

课堂讨论

● 数组

- + 元素访问效率高
- 空间扩充困难

● 链表

- + 节点扩充容易
- 节点访问效率低

● 结合二者的优点

```
const int M = 20;
```

```
struct Node
{
    int data[20];
    Node *next;
}; //60个整数需要3个节点
```

访问第 i 个元素（遍历链表的指针为 p ）：

第（ $i/20$ ）个节点：

第（ $p \rightarrow data[i\%20]$ ）个元素

补遗

1. 函数指针 难点
2. 结构数组与名表
 - 基于结构数组的信息检索程序
3. 联合
4. 栈
5. 文件

1. 函数指针：用指针操纵函数*

🌈 C程序运行期间，程序中每个函数的目标代码也占据一定的内存空间。C语言允许将该内存空间的首地址赋给函数指针类型变量（简称函数指针，注意与指针类型返回值的区别），然后通过函数指针来调用函数。

- 先构造一个函数指针类型
- 定义一个函数指针
或
- 在构造类型的同时定义函数指针
- 接着让函数指针 `pf` 指向内存的代码区
 - 用取地址操作符`&`获得函数的内存地址
 - 或
 - 直接用函数名获得函数的内存地址
- 然后通过函数指针调用函数`F`

```
typedef int (*PFUNC) (int);  
PFUNC pf;
```

或

```
int (*pf) (int);
```

```
pf = &F;
```

或

```
pf = F;
```

```
(*pf) (10);
```

或

```
pf(10); //实参为10
```

```
int F(int m)  
{  
    ...  
}
```

❁ 例：函数Integrate可以计算任意一个一元可积函数（由函数指针pfun操纵）在一个区间 $[x1, x2]$ 上的定积分。

➡ 该函数的调用形式：

Integrate(My_func, 1, 10);

//计算函数My_func在区间 $[1, 10]$ 上的定积分

Integrate(sin, 0, 1);

//计算函数sin在区间 $[0, 1]$ 上的定积分

Integrate(cos, 1, 2);

//计算函数cos在区间 $[1, 2]$ 上的定积分

➡ 该函数的原型：

```
double Integrate(double (*pfun)(double x), double x1, double x2)
```

➡ 即可以把一个函数名作为实参传给被调函数，被调函数的形参定义为一个函数指针₄

```
double My_func(double x)
{
    double f = x;
    return f;
}
```

`Integrate(cos, 1, 2);`

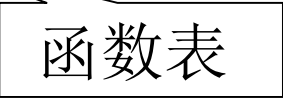
```
double Integrate(double (*pfun)(double x), double x1, double x2)
{
    double s = 0;
    int i = 1, n;          // i为步长, n为等份的个数, n越大, 计算结果精度越高
    printf("please input the precision: ");
    scanf("%d", &n);
    while(i <= n)
    {
        s += (*pfun)(x1 + (x2 - x1) / n * i);    //窄条的上底边≈下底边
        ++i;
    }
    s *= (x2 - x1) / n;    //纵坐标乘以高为每个窄条的面积
    return s;
}
```

横坐标作为实参, 返回值为对应的纵坐标 (即上底边或下底边)

● 例* 根据输入的要求，执行在函数表中定义的某个函数。

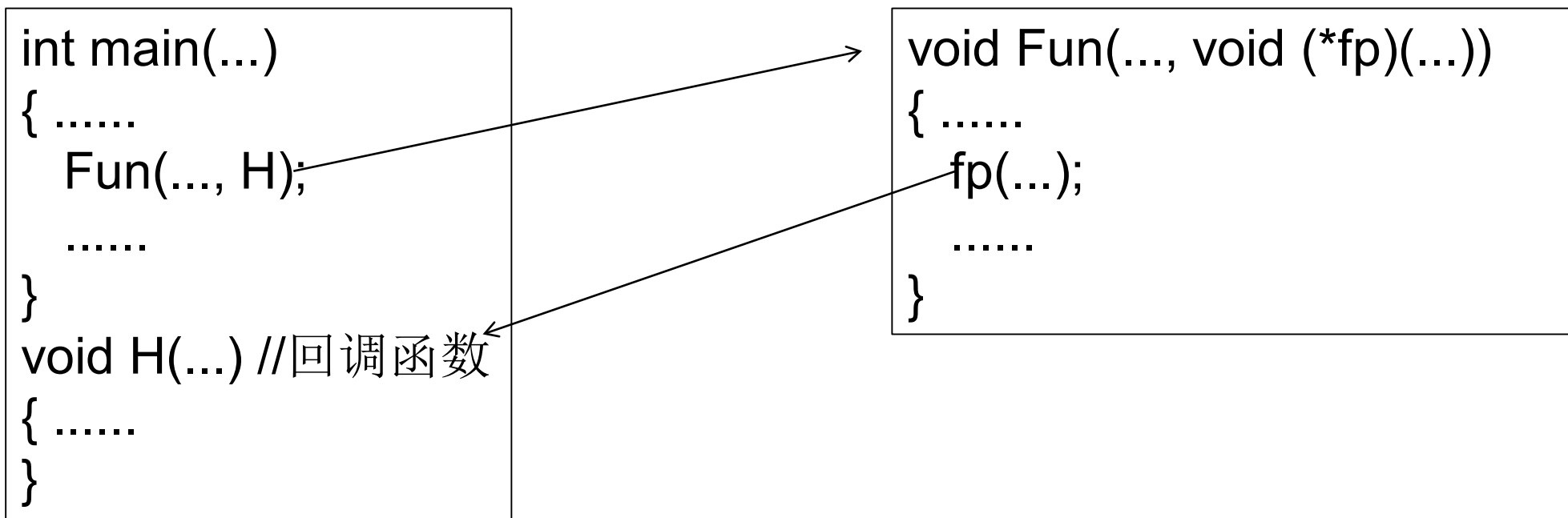
...

```
#include <cmath>
typedef double (*PF) (double);
PF func_list[8] = {sin, cos, tan, asin, acos, atan, log, log10};
int main( )
{
    int index;
    double x;
    do
    {
        printf("请输入要计算的函数(0:sin 1:cos 2:tan 3:asin 4:acos 5:atan 6:log 7:log10):\n");
        scanf("%d", &index);
    }while(index < 0 || index > 7);
    printf("请输入参数: ");
    scanf("%lf", &x);
    printf("结果为: %f \n", (*func_list[index])(x));
    return 0;
}
```



