

c语言概述

1. 为什么学习C语言
 - 1). C的起源和发展(ppt)
 - 2). C的特点

优点

代码量小 速度快 功能强大

缺点

危险性高

开发周期长

可移植性不强

- 3). C的应用领域

- 4). C的重要性

2. 怎样学习C语言

3. 学习的目标

4. 常见问题答疑

1. 学习java为什么建议先学C语言
2. 没学过计算机专业课程能够学懂C语言
3. 英语和数学不好能学好C么？

5. 课程计划

6. 举例子：一元二次方程

1> C编程预备计算机专业知识

1. cpu 内存条 硬盘 显卡 主板 显示器 之间的关系

2. HelloWorld程序如何运行起来的

3. 什么是数据类型

基本类型数据

整数

整型	—	int	—4
短整型	—	short int	—2
长整型	—	long int	—8

浮点数【实数】

单精度浮点数	—	float	—4
双精度浮点数	—	double	—8

字符

char	—1
------	----

复合类型数据

结构体
枚举
共用体

4. 什么是变量

变量的本质就是内存中一段存储空间

5. CPU 内存条 VC++6.0 操作系统 之间的关系

6. 变量为什么必须的初始化

所谓初始化就是赋值的意思

7. 如何定义变量

数据类型 变量名 = 要赋的值;

等价于

数据类型 变量名;

变量名 = 要赋的值;

举例子:

```
int i = 3; 等价于 int i; i = 3;  
int i, j; 等价于 int i; int j;  
int i, j = 3; 等价于 int i; int j; j = 3;  
int i = 3, j = 5; 等价于 int i; int j; i=3; j = 5;  
int i, j; i = j = 5; 等价于 int i, j; i = 5; j = 5;
```

8. 什么是进制 [ppt]

十进制就是逢十进一

二进制逢二进一

9. 常量在C语言中是如何表示的

整数

十进制: 传统的写法

十六进制: 前面加0x或0X

八进制: 前面0 注意是数字零不是字母o

浮点数

传统的写法

```
float x = 3.2; //传统
```

科学计数法

```
float x = 3.2e3; //x的值是 3200
```

```
float x = 123.45e-2; //x的值是1.2345
```

字符

单个字符用单引号括起来

'A' 表示字符A

'AB' 错误

"AB" 正确

字符串用双引号括起来

"A" 正确, 因为"A"代表了'A' '\0' 的组合

10. 常量以什么样的二进制代码存储在计算机中

整数是以补码的形式转化为二进制代码存储在计算机中的
实数是以IEEE754标准转化为二进制代码存储在计算机中的

具体可参见末尾的 [穿插在课堂中的零散知识笔记——浮点数的存错所带来的问题](#)

字符的本质实际也是与整数的存储方式相同

11. 代码规范化

代码的可读性更强[容易让自己和别人更清楚的看懂程序]
使程序不容易出错

12. 什么是字节

字节就是存储数据的单位，并且是硬件所能访问的最小单位

1字节 = 8位

1K = 1024字节

1M = 1024k

1G = 1024M

13. 不同类型数据之间相互赋值的问题

暂不考虑

```
int i = 45;
long j = 102345;
i = j;
printf("%ld %d\n", i, j);
float x = 6.6;
double y = 8.8;
printf("%f %lf\n", x, y);
```

14. 什么是ASCII

ASCII不是一个值，而是一种规定，

ASCII规定了不同的字符是使用哪个整数值去表示
它规定了

'A'	—	65
'B'	—	66
'a'	—	97
'b'	—	98
'0'	—	48

15. 字符的存储[字符本质上与整数的存储方式相同]

2> 基本的输入和输出函数的用法

printf() —将变量的内容输出到显示器上【重点】

四种用法

1. printf("字符串");
2. printf("输出控制符", 输出参数);
3. printf("输出控制符1 输出控制符2。 。 。", 输出参数1, 输出参数2,); ;
 输出控制符和输出参数的个数必须一一对应
4. printf("输出控制符 非输出控制符", 输出参数);

输出控制符包含如下

%d	—	int
%ld	—	long int
%c	—	char
%f	—	float
%lf	—	double
%x(或者%X后者%#X)	—	int 或 long int 或 short int
%o	—	同上
%s	—	字符串

为什么需要输出控制符

1. 01组成的代码可以表示数据也可以表示指令
2. 如果01组成的代码表示的是数据的话，那么同样的01代码组合以不同的输出格式输出就会有不同的输出结果

scanf() 【通过键盘将数据输入到变量中】

两种用法：

用法一： scanf("输入控制符", 输入参数);

