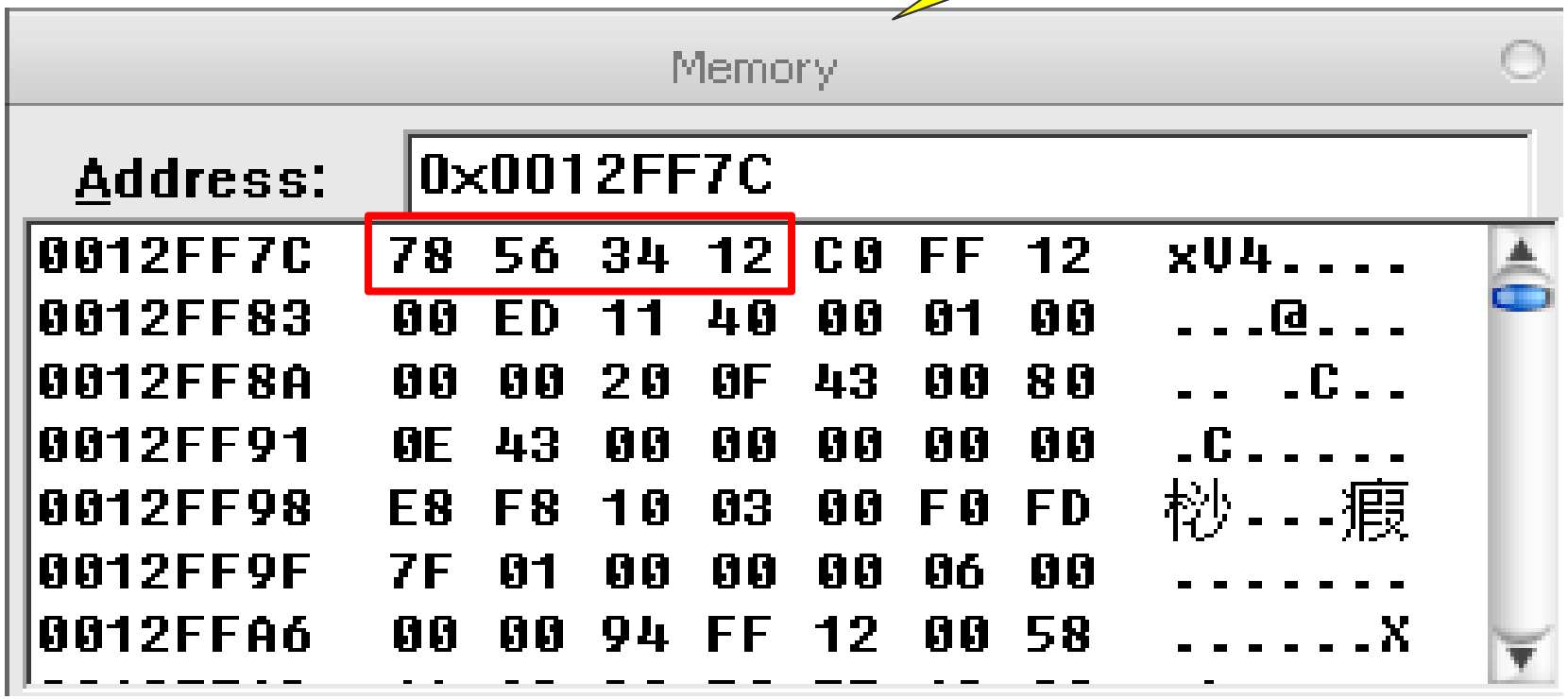
数据对齐的端模式(Endian)与处理器有关。

1. 小端对齐，　　如Intel 80X86系列。
2. 大端对齐，　　如PowerPC处理器。

端序的验证

```
int i = 0x12345678;
printf("%p",&i);
```

先在第2句设置断点，然后使用 F5 观察内存。



如果memory窗口没打开，VC6.0可用ALT+6 打开;VS 2005 要用“调试”->“窗口” ->“内存”观察。

例题

写一个C函数，若处理器是Big_endian，则返回非0值；
　若Little_endian，则返回0。
函数原型：bool IsBigendian()：

```
bool  IsBigendian()
{
    union {
          int i;
          char ch[4];
    } test;
    test.i= 0x12345678;
    return (test.ch[0]!=0x78)；
}
```