回调函数 (Callback Functions) **

- 一个函数 Fun 在执行的过程中，需要主调函数配合做些事，于是会调用主调函数提供的一个函数 H，H 被称为回调函数。回调函数通常是用函数指针传给被调用者。



例**：编写一个能根据不同要求进行排序的函数

```
struct Student
{
    int no;
    char name[20];
    .....
};

void Sort(Student st[], int num,
          bool (*less_than)(Student *st1, Student *st2))
{
    .....
    if (!less_than(&st[i], &st[j])) // 比较数组元素大小
    {
        ..... // 交换st[i]与st[j]
    }
    .....
}
```

```
bool less_than_by_no(Student *st1, Student *st2)
{ return (st1->no < st2->no);
}

bool less_than_by_name(Student *st1, Student *st2)
{ return (strcmp(st1->name, st2->name) < 0);
}

void F()
{
    .....
    Student st[100];
    .....
    Sort(st, 100, less_than_by_no); //按学号从小到大排序
    .....
    Sort(st, 100, less_than_by_name); //按姓名从小到大排序
    .....
} //若想由大到小排序，重新定义比较函数
```

例：快速排序

```
const int N = 5;
void Sort(int a[], int n, bool (*compr)(int x, int y))
{
    if(n==1 || !n) return;
    int key = a[0], t;
    int i=0, j=n-1;
    while(true)
    {
        while(!compr(a[j], key) && i < j) --j ;
        if(i==j) break;
        t = a[i], a[i] = a[j], a[j] = t;
        while(!compr(key, a[i]) && i < j) ++i;
        if(i==j) break;
        t = a[i], a[i] = a[j], a[j] = t;

    }
    Sort(a, i, compr);
    Sort(a + i + 1, n - i - 1, compr);
}
```

```
bool Less(int m, int n){ return m < n; } //回调函数
bool More(int m, int n){ return m > n; } //回调函数
Sort(a, N, Less); //升序
Sort(a, N, More); //降序
```

2. 结构类型数组与名表

- 结构数组可用于表示二维表格。比如，名表：

```
Stu stu_array[5]; //定义了一个一维结构数组
```

```
Stu stu_array[5] = { {1001, 'T', 'M', 20, 90.0}, ...,  
                    {1005, 'L', 'F', 18, 81.0} }; //初始化
```

```
#define N 5
enum FeMale {F, M};
struct Stu
{
    int id;
    char name;
    FeMale s;
    int age;
    float score;
};
```

stu_array[5]

结构变量

一维数组

	num	name	s	age	score
stu_array[0]	1001	T	M	20	90.0
stu_array[1]	1002	K	F	19	89.0
stu_array[2]	1003	M	M	19	95.5
stu_array[3]	1004	J	M	18	100.0
stu_array[4]	1005	L	F	18	81.0

例 基于结构数组的数据顺序查找。

```
float Search(Stu stu_array[], int count, int id)
```

```
{    int i;
    for(i = 0; i < count; ++i)
    {    if(id == stu_array[i].id)
        break;
    }
    if(i >= count)
        return -1.0;
    else
        return stu_array[i].score;
}
```

```
for(i = 0; i < count; ++i)
{    if(id == stu_array[i].id)
    {    return stu_array[i].score;
    }
    return -1.0;
```

```
#define N 5
enum FeMale {F, M};
struct Stu
{
    int id;
    char name;
    FeMale s;
    int age;
    float score;
};
```

● 例 基于结构数组的数据折半查找。

先按某一列排序

```
void  BublSort(Stu stu_array[ ], int count)
{
    int i, j;
    for(i = 0; i < count-1; ++i)
        for(j = 0; j < count-1-i; ++j)
            if(stu_array[j].id > stu_array[j+1].id)
            {
                Stu temp;
                temp = stu_array[j];
                stu_array[j] = stu_array[j+1];
                stu_array[j+1] = temp;
            }
}
```

```
float BiSearchR(Stu stu_array[], int first, int last, int id)
{
    if(first > last)
        return -1.0;
    int mid = (first + last) / 2;
    if(id == stu_array[mid].id)
        return stu_array[mid].score;
    else if(id > stu_array[mid].id)
        return BiSearchR(stu_array, mid + 1, last, id);
    else
        return BiSearchR(stu_array, first, mid - 1, id);
}
```

-
- ❁ 结构类型数组也可以用指针来操纵，操纵方法和用指针操纵其他基本类型数组的方法类似。比如，

```
Student stu[10], *psa;  
psa = stu;
```


3. 联合（union）类型

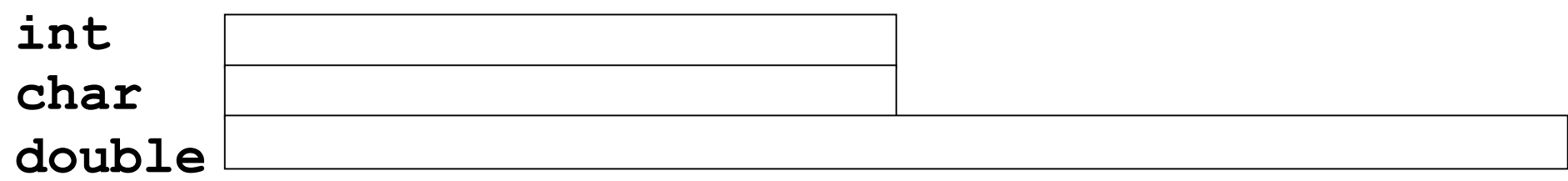
```
union myType
{
    int i;
    char c;
    double d;
};
```

与结构类型类似，
联合类型由程序员构造而成，
构造时需要用到关键字union。

联合变量的初始化、成员的操作方式
也与结构变量类似。

```
myType v;    // 定义了一个myType类型的联合变量v
```

与结构变量不同的是，系统采用覆盖技术按需要占用内存单元最多的成员为联合变量分配内存



```
struct B
{
    int i;
    char c;
    double d;
};
B b;
```

```
union A
{
    int i;
    char c;
    double d;
};
A v;
```

- 对于上述联合变量v，在程序中可以分时操作其中不同数据类型的成员。比如，

```
v.i = 12;           //以下只操作变量v的成员i
```

.....

```
v.c = 'X';          //以下只操作变量v的成员c
```

.....

```
v.d = 12.95;        //以下只操作变量v的成员d
```

.....