功能： 将从键盘输入的字符转化为输入控制符所规定格式的数据，然后存入以输入参数的值为地址的变量中

用法二： scanf("非输入控制符 输入控制符", 输入参数);

功能： 将从键盘输入的字符转化为输入控制符所规定格式的数据，然后存入以输入参数的值为地址的变量中

非输入控制符必须原样输入

如何使用scanf编写出高质量代码

1. 使用scanf之前最好先使用printf提示用户以什么样的方式来输入
2. scanf中尽量不要使用非输入控制符，尤其是不要用\n
3. 应该编写代码对用户的非法输入做适当的处理【非重点】

```
char ch;
while ( (ch=getchar()) != '\n')
    continue;
```

3> 运算符

算术运算符

+ - * / (除) % (取余数)

关系运算符

> >= < <= !=(不等于) == (等于)

逻辑运算符

!(非) &&(并且) ||(或)

!真	假
! 假	真

真&&真 真

真&&假 假

假&&真	假
假&&假	假
真 假	真
假 真	真
真 真	真
假 假	假

C语言对真假的处理

非零是真
零是假

真是1表示
假是0表示

&&左边的表达式为假 右边的表达式肯定不会执行
||左边的表达式为真 右边的表达式肯定不会执行

赋值运算符

= += *= /= -=

优先级别：

算术 > 关系 > 逻辑 > 赋值

附录的一些琐碎的运算符知识

自增 自减 三目运算符 逗号表达式

4> 流程控制【是我们学习C语言的第一个重点，要求所有的知识全部掌握】

1. 什么是流程控制

程序代码执行的顺序

2. 流程控制的分类

顺序

选择

定义

某些代码可能执行，也可能不执行，有选择的执行某些代码

分类

if

1. if最简单的用法

格式：

if (表达式)
语句

功能：

如果表达式为真，执行语句
如果表达式为假，语句不执行

2. if的范围问题

1.

```
if (表达式)
    语句A;
    语句B;
```

解释：if默认只能控制语句A的执行或不执行

if无法控制语句B的执行或不执行

或者讲：语句B一定会执行

2.

```
if (表达式)
{
    语句A;
    语句B;
}
```

此时if可以控制语句A和语句B

由此可见：if默认只能控制一个语句的执行或不执行

如果想控制多个语句的执行或不执行

就必须把这些语句用{}括起来

3. if..else...的用法

4. if..else if...else...的用法

格式：

```
if (1)
    A;
else if (2)
    B;
else if (3)
    C;
else
    D;
```

5. C语言对真假的处理

非零是真

零就是假

真用1表示

假用零表示

6. if举例—求分数的等级

7. if的常见问题解析

1>. 空语句的问题

if (3 > 2);

等价于

if (3 > 2)
 ; //这是一个空语句

2>.

if (表达式1)
 A;

```
else  
    B;  
是正确的
```

```
if (表达式1);  
    A;  
else  
    B;  
是错误的
```

3>.

```
if (表达式1)  
    A;  
else if (表达式2)  
    B;  
else if (表达式3)  
    C;  
else  
    D;
```

即便表达式1和2都成立，也只会执行A语句

4>.

```
if (表达式1)  
    A;  
else if (表达式2)  
    B;  
else if (表达式3)  
    C;
```

这样写语法不会出错，但逻辑上有漏洞

5>.

```
if (表达式1)  
    A;  
else if (表达式2)  
    B;  
else if (表达式3)  
    C;  
else (表达式4) //7行  
    D;
```

这样写是不对的，正确的写法是：

要么去掉7行的(表达式4)
要么在7行的else 后面加if

6>.

```
if (表达式1)  
    A;  
else if (表达式2)  
    B;
```

```
else if (表达式1)
    C;
else (表达式4);
    D;

这样写语法不会出错，但逻辑上是错误的
else (表达式4);
    D;
等价于
else
    (表达式4);
    D;

switch
    把电梯程序看懂就OK了
```

循环

定义：

某些代码会被重复执行

分类

for

1. 格式：

```
for (1; 2; 3)
    语句A;
```

2. 执行的流程【重点】

单个for循环的使用

多个for循环的嵌套使用

1.

```
for (1; 2; 3) //1
    for (4; 5; 6) //2
        A; //3
        B; //4
```