给四个字节里放上不同的数

只看低字节里的内容

编译器对字符串常量的处理

C语言将连续的多个字符串视为一个字符串。不管是否用空格、tab或回车隔开。例如：

```
void main( )
{
    char str[ ] = "Neu"   "soft";
    printf("%d\n",sizeof(str));
    printf("%s\n",str);
}
```

其间的空格不影响它们同是一个串。

```
8
Neusoft
```

编译器对字符串常量的处理

“ ”和’ ’的使用在本质上是完全不同的。如：

```
#include <stdio.h>
void main( )  {
    int x = (int)“A”;     // ?
    int z = (int)“B”;     // ?
    int y ='A' ;          // ?
    int *p = &y;
    printf("%x\n",x);
    printf(“%x\n”,z);
    printf("%x\n",p);
}
```

x 记录的是串"A"的首地址:0x00422024

串“B”的首地址：0x00422020

'A'的首地址是：0x0012ff74

逻辑表达式的短路性

逻辑表达式有“短路性 ”，这样做可以提高运行效率。但客观效果会使有些表达式可能没有被执行。

a&&(b= 1)

对于&&，如果a表达式为假，则不再计算b=1；

a||(b= 1)

对于||，如果a表达式为真，则不再计算b=1.

条件表达式的短路性

条件表达式也有“短路性”。看下面例题：

```
#include <stdio.h>
int main()  {
    int a =0;
    int b = 1;
    int c;
    c = a > b ? ++a : ++b;
    printf(“%d, %d, %d\n”, a, b, c);   // 结果如何？
    return 0;
}
```

是 0,2,2。

首先要弄清运算次序。尽管自增运算的优先级高于三目运算，但是按前面“条件运算符的结合方向”的讨论结果， ++a和++b仅是三目运算表达式的组成部分，应先算a > b，根据其真假来决定是计算 ++a还是 ++b。于是最后结果是 0, 2, 2

编译器对sizeof的理解

```
int main()
{
    int i = 10;
    printf("i : %d\n",i);
    printf("sizeof(i++) is: %d\n",sizeof(i++));
    printf("i : %d\n",i);
    return 0;
}
```

你认为输出的是：
10
4
11
其实不是！

其实结果是：
10
4
10
为什么？

原因是，**sizeof**不是一个函数，是个操作符，仅仅求**i++**表达式的类型尺寸，而不计算表达式的值。
这是一件可以在程序运行前（编译时）完成的事，所以，**sizeof(i++)**直接就被**4**取代了，并不计算**i++**，在运行时也就不会有 **i++**这个表达式。

typedef

功能：用自定义的类型名为已有数据类型命名。

一般形式： typedef type NAME;

例如：

```
typedef int INTEGER;
typedef float REAL;
INTEGER a,b,c;
REAL f1,f2;
```

能隐藏笨拙的语法构造以及平台相关的数据类型，从而增强可移植性和未来的可维护性，简化代码。但降低了代码的可读性。

说明:

1. typedef 没有创造新数据类型；
2. typedef 是专用于定义类型的，不能定义变量；
3. typedef 与 define 不同。 （见下页）

typedef与 define 区别

define 预编译时处理，简单字符置换。
typedef 编译时处理，为已有类型命名 。

```
#define  dpchar  char *
dpchar  p1, p2;

typedef  char *  tpchar;
tpchar  p3, p4;
```

二者的使用一样效果吗？

p1是指针而p2不是指针，仅是个char型的普通变量。
p3、p4则不同，它们都是char型的指针。

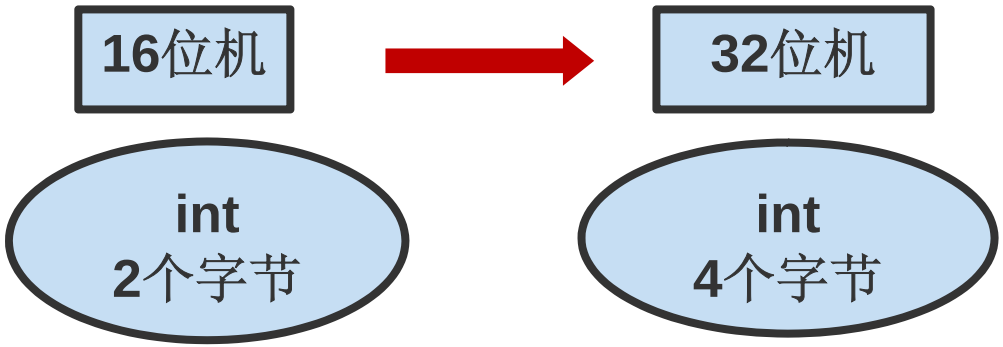
typedef的用处

1、增强代码的可移植性。

例如：对于16位机器： typedef short INT16;