- 当给一个联合变量的某成员赋值后，再访问该变量的另外一个成员，将得不到原来的值。比如，

```
v.i = 12;
```

```
printf("%f", v.d);   //不会输出12
```

- 即可以分时把v当作不同类型的变量来使用，但不可以同时把v当作不同类型的变量来使用。

- 联合类型使程序呈现出某种程度的多态性。这种多态性的好处是：在提高程序灵活性的同时，可以实现多种数据共享内存空间。比如，

```
union Array
{
    int int_a[100];
    double dbl_a[100];
};
```

```
Array buffer;
```

```
... buffer.int_a ...    //使用数组int_a，只有一半存储空间闲置
```

```
.....
```

```
... buffer.dbl_a ...    //使用数组dbl_a，没有存储空间闲置
```

如果不使用联合类型，例如，

```
int int_a[100];
```

```
double dbl_a[100];
```

```
... int_a ...      //使用数组int_a (dbl_a所占的存储空间闲置)
```

```
.....
```

```
... dbl_a ...      //使用数组dbl_a (int_a所占的存储空间闲置)
```

```
.....
```

或 含联合类型的结构数组

🌈 如果是联合类型的数组，则其每一个元素的类型可以不同。比如，

```
enum Grade{UNDERGRAD, MASTER, PHD, FACULTY};    // 职级
```

```
union Performanc
```

```
{  int nPaper;    // 已发表论文篇数
```

```
    float gpa; //GPA
```

```
}; // 业绩类型
```

```
struct Person
```

```
{  char id[20];
```

```
    char name[20];
```

```
    enum Grade grd;
```

```
    union Performanc pfmc; // 每个人业绩属性的类型可以不同
```

```
};
```

```
const int N = 800;           //人员的个数
void input(Person prsn[ ], int num);
int main( )
{   Person prsn[N] = {{0,0, UNDERGRAD,0}};
    input(prsn, N);
    float maxgpa = 0.1;
    int maxMaster = 0;
    int maxPhd = 0;
    int maxFaculty = 0;
```

```
for(int i = 0; i < N; ++i)
{
    if(prsn[i].grd == UNDERGRAD
        && prsn[i].pfmc.gpa > maxgpa)
        maxgpa = prsn[i].pfmc.gpa;
    if( prsn[i].grd == MASTER
        && prsn[i].pfmc.nPaper > maxMaster)
        maxMaster = prsn[i].pfmc.nPaper;
    if( prsn[i].grd == PHD
        && prsn[i].pfmc.nPaper > maxPhd)
        maxPhd = prsn[i].pfmc.nPaper;
    if( prsn[i].grd == FACULTY
        && prsn[i].pfmc.nPaper > maxFaculty)
        maxFaculty = prsn[i].pfmc.nPaper;
}
```

```
for(int i = 0; i < N; ++i)
    if(prsn[i].grd == UNDERGRAD
        && prsn[i].pfmc.gpa == maxgpa)
        printf( "本科生获奖者: %s \n", prsn[i].name);
//...
return 0;
}
```

```

void input(Person prsn[ ], int num)
{
    int g = 0;
    for(int i = 0; i < num; ++i)
    {
        printf( "输入人员的编号与姓名: \n");
        scanf("%s%s", prsn[i].id, prsn[i].name);
        printf( "输入0~3分别代表本科、硕士、博士生和教师: ");
        scanf("%d", &g);
        switch(g)
        {
            case 0: prsn[i].grd = UNDERGRAD;
                    printf( "输入学分绩: ");
                    scanf("%f", &prsn[i].pfmc.gpa);
                    break;
            case 1: prsn[i].grd = MASTER;
                    printf( "输入已发表论文篇数: ");
                    scanf("%d", &prsn[i].pfmc.nPaper);
                    break;
            .....
            .....
        }
    }
}

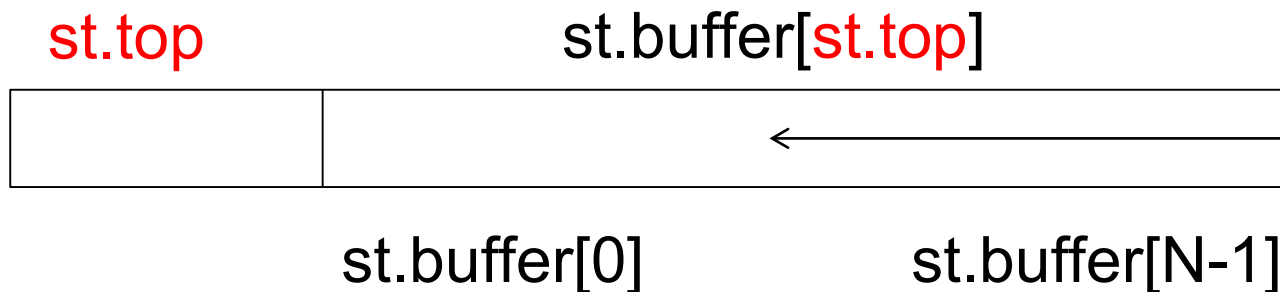
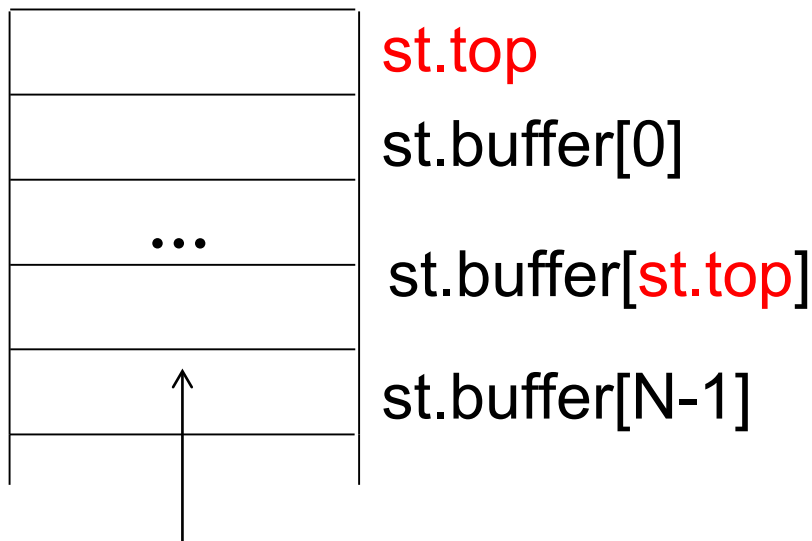
```

4. 栈

一种带有操作约束的结构类型

- 对栈只能进行两种操作
 - 进栈（增加一个元素）
 - 出栈（删除一个元素）
- 这两个操作必须在栈的同一端（称为栈顶，top）进行
 - 即后进先出（Last In First Out，简称LIFO）
 - 若 `push(x)`；`pop(y)`； 则 `x==y`
- eg. 函数调用过程中用栈保存函数的局部变量、参数、返回值以及返回地址

```
struct Stack
{
    int top;
    int buffer[N];
}st;
```