整体是两个语句， 1 2 3 是第一个语句

4 是第二个语句

2.

```
for (1; 2; 3)
    for (4; 5; 6)
    {
        A;
        B;
    }
```

整体是一个语句

3.

```
for (7; 8; 9)
    for (1; 2; 3)
    {
        A;
```

```
B;  
for (4; 5; 6)  
C;  
}
```

整体是一个语句

3. 范围问题

4. 举例：

```
1 + 2 + 3 + .... + 100  
1 + 1/2 + 1/3 + .... + 1/100
```

【本程序对初学者而言很重要，具体细节可参见我录制的相关视频】

`while`

1. 执行顺序

格式：

```
while (表达式)  
语句;
```

2. 与`for`的相互比较

`for`和`while`可以相互转换

```
for (1; 2; 3)  
A;
```

等价于

```
1;  
while (2)  
{  
    A;  
    3;  
}
```

`while`和`for`可以相互转化

但`for`的逻辑性更强，更不容易出错，推荐多使用`for`

3. 举例

从键盘输入一个数字，如果该数字是回文数，
则返回yes，否则返回no

回文数：正着写和倒着写都一样

比如：121 12321 都是回文数

4. 什么时候使用`while`，什么时候使用`for`

没法说，用多了自然而然就知道了

`do...while`

格式

```
do  
{  
    ....
```

```
    } while (表达式);
do...while. 并不等价于for, 当然也不等价于while
主要用于人机交互
一元二次方程
```

break和continue

break

break如果用于循环是用来终止循环

break如果用于switch, 则是用于终止switch

break不能直接用于if, 除非if属于循环内部的一个子句

例子:

```
for (i=0; i<3; ++i )
{
    if (3 > 2)
        break; //break虽然是if内部的语句,
                //但break终止的确是外部的for循环
    printf("嘿嘿!\n"); //永远不会输出
}
```

在多层循环中, break只能终止最里面包裹的那个循环

例子:

```
for (i=0; i<3; ++i)
{
    for (j=1; j<4; ++j)
        break; //break只能终止距离它最近的循环
    printf("同志们好!\n");
}
```

在多层switch嵌套中, break只能终止距离它最近的switch

例子:

```
int x=1, y=0, a=0, b=0;
switch(x) // 第一个switch
{
case 1:
    switch(y) // 第二个switch
    {
case 0:
    a++;
    break; //终止的是第二个switch
case 1:
    b++;
    break;
}
b = 100;
break; //终止的是第一个switch
case 2:
    a++;
    b++;
    break;
```

```

    }
    printf("%d %d\n", a, b); //26行
    最终输出结果是: 1 100

continue
用于跳过本次循环余下的语句,
转去判断是否需要执行下次循环
例子:
1.
for (1; 2; 3)
{
    A;
    B;
    continue; //如果执行该语句, 则执行完该语句后, 会执行语句3, C和D都会被跳过去, C
    和D不会被执行
    C;
    D;
}

2.
while (表达式)
{
    A;
    B;
    continue; //如果执行该语句, 则执行完该语句后,
               //会执行表达式, C和D都会被跳过去, C和D不会被执行
    C;
    D;
}

```

5> 数组

为什么需要数组

为了解决大量同类型数据的存储和使用问题

为了模拟现实世界

数组的分类

一维数组

怎样定义一维数组

为n个变量连续分配存储空间

所有的变量数据类型必须相同

所有变量所占的字节大小必须相等

例子:

int a[5];

一维数组名不代表数组中所有的元素,

一维数组名代表数组第一个元素的地址

有关一维数组的操作

初始化

完全初始化

```

int a[5] = {1, 2, 3, 4, 5};
不完全初始化，未被初始化的元素自动为零
int a[5] = {1, 2, 3};
不初始化，所有元素是垃圾值
int a[5];
清零
int a[5] = {0};
错误写法：
int a[5];
a[5] = {1, 2, 3, 4, 5}; //错误
只有在定义数组的同时才可以整体赋值,
其他情况下整体赋值都是错误的

int a[5] = {1, 2, 3, 4, 5};
a[5] = 100; //error 因为没有a[5]这个元素，最大只有a[4]

```

```

int a[5] = {1, 2, 3, 4, 5};
int b[5];
如果要把a数组中的值全部复制给b数组
错误的写法：
b = a; // error
正确的写法
for (i=0; i<5; ++i)
    b[i] = a[i];

```

赋值
排序
求最大/小值
倒置
查找
插入
删除

二维数组

```

int a[3][4];
总共是12个元素，可以当做3行4列看待，这12个元素的名字依次是
a[0][0] a[0][1] a[0][2] a[0][3]
a[1][0] a[1][1] a[1][2] a[1][3]
a[2][0] a[2][1] a[2][2] a[2][3]
a[i][j] 表示第i+1行第j+1列的元素
int a[m][n]; 该二维数组右下角位置的元素只能是a[m-1][n-1]