INT16 num1; //使用新类型名

INT16 num2; … INT16 numn;



在16位机上，如果不使用typedef，而是直接用int定义变量，则移植到32位机平台时，需要将代码中所有的int改为short，改动量很大，而且容易遗漏。

typedef的用处

2、简化代码。

例如，用于重定义struct，union等类型：

```
typedef  struct  student
{     int  num;
      int    age;
} STU;
```

这时使用：

```
STU  stu1;   // 相当于struct  student  stu1;
```

3、还可以掩饰复合类型，如指针和数组 ：

```
typedef  char UINT8[32];
```

其中 UINT8 等价于 char [32]。

如 UINT8 devString; == char devString[32];

typedef的用处

4、还可以掩饰函数指针

如： typedef int (* pFun)(int);

pFun p;

其中 pFun p; == int (*p)(int);

例：

```
typedef int* (* pt )(int* (*pn)(int * p1,int *p2),int * p3);
 pt p;
```

常能见到一些程序里出现size_t类型，这其实是在<stddef.h>头文件里定义的unsigned int类型的别名。另外还有 ptrdiff_t类型，它也是标准库定义的类型名，用于表示两个指针之差。是 int 或long。

typedef的用处

5、这里的 typedef 掩饰了一个结构体指针

```
typedef struct student
{
        int i;
        char name[10];
}*pSTU;
```

pSTU是什么？如何理解?

把“ struct student { /*code*/} *”看成一个整体，typedef就是给“ struct student {/*code*/} * ”取了个别名叫“ pSTU”。

这种技巧在系统头文件中常被使用！

typedef的其它用法

再看typedef的如下的用法：

```
typedef   int   ARRAY[10];
ARRAY   a;
ARRAY   b[5];
ARRAY   *c;
```

ARRAY 所代表的类型是什么？

int [10]

int a[10];

int b[5][10];

int (*c)[10];

typedef可以定义多个类型名

```
typedef  struct  student
{
   int i;
   char name[10];
}STU，*pSTU;
```

这里 struct student、 STU、pSTU 都是类型名。
这种技巧在实际开发中经常被使用！
现在我们来判断 s1,s2 各是什么？如何理解?

```
STU  s1;
pSTU  s2 ;
```

它是struct student类型的变量。

它是struct student*类型的变量，即s2 是个指针。

typedef的错误使用

typedef static int STINT;

该语句编译错误：

error C2159: more than one storage class specified

分析：

typedef 与 static ,auto, extern, register等关键字被看做同是存储类型。

C语言规定，存储类型每句只能使用一个。编译不通过的原因是因为声明中存在了多个存储类关键字。

编程练习

中国有句俗语叫“三天打鱼两天晒网”。某人从**1990**年**1**月**1**日起开始“三天打鱼两天晒网”，问这个人在以后的某一天中，是在“打鱼”还是在“晒网”？

问题分析与算法设计：

根据题意可以将解题过程分为四步：

1）屏幕提示，输入日期值，判是否合理；

2）计算从1990年1月1日开始至指定日期共有多少天；

3）由于“打鱼”和“晒网”的周期为5天，所以将计算出的天数用5求余；

4）根据余数判断他是在“打鱼”还是在“晒网”；

若余数为1,2,3，则他是在“打鱼”，否则是在“晒网”。

在这四步中，关键是第二步。求从1990年1月1日至指定日期之间共有多少天，还要判断经历年份中是否有闰年，二月为29天，平年为28天。

闰年的方法可以用如下伪语句描述：

如果　((年能被4除尽 且 不能被100除尽)或 能被400除尽)

　　则　　该年是闰年；

　　否则　　不是闰年。


谢谢！
Neusoft
Programming Your Future