栈的实现

```
const int N = 100;
struct Stack
{
    int top;
    int buffer[N];
};
#include <stdio.h>
int main()
{
    Stack st;           //定义栈数据
    st.top = -1;         //初始化栈
    st.top++;
    st.buffer[st.top] = 12; //存数入栈
    int x = st.buffer[st.top]; //取数出栈
    st.top--;
    return 0;
}
```

操作者（如main函数的编写者）必须知道栈的数据表示，数据表示发生变化将影响操作；

且可以修改数据（如成员buffer的数据类型），数据没有得到保护（如误把`st.top++`写成`st.top--`）；

会忘记初始化；

栈的实现与使用混在一起；

需求的改变会导致整个程序结构的变动，难以维护。

过程抽象

```
const int N = 100;

struct Stack
{
    int top;
    int buffer[N];
};
```

```
void Init(Stack *s)
{
    s -> top = -1;
} //初始化栈
```

```
bool Push(Stack *s, int i)
{
    if(s -> top == N-1)
    {
        printf("Stack is overflow\n");
        return false;
    }
    else
    {
        s -> top++;
        s -> buffer[s -> top] = i;
        return true;
    }
} //入栈
```

```
bool Pop(Stack *s, int *i)
{
    if(s -> top == -1)
    {
        printf("Stack is empty.\n");
        return false;
    }
    else
    {
        *i = s -> buffer[s -> top];
        s -> top--;
        return true;
    }
} //出栈
```

C++

过程抽象

```
const int N = 100;

struct Stack
{
    int top;
    int buffer[N];
};
```

```
void Init(Stack& s)
{
    s.top = -1;
    //初始化栈
}
```

```
bool Push(Stack& s, int i)
{
    if(s.top == N-1)
    {
        printf("Stack is overflow\n");
        return false;
    }
    else
    {
        s.top++;
        s.buffer[s.top] = i;
        return true;
    }
} //入栈
```

```
bool Pop(Stack& s, int& i)
{
    if(s.top == -1)
    {
        printf("Stack is empty.\n");
        return false;
    }
    else
    {
        i = s.buffer[s.top];
        s.top--;
        return true;
    }
} //出栈
```

- ❁ 初始化、入栈及出栈过程，均抽象为函数来实现。一个函数对应一个功能，可以清晰地描述计算任务；还能实现一定程度的软件复用。
- ❁ 上述函数的声明与结构类型的构造可以放在头文件中，以便于调用，调用者不必知道数据的表示，main函数可以得到简化。

```
#include "....."
int main()
{
    Stack st;           //定义栈数据
    Init(st);           //初始化栈
    Push(st, 12);       //存数
    int x;
    Pop(st, x);         //取数
    printf("x: %d\n", x);
    return 0;
}
```

过程抽象的优劣

```
int main()
{
    Stack st;           //定义栈数据
    st.top = -1;         //初始化栈
    st.top++;
    st.buffer[st.top] = 12;
    int x = st.buffer[st.top];
    st.top--;
    printf("x: %d\n", x);
    return 0;
}
```

```
#include "....."
```

```
int main()
```

```
{
    { Stack st; //定义栈数据
      Init(st); //初始化栈
```

```
    Fun(st);
```

```
    { Push(st, 12);
```

```
      int x;
```

```
      Pop(st, x);
```

```
      printf("x: %d\n", x);
```

```
      return 0;
```

```
}
```

```
    st.top--;
```

```
    st.buffer[st.top] = 12;
```

```
void f(Stack& s)
{.....
  //把栈修改成普通数组
}
```

存在的问题

- 数据类型的构造与对数据操作的定义是分开的，二者之间没有显式的联系，Init、Push、Pop函数在形式上跟其他函数没有区别；
- 数据表示仍然是公开的，对数据缺乏足够的保护，其他操作或函数也能操作数据；
- 仍会忘记调用Init(st)对栈进行初始化；
- 需求的改变仍会导致整个程序结构的变动，难以维护。

C++

数据抽象

```
const int N = 100;
class Stack
{
public:
    Stack();
    bool Push(int i);
    bool Pop(int& i);
private:
    int top;
    int buffer[N];
};
```

```
Stack::Stack() { top = -1; }
bool Stack::Push(int i)
{
    if(top == N-1)
    {
        cout << "Stack is overflow\n";
        return false;
    }
    else
    {
        top++;
        buffer[top] = i;
        return true;
    }
}
bool Stack::Pop(int& i)
{
    if(top == -1)
    {
        cout << "Stack is empty\n";
        return false;
    }
    else
    {
        i = buffer[top];
        top--;
        return true;
    }
}
```

```

int main()
{
    Stack st;           } //会自动调用st.Stack()初始化, 不允许 st.top = -1;
    st.Push(12);        } //不允许或 st.top++; 或 st.buffer[st.top] = 12;
    int x;              }
    st.Pop(x);          } //不允许st.Fun();
    ...
    return 0;
}

```

- 数据类型的构造和对数据操作的定义构成了一个整体, 对数据操作的定义是数据类型构造的一部分;
- 只能通过提供的成员函数来操作栈;
- 自动进行初始化;
- 如果改变栈的实现 (比如用链表实现) 对使用者没有影响。

```

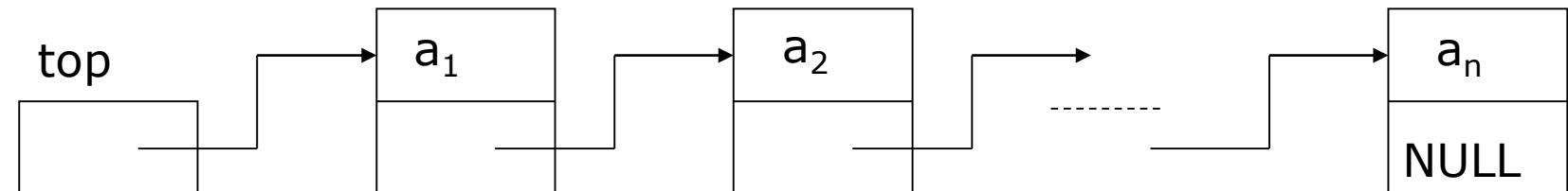
#include "....."
int main()
{
    { Stack st;           //定义栈数据
      Init(st);           //初始化栈
    { Push(st, 12);       //存数
      int x;
    { Pop(st, x);         //取数
      printf("x: %d\n", x);
      return 0;
    }
}