```

初始化

```

int a[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
int a[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
}

```

```
{9, 10, 11, 12}  
};
```

操作

输出二维数组内容:

```
int a[3][4] = {  
    {1, 2, 3, 4},  
    {5, 6, 7, 8},  
    {9, 10, 11, 12}  
};  
int i, j;  
  
//输出数组内容  
for (i=0; i<3; ++i)  
{  
    for (j=0; j<4; ++j)  
        printf("%d ", a[i][j]);  
    printf("\n");  
}
```

对二维数组排序

求每一行的最大值

判断矩阵是否对称

矩阵的相乘

多维数组

是否存在多维数组

不存在

因为内存是线性一维的

n维数组可以当做每个元素是n-1维数组的一维数组

比如:

```
int a[3][4];  
    该数组是含有3个元素的一维数组  
    只不过每个元素都可以再分成4个小元素  
int a[3][4][5];  
    该数组是含有3个元素的一维数组  
    只不过每个元素都是4行5列的二维数组
```

6> 函数【C语言的第二个重点】:

为什么需要函数

避免了重复性操作

有利于程序的模块化

什么叫函数

逻辑上: 能够完成特定功能的独立的代码块

物理上:

能够接收数据 [当然也可以不接受数据]

能够对接受的数据进行处理

能够将数据处理的结果返回[当然也可以不返回任何值]
总结： 函数是个工具，它是为了解决大量类似问题而设计的
函数可以当做一个黑匣子

如何定义函数

函数的返回值 函数的名字(函数的形参列表)

```
{  
    函数的执行体  
}
```

1. 函数定义的本质是详细描述函数之所以能够实现某个特定功能的具体方法

2. `return` 表达式；的含义：

- 1> 终止被调函数，向主调函数返回表达式的值
- 2> 如果表达式为空，则只终止函数，不向主调函数返回任何值
- 3> `break`是用来终止循环和`switch`的，`return`是用来终止函数的

例子：

```
void f()  
{  
    return; //return只用来终止函数，不向主调函数返回任何值  
}  
  
int f()  
{  
    return 10; //第一： 终止函数， 第二： 向主调函数返回10  
}
```

3. 函数返回值的类型也称为函数的类型，因为如果 函数名前的返回值类型 和 函数执行体中的 `return` 表达式；中表达式的类型不同的话，则 最终函数返回值的类型 以函数名前的返回值类型为准

例子：

```
int f()  
{  
    return 10.5; //因为函数的返回值类型是int  
                //所以最终f返回的是10而不是10.5  
}
```

函数的分类

有参函数 和 无参函数

有返回值函数 和 无返回值函数

库函数 和 用户自定函数

值传递函数 和 地址传递函数

普通函数 和 主函数(`main`函数)

一个程序必须有且只能有一个主函数

主函数可以调用普通函数 普通函数不能调用主函数

普通函数可以相互调用

主函数是程序的入口，也是程序的出口

注意的问题

函数调用和函数定义的顺序

如果函数调用写在了函数定义的前面，则必须加函数前置声明

函数前置声明：

1. 告诉编译器即将可能出现的若干个字母代表的是一个函数
2. 告诉编译器即将可能出现的若干个字母所代表的函数的形参和返回值的具体情况
3. 函数声明是一个语句，末尾必须加分号
4. 对库函数的声明是通过 `# include <库函数所在的文件的名字.h>` 来实现的

形参和实参

个数相同 位置一一对应 数据类型必须相互兼容

如何在软件开发中合理的设计函数来解决实际问题

一个函数的功能尽量独立，单一

多学习，多模仿牛人的代码

函数是C语言的基本单位，类是Java, C#, C++的基本单位

常用的系统函数

`double sqrt(double x);`

求的x的平方根

`int abs(int x)`

求x的绝对值

`double fabs(double x)`

求x的绝对值

专题：

递归

可以参见我的数据结构视频

7> 变量的作用域和存储方式：

按作用域分：

全局变量

在所有函数外部定义的变量叫全局变量

全局变量使用范围：从定义位置开始到整个程序结束

局部变量

在一个函数内部定义的变量或者函数的形参 都统称为局部变量

```
void f(int i)
{
    int j = 20;
}
```

i和j都属于局部变量

局部变量使用范围：只能在本函数内部使用

注意的问题：

全局变量和局部变量命名冲突的问题

在一个函数内部如果定义的局部变量的名字和全局变量名一样时，局部变量会屏蔽掉全局变量

按变量的存储方式

静态变量

自动变量

寄存器变量

8> 指针 (C语言的第三个重点) :

指针的重要性

表示一些复杂的数据结构

快速的传递数据, 减少了内存的耗用【重点】

使函数返回一个以上的值【重点】

能直接访问硬件

能够方便的处理字符串

是理解面向对象语言中引用的基础

总结: 指针是C语言的灵魂

指针的定义

地址

内存单元的编号

从零开始的非负整数

范围: 4G 【0—4G-1】

指针

指针就是地址, 地址就是指针

指针变量就是存放内存单元编号的变量, 或者说指针变量就是存放地址的变量

指针和指针变量是两个不同的概念

但是要注意: 通常我们叙述时会把指针变量简称为指针, 实际它们含义并不一样

指针的本质就是一个操作受限的非负整数

指针的分类

1. 基本类型指针【重点】

```
int * p; //p是变量的名字, int * 表示p变量存放的是int类型变量的地址
          //int * p; 不表示定义了一个名字叫做*p的变量
          // int * p; 应该这样理解: p是变量名, p变量的数据类型是 int *类型
          // 所谓int * 类型 实际就是存放int变量地址的类型

int i = 3;
int j;
p = &i;
/*
1. p保存了i的地址, 因此p指向i
2. p不是i, i也不是p, 更准确的说: 修改p的值不影响i的值, 修改i的值也不会影响p的值
3. 如果一个指针变量指向了某个普通变量, 则
   *指针变量 就完全等同于 普通变量
```

例子:

如果p是个指针变量, 并且p存放了普通变量i的地址

则p指向了普通变量i

*p 就完全等同于 i

或者说: 在所有出现*p的地方都可以替换成i

在所有出现i的地方都可以替换成*p

*p 最准确的解释是: *p 表示的是以p的内容为地址的变量

*/

```
j = *p; //等价于 j = i;  
printf("i = %d, j = %d\n", i, j);
```

附注：

*的含义

1. 乘法
2. 定义指针变量

```
int * p;  
//定义了一个名字叫p的变量, int *表示p只能存放int变量的地址
```

3. 指针运算符

该运算符放在已经定义好的指针变量的前面

如果p是一个已经定义好的指针变量

则 *p表示 以p的内容为地址的变量

如何通过被调函数修改主调函数普通变量的值

1. 实参必须为该普通变量的地址
2. 形参必须为指针变量
3. 在被调函数中通过
 $*\text{形参名} = \dots$
的方式就可以修改主调函数相关变量的值

2. 指针和数组

指针和一维数组

一维数组名

一维数组名是个指针常量

它存放的是一维数组第一个元素的地址

下标和指针的关系

如果p是个指针变量, 则

$p[i]$ 永远等价于 $*(p+i)$

确定一个一维数组需要几个参数【如果一个函数要处理一个一维数组, 则需要接收该数组的哪些信息】

需要两个参数:

数组第一个元素的地址

数组的长度

指针变量的运算

指针变量不能相加 不能相乘 也不能相除

如果两个指针变量指向的是同一块连续空间中的不同存储单元,
则这两个指针变量才可以相减

一个指针变量到底占几个字节【非重点】

预备知识:

`sizeof(数据类型)`

功能: 返回值就是该数据类型所占的字节数

例子: `sizeof(int) = 4 sizeof(char) = 1`

`sizeof(double) = 8`

`sizeof(变量名)`

功能: 返回值是该变量所占的字节数

假设p指向char类型变量(1个字节)
假设q指向int类型变量(4个字节)
假设r指向double类型变量(8个字节)
请问： p q r 本身所占的字节数是否一样 ?
答案： p q r 本身所占的字节数是一样的

总结：

一个指针变量，无论它指向的变量占几个字节
该指针变量本身只占四个字节

一个变量的地址是用该变量首字节的地址来表示

指针和二维数组

3. 指针和函数

4. 指针和结构体

5. 多级指针

示例：

```
int i = 10;
int * p = &i; //p只能存放int类型变量的地址
int ** q = &p; //q是int **类型， 所谓int **类型就是指q只能存放int *类型变量的地址,
int *** r = &q; //r是int ***类型， 所谓int ***类型就是指r只能存放int ** 类型变量的地址,
                //r = &p; //error 因为r是int *** 类型， r只能存放int **类型变量的地址
printf("i = %d\n", ***r); //输出结果是10 只有 ***r才表示的是i, *r或 **r或 ****r代表的都不是i
```