```

用链表实现栈类Stack

```
class Stack
{
public:
    Stack();
    bool push(int i);
    bool pop(int& i);

private:
    struct Node
    {
        int content;
        Node *next;
    } *top;
};
```



```
Stack::Stack() { top = NULL; }
```

```
bool Stack::Push(int i)
{
    Node *p = new Node;
    if(p == NULL)
    {
        cout << "Stack is overflow\n";
        return false;
    }
    else
    {
        p -> content = i;
        p -> next = top;
        top = p;
        return true;
    }
}
```

```
bool Stack::Pop(int& i)
{
    if(top == NULL)
    {
        cout << "Stack is empty\n";
        return false;
    }
    else
    {
        Node *p = top;
        top = top -> next;
        i = p -> content;
        delete p;
        return true;
    }
}
```

```
int main()
{
    Stack st;
    st.Push(12);
    int x;
    st.Pop(x);
    ...
    return 0;
}
```

//修改栈的实现之后，使用者的代码不受影响

5. 文件

- 概述
- 文件类型指针
- 文件的打开
- 文件的读写
- 文件的定位
- 文件的关闭
- 注意事项

概述

- 程序运行结果有时需要永久性地保存起来，以供其他程序或本程序下一次运行时使用。程序运行所需要的数据也常常要从其他程序或本程序上一次运行所保存的数据中获得。
- 用于永久性保存数据的设备称为外部存储器（简称：外存），如：磁盘、磁带、光盘等。
- 在外存中保存数据的方式通常有两种：文件和数据库。本课程只介绍文件方式。
- 计算机中的文件是一种数据集合，每个文件由若干个数据项序列组成，操作系统将其组织在特定的目录（文件夹）中。
- 每个文件由“文件名.扩展名”来标识，扩展名通常有1~3个字母，例如
 - “文件名.c”表示C程序的源文件，
 - “文件名.exe”表示可执行文件，
 - “文件名.jpg”表示一种图像的压缩数据文件，
 - “文件名.cpp”表示C++程序的源文件，
 - “文件名.txt”表示文本文件，
 - “文件名.dat”可以表示自定义数据文件
 -

- 在C/C++中，文件被看作外存设备中的一段字节串（一般用十六进制数表示），这是一种流式文件处理方式。

C家族语言基本延续了这种无结构的流式文件处理方式，与之相对的是有结构的记录式文件处理方式。

- 根据文件中数据存储时的编码可以将C文件分为：

(1) 二进制文件 (binary)

- 按数据对应的二进制数所组成的字节串存储。例如，对于32位机，整数365可存为00 00 01 6d四个字节串，整数2147483647存为7f ff ff ff四个字节串，字符'A'可存为41一个字节串，字符串"365"可存为33 36 35三个字节串。
- 可以包含任意的二进制字节。
- 一般用于存储无显式结构的数据。

(2) 文本文件 (text)

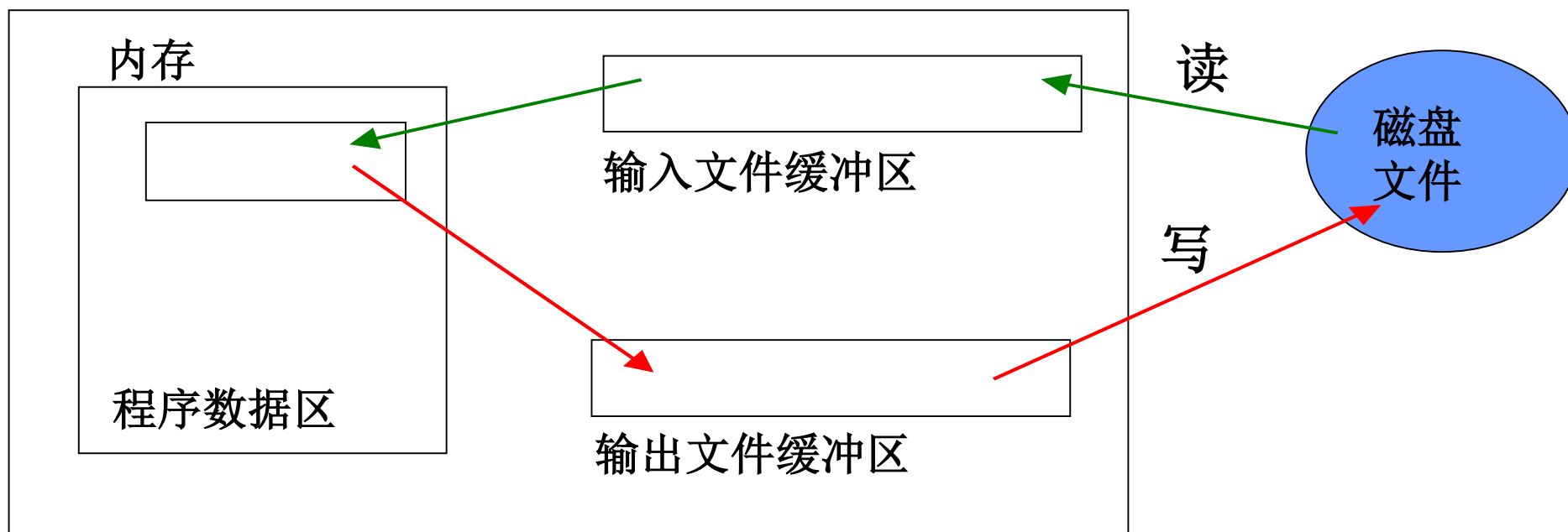
- 按数据中每个字符的ASCII码组成的字节串存储。例如，整数365存为33 36 35三个字节串，整数2147483647存为32 31 34 37 34 38 33 36 34 37十个字节串，字符'A'存为41一个字节串，字符串"365"存为33 36 35三个字节串。
- 只包含可显示字符和有限的几个控制字符（如：'\r'、'\n'、'\t'等）。
- 一般用于存储具有“行”结构的文本数据。

-
- 可见，文本文件的平台无关性更好，但文本文件中的数据只能按文本含义来理解，而二进制文件中的数据可以由读/写程序自行约定为各种含义。另外，文本（字符与字符串）在二进制文件和文本文件中一般没有什么不同，但一些特殊字符，例如，表示回车换行的转义字符等，因不同操作系统的处理方式不同会有差别，编程时需注意这个问题（参见后面的**注意事项**）。