专题：

动态内存分配【重点难点】

传统数组的缺点：

1. 数组长度必须事先制定，且只能是常整数，不能是变量

例子：

```
int a[5]; //OK
int len = 5; int a[len]; //error
```

2. 传统形式定义的数组，该数组的内存程序员无法手动释放
在一个函数运行期间，系统为该函数中数组所分配的空间
会一直存在，直到该函数运行完毕时，数组的空间才会被
系统释放

3. 数组的长度一旦定义，其长度就不能在更改

数组的长度不能在函数运行的过程中动态的扩充或缩小

4. A函数定义的数组，在A函数运行期间可以被其它函数使用，
但A函数运行完毕之后，A函数中的数组将无法在被其他
函数使用

传统方式定义的数组不能跨函数使用

为什么需要动态分配内存

动态数组很好的解决了传统数组的这4个缺陷

传统数组也叫静态数组

动态内存分配举例_ 动态数组的构造

假设动态构造一个int型一维数组[至少保证要看懂]

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

1、 本语句分配了两块内存，一块内存是动态分配的，
总共len个字节，另一块是静态分配的，

并且这块静态内存是p变量本身所占的内存，总共4个字节

- 1、 malloc只有一个int型的形参，表示要求系统分配的字节数
- 2、 malloc函数的功能是请求系统len个字节的内存空间，如果请求分配成功，则返回第一个字节的地址，如果分配不成功，则返回NULL

malloc函数能且只能返回第一个字节的地址，所以我们需要把这个无任何实际意义的第一个字节的地址（俗称干地址）转化为一个有实际意义的地址，因此

malloc前面必须加（数据类型 *），表示把这个无实际意义的第一个字节的地址

转化为相应类型的地址。如：

```
int *p = (int *)malloc(50);
```

表示将系统分配好的50个字节的第一个字节的地址转化为int *型的地址，更准确的说是把第一个字节的地址转化为四个字节的地址，这样p就指向了第一个的四个字节，p+1就指向了第2个的四个字节，p+i就指向了第i+1个的4个字节。p[0]就是第一个元素，p[i]就是第i+1个元素

```
double *p = (double *)malloc(80);
```

表示将系统分配好的80个字节的第一个字节的地址转化为double *型的

地址，更准确的说是把第一个字节的地址转化为8个字节的地址，这样p就指向了第一个的8个字节，p+1就指向了第2个的8个字节，p+i就指向了第i+1个的8个字节。p[0]就是第一个元素，p[i]就是第i+1个元素

3. free(p)；

表示把p所指向的内存给释放掉 p本身的内存是静态的，
不能由程序员手动释放，p本身的内存只能在p变量所在的
函数运行终止时由系统自动释放

静态内存和动态内存的比较【重点】

静态内存是由系统自动分配，由系统自动释放

静态内存是在栈分配的

动态内存是由程序员手动分配，手动释放

动态内存是在堆分配的

跨函数使用内存的问题【重点】

静态内存不可以跨函数使用

所谓静态内存不可以跨函数使用更准确的说法是：

静态内存存在函数执行期间可以被其它函数使用，

静态内存存在函数执行完毕之后就不能再被其他函数使用了

动态内存可以跨函数使用

动态内存存在函数执行完毕之后仍然可以被其他函数使用

9> 结构体

为什么需要结构体

为了表示一些复杂的事物，而普通的基本类型无法满足实际要求

什么叫结构体

把一些基本类型数据组合在一起形成一个新的复合数据类型，这个叫做结构体

如何定义结构体

3种方式，推荐使用第一种：

//第一种 这只是定义了一个新的数据类型，并没有定义变量

```
struct Student
```

```
{  
    int age;  
    float score;  
    char sex;  
};
```

//第二种方式

```
struct Student2  
{  
    int age;  
    float score;  
    char sex;  
} st2;
```

//第三种方式

```
struct  
{  
    int age;  
    float score;  
    char sex;  
} st3;
```

怎样使用结构体变量

赋值和初始化

定义的同时可以整体赋初值

如果定义完之后，则只能单个的赋初值

如何取出结构体变量中的每一个成员【重点】

1. 结构体变量名. 成员名

2. 指针变量名->成员名 (第二种方式更常用)