● 对文件的操作（即对文件的访问）通常是按**字节**为单位**顺序**进行的，包括：

- 读操作：一般指从外存设备将数据逐个字节读至内存（对于内存而言，是输入数据）
- 写操作：一般指将内存的数据逐个字节写至外部设备（对于内存而言，是输出数据）

- 对外部设备的访问，速度比内存访问速度低得多，为了减少访问时间，提高程序执行效率，C语言采用缓冲机制，一次读/写一批数据，存于缓冲区，以减少读/写次数。缓冲区的大小由具体的执行环境确定。



文件类型指针

- 为了对文件进行有效管理，头文件 `stdio.h` 中一般定义了一个名为 `FILE` 的结构类型，例如：

```
typedef struct
{
    short level;           //缓冲区满空程度
    unsigned flags;        //文件状态标志
    char fd;               //文件描述符
    unsigned char hold;    //无缓冲则不读取字符
    short bsize;           //缓冲区大小
    unsigned char *buffer; //数据缓冲区
    unsigned char *curp;   //当前位置指针
                           //每读/写一个字节，自动自增1
    short token;           //用于有效性检查
} FILE ;
```

-
- 对于每一个要操作的文件，都必须定义一个FILE类型的指针变量，并使它指向“文件信息描述区”，以便对文件进行读/写操作。
 - “文件信息描述区”由执行环境在程序打开文件时自动创建。
 - 文件的**打开、读/写操作、关闭**等环节需要调用相应的库函数。

文件的打开

在对文件进行读写操作前，要先打开文件，以便为文件建立“文件信息描述区”，即用程序内部一个表示文件的变量/对象与外部一个具体文件之间建立联系，并指定按文本文件还是按二进制文件来打开。

文件的打开是通过库函数`fopen`实现的，其原型为：

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

- 参数`filename`是要打开的文件名（包括路径）；
- 参数`mode`是文件的处理模式，它可以是：`r`、`w`、`a`...

打开模式	描 述
r	只读，打开已有文件，不能写
w	只写，创建或打开，覆盖已有文件
a	追加，创建或打开，在已有文件末尾追加
r+	读写，打开已有文件
w+	读写，创建或打开，覆盖已有文件
a+	读写，创建或打开，在已有文件末尾追加
t	按文本方式打开 (缺省)
b	按二进制方式打开

- 如果成功打开文件，则函数fopen的返回值为被打开文件信息描述区的地址，否则返回空指针。比如，

//字符串中的反斜杠需用转义符

```
FILE *pfile = fopen("d:\\data\\tfile.txt", "w");
```

```
if(pfile == NULL)
```

在Unix和Linux环境下: "d:/data/tfile.txt"

```
    printf("Error! \n");
```

```
else
```

```
    printf("file1.txt has been opened. \n");
```

执行该代码前，用户需先在计算机的d盘建立data目录。该程序执行后，用户可以搜索到相应目录下新创建的tfile.txt文件。

不能成功打开文件的原因有多种，包括当前用户没有磁盘的访问权限、目录不存在等，如果是读模式或读更新模式，文件不存在也会导致文件打开失败。

文件的读/写操作

- 文件的读/写操作也是通过库函数实现的，常用文件操作库函数有

函数	功能
fputc	输出字符
fgetc	输入字符
fputs	输出字符串
fgets	输入字符串

函数	功能
fprintf	格式化输出
fscanf	格式化输入
fwrite	输出数据块
fread	输入数据块

fputc

🌈 **`int fputc(int c, FILE *stream);`**

- 该函数的功能是将字符c写至文件，正常情况下返回字符c的ASCII码，否则（发生写入错误等异常时）返回EOF。

fprintf

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

- 该函数的功能是将**基本类型**数据**写至文件**，正常情况下返回传输字符的个数，否则返回一个负整数。参数format与printf函数的参数类似。例如，

```
pfile = fopen("d:\\data\\tfile.txt", "w");
char name[20];          //学生姓名
int num;                //学号
scanf("%d", &num);
while(num > 0)          //可以输入学号0结束程序
{
    scanf("%s", name);
    fprintf(pfile, "%d %s\n", num, name);
    scanf("%d", &num);
}
```

fwrite

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

- 该函数的功能是按字节将ptr指向的nmemb个字节块的数据写至文件。size_t是在头文件stddef.h中定义的类型，相当于unsigned int。参数size为字节块的大小。正常情况下返回实际写至文件的字节块的个数，异常情况下返回值小于nmemb。如果size或nmemb的值为0，则返回0，文件内容不变。
- 例如，“`fwrite(&i, sizeof(i), 1, pfile);`”表示向某二进制文件写入i的值。
- 又如，“`fwrite("\r\n", 1, 2, pfile);`”表示向某二进制文件写入两个字符。

fgetc

🌈 **int fgetc(FILE *stream);**

- ➡ 该函数功能是从文件读取一个字符，正常情况下返回字符的ASCII码，否则返回EOF。

fscanf

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

- 该函数的功能是从文件读取基本类型数据（文件中的整数、小数之间应有分隔符），正常情况下返回读取数据的个数，否则返回EOF。参数format与scanf函数的参数类似。例如，

```
pfile = fopen("d:\\data\\tfile.txt", "r");
```

```
int i = 0;
```

```
if(fscanf(pfile, "%d", &i) != EOF)
```

```
    //从文件中读取一个整数，赋给变量i
```

```
    printf("%d", i);
```

fread

size_t **fread**(const void *ptr, size_t **size**, size_t **nmemb**, FILE *stream);

- 该函数的功能是从文件将nmemb个字节块的数据按字节读至ptr所指向的字节块，返回实际读取字节块的个数。例如，“fread(&i, sizeof(i), 1, pfile);”表示从某二进制文件中读取数据，赋给变量i。

-
- 从文件中**读取**数据时，要根据文件中数据的存储格式选用恰当的库函数，否则无法正确读取数据。一般地，用fgetc、fgets、fscanf和fread函数一一对应fputc、fputs、fprintf和fwrite函数产生的文件数据进行读操作，并且要保持其中参数类型一致。

文件的定位

- 一般情况下，文件的读/写操作是顺序进行的，即，在进行读操作时，如果要读文件中的第 n 个字节，则必须先读前 $n-1$ 个字节；在进行写操作时，如果要写第 n 个字节，则也必须先写前 $n-1$ 个字节。这种文件的访问方式效率往往不高。
- 每个打开的文件都有一个位置指针，指向当前读/写位置，每读/写一个字符，文件的位置指针都会自动往后移动一个位置。文件位置指针还可以用库函数来显式指定

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

- 该函数的功能是将文件位置指针指向位置 `whence + offset`，参数 `whence` 指出参考位置，其取值可以为 `SEEK_SET`（系统定义的值为0的宏，表示文件头），`SEEK_CUR`（系统定义的值为1的宏，表示当前位置）或 `SEEK_END`（系统定义的值为2的宏，表示文件末尾），参数 `offset` 偏移参考位置的字节数，它可以为正值（向后偏移）或负值（向前偏移）。例如，

```
fseek(pfile, 10, SEEK_SET); //位置指针移至第11个字节处
```

```
fseek(pfile, 10, SEEK_CUR); //从当前位置向后移动10个字节
```

```
fseek(pfile, -10, SEEK_END); //移至倒数第10个字节处
```

-
- ❁ 文件位置指针的当前位置可以通过库函数获得:

```
long ftell(FILE *stream);    //返回位置指针的位置
```

- ❁ 也可以用库函数将文件位置指针拉回到文件头部:

```
void rewind(FILE *stream);  
//等价于 fseek(stream, 0, SEEK_SET)
```

文件的关闭

完成文件的相关操作之后，应及时关闭文件，以释放缓冲区的空间。关闭文件时，执行环境先将输出文件缓冲区的内容都写入文件（无论缓冲区是否为满），然后关闭文件，这样可防止丢失准备写入文件的数据。

文件的关闭是通过库函数`fclose`实现的，`fclose`函数的原型为：

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

◆ 参数`pfile`是要关闭的文件信息描述区指针。

◆ 如果成功关闭文件，则函数`fclose`的返回值为0，否则返回EOF。

-
- 概述
 - 文件类型指针
 - 文件的打开
 - 文件的读写
 - 文件的定位
 - 文件的关闭
 - 注意事项

EOF

- EOF代表与任何字符的ASCII码都不相同的一个值，是程序中表示文本文件操作异常的宏名，能增强程序的可移植性。开发环境通常在头文件 `stdio.h` 中进行类似如下的定义：

```
#define EOF -1
```

- 最典型的文本文件操作异常是从文本文件末尾（`end of file`）读数据，即读到文本文件结束标志时发生异常。

文件结束标志

- 文件结束标志跟操作系统有关，例如，在DOS和Windows环境下，键入Ctrl+Z（ASCII码为0x1A）作为文件的结束标志，在UNIX和Linux环境下，键入Ctrl+D（ASCII码为0x04）作为文件的结束标志
- 例如，库函数getchar将键盘看作输入设备文本文件，当用户从键盘输入文件结束标志时，库函数读到后会返回EOF：

```
char ch;  
while((ch = getchar()) != EOF)  
    putchar(ch);
```

若将"char"改成"unsigned char"，则当getchar函数返回EOF时，会被隐式转换成无符号数255，从而与此段代码中的EOF（-1）不等，以致即使输入文件结束符循环也无法正常结束。对于char默认为unsigned char的开发环境，应将变量ch定义为"int"，以便能涵盖正常值（ASCII码）与异常值（EOF对应的值）。

\n

- 类似地，行结束标志也跟操作系统有关。例如，向文本文件写一个字符\n，作为某行的结束：

```
pfile = fopen("d:\\data\\tfile.txt", "w");  
fputc('\\n', pfile);
```

- 上述代码在不同环境下执行后效果不同，在Windows环境下查看文件tfile.txt的属性，大小为2字节，而不是1字节。在DOS和Windows环境下，ASCII码为0x0A的回车换行符 \n 写入文本文件时，会自动在前面添加一个ASCII码为0x0D的回车符\r。而在UNIX和Linux环境下，则不会有此现象。

- 所以，也应注意代码的通用性，通常改为：

```
pfile = fopen("d:\\data\\tfile.txt", "wb");  
fputc('\\n', pfile);
```

按二进制方式打开

- 在Windows环境下执行后，文件tfile.txt的大小也为1字节

- 此外，从文本文件读取数据时，对于ASCII码为0x0D的回车符\r + ASCII码为0x0A的回车换行符\n，在DOS和Windows环境下，ASCII码为0x0D的回车符\r会丢弃（在UNIX和Linux环境下则不会丢弃）：

```
char ch;
```

假定文件中含一个\r和一个\n

```
pfile = fopen("d:\\data\\tfile.txt", "r");
```

```
while(fscanf(pfile, "%c", &ch) != EOF)
```

```
    printf("%d\n", ch);
```

或while(fread(&ch, 1, 1, pfile) != 0

在上述代码在Windows下只显示10（回车换行符\n的ASCII码），添加模式字母b，改为按二进制方式打开文件，通常可提高代码的通用性。

● 例

```
const int N = 10;
struct
{
    int no;
    float score;
} stu[N];
```

文件(F) 编辑(E) 格式(O)

```
1 80
1 80
1 80
1 80
1 80
1 80
1 80
1 80
1 80
1 80
1 80
```

```
FILE *fp;
char file[] = "c:\\user\\stuFile.txt";
if(! (fp=fopen(file, "r+")))
{
    printf("Open file error!\n");
    exit(0);
}
float s=0;
for(int i = 0; i < N; ++i)
{
    fseek(fp, 3, SEEK_CUR);
    fscanf(fp, "%f", &stu[i].score);
    s += stu[i].score;
}
printf("%f \n", s);
fclose(fp);
```

feof

❁ **int feof(FILE *stream);**

- 该函数的功能是判断文件是否结束，**如果文件结束（即文件位置指针在文件末尾）并继续进行读操作**，则返回非0数，否则返回0。例如，

```
pfile = fopen("d:\\data\\tfile.txt", "r");
int i = 0;
while(!feof(pfile))
{
    if(fscanf(pfile, "%d", &i) != EOF)
        printf("%d\\n", i);
}
```

- 文件结束和其他文件操作异常都有可能相关函数返回EOF，feof函数可以专门用来判断文件是否结束（其他错误可以用函数ferror()进行甄别）。

注意文件尾部的空白符带来的问题：

```
pfile = fopen("d:\\data\\tfile.txt", "r");
int i = 0;
while(!feof(pfile))
{
    fscanf(pfile, "%d", &i); //读之后文件指针移动
    printf("%d\\n", i);
}
```

第二次 **fscanf** 读一个整数后一切正常，**feof** 返回 非0

第三次 **fscanf** 读不到一个整数后，**i** 维持不变，**feof** 返回 0

1\\n
5\\n

输出
1
5
5

修改:

```
pfile = fopen("d:\\data\\tfile.txt", "r");
int i = 0;
fscanf(pfile, "%d", &i); //读之后文件指针移动
while(!feof(pfile))
{
    printf("%d\\n", i); //注意顺序
    fscanf(pfile, "%d", &i); //读之后文件指针移动
}
```