指针变量名->成员名 在计算机内部会被转化成 (*指针变量名). 成员名
的方式来执行

所以说这两种方式是等价的

例子：

```
struct Student  
{
```

```

        int age;
        float score;
        char sex;
    };

    int main(void)
{
    struct Student st = {80, 66.6, 'F'}; //初始化 定义的同时赋初值
    struct Student * pst = &st; //&st不能改成st

    pst->age = 88;//第二种方式
    st.age = 10; //第一种方式

    return 0;
}

1. pst->age 在计算机内部会被转化成 (*pst). age, 没有什么为什么,
   这就是->的含义, 这也是一种硬性规定
2. 所以 pst->age 等价于 (*pst). age 也等价于 st. age
3. 我们之所以知道pst->age等价于 st. age, 是因为pst->age是被转化
   成了(*pst). age来执行
4. pst->age 的含义:
   pst所指向的那个结构体变量中的age这个成员

```

结构体变量和结构体指针变量作为函数参数传递的问题

推荐使用结构体指针变量作为函数参数来传递

结构体变量的运算

结构体变量不能相加, 不能想减, 也不能相互乘除

但结构体变量可以相互赋值

例子:

```

struct Student
{
    int age;
    char sex;
    char name[100];
}; //分号不能省
struct Student st1, st2;
st1+st2 st1*st2 st1/st2 都是错误的
st1 = st2 或者st2 = st1 都是正确的

```

举例

动态构造存放学生信息的结构体数组

动态构造一个数组, 存放学生的信息

然后按分数排序输出

10> 枚举

什么是枚举

把一个事物所有可能的取值一一列举出来

怎样使用枚举

枚举的优缺点

- 代码更安全
- 书写麻烦

专题：

补码：

原码

也叫 符号-绝对值码

最高位0表示正 1表示负，其余二进制位是该数字的绝对值的二进制位

原码简单易懂

加减运算复杂

存在加减乘除四种运算，增加了CPU的复杂度

零的表示不唯一

反码

反码运算不便，也没有在计算机中应用

移码

移码表示数值平移n位，n称为移码量

移码主要用于浮点数的阶码的存储

补码

已知十进制求二进制

求正整数的二进制

除2取余，直至商为零，余数倒叙排序

求负整数的二进制

先求与该负数相对应的正整数的二进制代码，然后将

所有位取反，末尾加1，不够位数时，左边补1

求零的二进制

全是零

已知二进制求十进制

如果首位是0，则表明是正整数，按普通方法来求

如果首位是1，则表明是负整数

将所有位取反，末尾加1，所得数字就是该负数的绝对值

如果全是零，则对应的十进制数字就是零

学习目标：

在VC++6.0中一个int类型的变量所能存储的数字的范围是多少

int类型变量所能存储的最大正数用十六进制表示是： 7FFFFFFF

int类型变量所能存储的绝对值最大的负整数用十六进制表示是： 80000000

具体可以参见“[8位二进制所代表的十进制 示意图.jpg](#)”

绝对值最小负数的二进制代码是多少

最大正数的二进制代码是多少

已知一个整数的二进制代码求出原始的数字

数字超过最大正数会怎样

不同类型数据的相互转化

进制转化 [ppt]