1\\n
5\\n

输出
1
5

或

```
pfile = fopen("d:\\data\\tfile.txt", "r");
int i = 0;
while(!feof(pfile))
{
    if(fscanf(pfile, "%d", &i) != EOF) //读之后文件指针移动
        printf("%d\\n", i);
}
```

文件尾部无空白符:

```
pfile = fopen("d:\\data\\tfile.txt", "r");
int i = 0;
while(!feof(pfile))
{
    fscanf(pfile, "%d", &i); //读之后文件指针移动
    printf("%d\\n", i);
}
```

第二次 `fscanf` 读一个整数时读不到数据分隔符, `feof` 返回 0

1\\n
5

输出
1
5

或

```
pfile = fopen("d:\\data\\tfile.txt", "r");
int i = 0;
while(!feof(pfile))
{
    if(fscanf(pfile, "%d", &i) != EOF) //读之后文件指针移动
        printf("%d\\n", i);
}
```

❁ 例：带形参的 `main` 函数

- ❁ 一般情况下，程序不需要调用者（比如操作系统）提供参数，定义`main`函数时不用定义形参，如果程序需要用到调用者提供的参数，则可以在定义`main`函数时给出形参的定义。
- ❁ 一个文件拷贝程序`copy.c`，可以按“`copy file1 file2`”的命令形式来执行文件`file2`至文件`file1`的拷贝。

```
int main(int argc, char *argv[ ])
{
    FILE *fp1, *fp2;
    if( !(fp2 = fopen(argv[2], "r+"))
        || !(fp1 = fopen(argv[1], "w+")) )
        // "c:\\user\\file2.txt"
    {
        printf("Open file error!\n");
        exit(0);
    }

    int s = 0;
    fscanf(fp2, "%d", &s);
    fprintf(fp1, "%d", s);

    fclose(fp1);
    fclose(fp2);
    return 0;
}
```

- ❁ 例 假定无线传感器网络采集的火山口温度值存于数据文件vol.dat中，编程对异常采集数据进行预处理（比如将温度值为0的采集数据用前后相邻两个采集数据的平均值代替，最后一个温度值若为0用前一个数代替）。
- ❁ [分析] 可以采用随机数生成的方式模拟数据采集，以便向所创建的文件中写入原始数据。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 100
void DatGenerator(FILE **fpp);
void PreTreat();
```

```
int main( )
{
    FILE *fp;
    char *fname = "d:\\data\\vol.dat";
    if( (fp = fopen(fname, "wb")) == NULL )
        //打开、建立数据文件
    {
        printf("Can't open this file ! \n");
        exit(0);
        //exit(0) 为结束程序运行的库函数
        //其相关信息在头文件stdlib.h中
    }
    DatGenerator(&fp); //调用函数，产生原始数据
    fclose(fp);        //关闭文件，保存数据
```

```
printf("原始数据为: \n");
FILE *pfile = fopen("d:\\data\\vol.dat", "rb");
float t;
while(!feof(pfile))
{
    if(fread(&t, sizeof(t), 1, pfile) != 0)
        printf("%f\n", t);
}                                //观察产生的原始数据
fclose(pfile);

PreTreat();                    //调用函数，预处理数据

return 0;

}
```

```
void DatGenerator(FILE **fpp)
    //注意实参是指针变量的地址
{
    srand(time(0));
    for(int i = 0; i < N; ++i)
    {
        float j = 100.0*rand()/RAND_MAX;
        //生成随机数，模拟数据采集
        fwrite(&j, sizeof(j), 1, *fpp);    //写入文件

        float k = 0;
        fwrite(&k, sizeof(k), 1, *fpp);
        //为便于调试，故意间隔写入异常数据0
    }
}
```

```
void PreTreat()
{
    FILE *pfile = fopen("d:\\data\\vol.dat", "r+b");
    float t, tNext, tPre;
    fread(&tPre, sizeof(t), 1, pfile);
        //读取第一个数据作为前一个数据
    fread(&t, sizeof(t), 1, pfile);
        //读取第二个数据作为当前待处理数据
    while(!feof(pfile))
    {
        if(fread(&tNext, sizeof(t), 1, pfile) != 0)
            //读取下一个数据
            {
                if(t <= 0.1)
                {
                    t = (tPre + tNext)/2;
                    fseek(pfile, -2*sizeof(t), SEEK_CUR);
                    //位置指针往回移至待处理数据前端
```

```
        fwrite(&t, sizeof(t), 1, pfile); //改数
        fseek(pfile, sizeof(t), SEEK_CUR);
        //恢复位置指针至下一个数据前端
    }
    tPre = t; //当前数据处理完作为下一次处理的前一数
    t = tNext; //下一个数据作为待处理数据
}
//调试阶段可将此循环改为for(int i=0; i < 9; ++i)
fseek(pfile, -sizeof(t), SEEK_CUR);
//恢复位置指针至最后一个数据前端
fwrite(&tPre, sizeof(t), 1, pfile); //修改最后一个数据
printf("\n预处理后的数据为: \n");
rewind(pfile);
```

```
while(!feof(pfile))
{
    if(fread(&t, sizeof(t), 1, pfile) != 0)
        printf("%f\n", t);
}    //观察预处理结果数据

fclose(pfile);
}
```

小结

- ❁ 函数指针不是指向内存的数据区，而是代码区。
- ❁ 结构数组可存储名表，在实际生产生活中发挥作用。
- ❁ 联合是一种与结构类似的派生数据类型，与结构类型的不同点，仅在于存储方式（系统对联合类型的成员采用了覆盖存储技术），可以在实现多态性程序的同时节约内存空间，程序的多态性可以提高程序的可读性。
- ❁ 栈是一种数据结构，其实现可以看出过程抽象与数据抽象方法的优势。
- ❁ 程序执行过程中，数据一般以变量、字符串等形式存在，存储在栈中；当数据量比较大或需要长期保存时，则往往存储在文件中。（程序的代码也是以文件的形式保存的。另外，C/C++语言把标准输入/输出设备，比如键盘、显示器和打印机，也看成一种文件。）



要求：

- ◆ 了解以上概念和应用
- ◆ 掌握
 - 函数指针、结构数组、联合、栈的特点
 - 文件的打开、关闭、读/写等操作方法与注意事项
 - 一个程序代码量 ≈ 100 行
- ◆ 继续保持良好的编程习惯

Thanks!