字符串的处理

链表

算法：

通俗定义：

解题的方法和步骤

狭义定义：

对存储数据的操作

对不同的存储结构，要完成某一个功能所执行的操作是不一样的

比如：

要输出数组中所有的元素的操作和

要输出链表中所有元素的操作肯定是不一样的

这说明：

算法是依附于存储结构的

不同的存储结构，所执行的算法是不一样的

广义定义：

广义的算法也叫泛型

无论数据是如何存储的，对该数据的操作都是一样的

我们至少可以通过两种结构来存储数据

数组

优点：

存取速度快

缺点：

需要一个连续的很大的内存

插入和删除元素的效率很低

链表

专业术语：

首节点

存放第一个有效数据的节点

尾节点

存放最后一个有效数据的节点

头结点

头结点的数据类型和首节点的类型是一模一样的

头结点是首节点前面的那个节点

头结点并不存放有效数据

设置头结点的目的是为了方便对链表的操作
头指针
存放头结点地址的指针变量

确定一个链表需要一个参数
头指针

优点：

插入删除元素效率高
不需要一个连续的很大的内存

缺点：

查找某个位置的元素效率低

杂记> 穿插在课堂中的零散知识笔记

算法：

解题的方法和步骤

如何看懂一个程序，分三步：

1. 流程
2. 每个语句的功能
3. 试数

如何学习一些需要算法的程序【如何掌握一个程序】

1. 尝试自己去编程解决它
但要意识到大部分人都是自己无法解决的，这时不要气馁，也不要自卑，
如果十五分钟还想不出来，此时我建议您就可以看答案了
2. 如果解决不了，就看答案
关键是把答案看懂，这个要花很大的精力，也是我们学习的重点，
看懂一个程序要分三步：流程、每个语句的功能、试数
3. 看懂之后尝试自己去修改程序，并且知道修改之后程序的输出结果的含义
不建议看懂程序之后就立即自己敲程序
4. 照着答案去敲
5. 调试错误
6. 不看答案，自己独立把答案敲出来
7. 如果程序实在无法彻底理解，就把它背会，不过无法彻底理解的程序非常少，
我自己只有在学数据结构时碰到过一个，学其他语言都没有碰到过

强制类型转化

格式：

(数据类型) (表达式)

功能：

把表达式的值强制转化为前面所执行的数据类型

例子：

(int) (4.5+2.2) 最终值是 6
(float) (5) 最终值是 5.000000

浮点数的存错所带来的问题

float和double都不能保证可以把所有的实数都准确的保存在计算机中

例子

```
float i = 99.9;  
printf("%f\n", i);
```

最终在VC++6.0中的输出结果是：99.900002

因为浮点数无法准确存储，所以就衍生出来两个编程问题：

有一个浮点型变量x，如何判断x的值是否是零

```
if (|x-0.000001| <= 0.000001)  
    是零  
else  
    不是零
```

为什么循环中更新的变量不能定义成浮点型

一些琐碎的运算符知识

1. 自增[或者自减]

分类：

前自增 — ++i
后自增 — i++

前自增和后自增的异同：

相同：

最终都使i的值加1

不同

前自增整体表达式的值是i加1之后的值

后自增整体表达式的值是i加1之前的值

为什么会出现自增

代码更精练

自增的速度更快

学习自增要明白的几个问题

1. 我们编程时应该尽量屏蔽掉前自增和后自增的差别

2. 自增表达式最好不要作为一个更大的表达式的一部分来使用

或者说

i++ 和 ++i 单独成一个语句，不要把它作为一个完整复

合语句的一部分来使用

如：

```
int m = i++ + ++i + i + i++; //这样写不但是不规范的代码,  
                                //而且是不可移植的代码
```

```
printf("%d %d %d", i++, ++i, i); //同上
```

2. 三目运算符

格式

A ? B : C

等价于

```
if (A)  
    B;  
else  
    C;
```

3. 逗号表达式

格式

(A, B, C, D)

功能：

从左到右执行

最终表达式的值是最后一项的值

位运算符：

& — 按位与

&& 逻辑与 也叫并且

&&与& 的含义完全不同

1&1 = 1

1&0 = 0

0&1 = 0

0&0 = 0

5&7=5 21&7=5

5&1=1 5&10=0

| — 按位或

||逻辑或

|按位或

1|0 = 1

1|1 = 1

0|1 = 1

0|0 = 0

~ — 按位取反

~i就是把i变量所有的二进制位取反

~ — 按位异或

相同为零

不同为1

1^0 = 1

0^1 = 1

1^1 = 0

0^0 = 0

<< — 按位左移

i<<3 表示把i的所有二进制位左移3位, 右边补零

左移n位相当于乘以2的n次方, 前提是数据不能丢失

面试题：

A) i = i*8;

B) i = i<<3;

请问上述两个语句, 哪个语句执行的速度快

答案：B快

>> — 按位右移

i>>3 表示把i的所有二进制位右移3位, 左边一般是0, 当然也可能补1

右移n位相当于除以2的n次方, 前提是数据不能丢失

面试题：

- A) $i = i/8;$
- B) $i = i>>3;$

请问上述两个语句，哪个语句执行的速度快

答案：B快

位运算的现实意义

通过位运算符我们可以对数据的操作精确到每一位

复习进制的知识：

- 1. 什么叫n进制
逢n进一
- 2. 把r进制转成十进制
- 3. 十进制转成r进制
- 4. 不同进制所代表的数值之间的关系

十进制的 3981 转化化成 十六进制是 F8D

十进制的3981和十六进制的F8D所代表的本质上都是同一个数字

二进制全部为零的含义 —0000000000000000 的含义

- 1. 数值零
- 2. 字符串结束标记符 '\0'
- 3. 空指针NULL

NULL本质也是零，而这个零不代表数字零，而表示的是内存单元的编号零

我们计算机规定了，以零为编号的存储单元的内容不可读，不可写

The end
2009年12月1日