

§ 2. 线性表

2. 2. 线性表的顺序表示和实现

2. 2. 2. 线性表顺序表示的基本操作的实现

2. 2. 2. 2. C++语言版

★ 程序的组成

- `linear_list_sq.h` : 头文件
- `linear_list_sq.cpp` : 具体实现
- `linear_list_sq_main.cpp` : 使用（测试）示例

★ 相比C的好处

- `class`既可以定义成员，又可以有成员函数
- 相比C语言自己控制`InitList`和`DestroyList`的调用时机（可能产生各种错误），C++的构造函数和析构函数会被系统自动调用
- 有继承机制，易于扩充

§ 2. 线性表

2. 2. 线性表的顺序表示和实现

2. 2. 2. 线性表顺序表示的基本操作的实现

2. 2. 2. 2. C++语言版

★ ElemType => int 的实现过程

/ linear_list_sq.h 的组成 */*

```
#define TRUE          1
#define FALSE        0
#define OK           1
#define ERROR        0
#define INFEASIBLE   -1
#define OVERFLOW     -2
```

```
typedef int Status;
```

P. 10 预定义常量和类型

```
#define LIST_INIT_SIZE 100    //初始大小，可按需修改
#define LISTINCREMENT 10     //空间不够时每次
```

```
typedef int ElemType;        //可根据需要修改元素的类型
```

P. 22描述的补充

/* linear_list_sq.h 的组成 */

```
class sqliist {
```

```
    protected:
```

```
        ElemType *elem;
```

```
        int length;
```

```
        int listsize;
```

```
    public:
```

```
        sqliist();    //构造函数，替代InitList
```

```
        ~sqliist();  //析构函数，替代DestroyList
```

```
        Status ClearList();
```

```
        Status ListEmpty();
```

```
        int ListLength();
```

```
        Status GetElem(int i, ElemType &e);
```

```
        int LocateElem(ElemType e,
```

```
                        Status (*compare)(ElemType e1, ElemType e2));
```

```
        Status PriorElem(ElemType cur_e, ElemType &pre_e);
```

```
        Status NextElem(ElemType cur_e, ElemType &next_e);
```

```
        Status ListInsert(int i, ElemType e);
```

```
        Status ListDelete(int i, ElemType &e);
```

```
        Status ListTraverse(Status (*visit)(ElemType e));
```

```
};
```

P. 19-20 抽象数据类型定义
转换为实际的C++语言定义

★ 需要改变值的参数都用引用
★ 成员函数形式，参数都不带L
（表达为L的成员函数形式）

/ linear_list_sq.cpp 的实现 */*

```
#include <iostream>
#include <stdlib.h>           //malloc/realloc函数
#include <unistd.h>           //exit函数
#include "linear_list_sq.h"   //形式定义
using namespace std;
```

/ 构造函数，替代 InitList */*

```
sqlist::sqlist()
{
    elem = new ElemType[LIST_INIT_SIZE];
    if (elem == NULL)
        exit(OVERFLOW);
    length = 0;
    listsize = LIST_INIT_SIZE;
}
```

★ 参数不带L
★ elem/length/listsize
等成员也不是L->*或L.*
而是隐含的this指针

```
/* linear_list_sq.cpp 的实现 */
```

```
/* 析构函数，替代 DestroyList */
```

```
sqlist::~~sqlist()
```

```
{
```

```
    if (elem)
```

```
        delete elem;
```

```
    length = 0;
```

```
    listsize = 0;
```

```
}
```

```
/* linear_list_sq.cpp 的实现 */
```

```
/* 清除线性表（已初始化，不释放空间，只清除内容） */
```

```
Status sqlist::ClearList()
```

```
{
```

```
    length = 0;
```

```
    return OK;
```

```
}
```

```
/* linear_list_sq.cpp 的实现 */
```

```
/* 判断是否为空表 */
```

```
Status sqlist::ListEmpty()
```

```
{
```

```
    if (length == 0)
```

```
        return TRUE;
```

```
    else
```

```
        return FALSE;
```

```
}
```



```
/* linear_list_sq.cpp 的实现 */
```

```
/* 求表的长度 */
```

```
int sqlist::ListLength()  
{  
    return length;  
}
```

/* linear_list_sq.cpp 的实现 */

/* 取表中元素 */

Status sqliist::GetElem(int i, ElemType &e)

{

if (i<1 || i>length) //不需要多加 || L.length>0
return ERROR;

e = elem[i-1]; //下标从0开始, 第i个实际在elem[i-1]中
return OK;

}

形参是ElemType &e
因此 e=elem[i-1];
不需要 *e

```
/* linear_list_sq.cpp 的实现 */
```

```
/* 查找符合指定条件的元素 */
```

```
int sqliist::LocateElem(ElemType e, Status (*compare)(ElemType e1, ElemType e2))  
{  
    ElemType *p = elem;  
    int      i = 1;  
  
    while(i<=length && (*compare)(*p++, e)==FALSE)  
        i++;  
  
    return (i<=length) ? i : 0;    //找到返回i, 否则返回0  
}
```

/* linear_list_sq.cpp 的实现 */

/* 查找符合指定条件的元素的前驱元素 */

Status sqliist::PriorElem(ElemType cur_e, ElemType &pre_e)

{

ElemType *p = elem;

int i = 1;

/* 循环比较整个线性表 */

while(i<=length && *p!=cur_e) {

i++;

p++;

}

if (i==1 || i>length) //找到第1个元素或未找到

return ERROR;

pre_e = *--p;

//取前驱元素的值

return OK;

}

形参是ElemType &pre_e
因此 pre_e=*--p;
不需要 *pre_e

/* linear_list_sq.cpp 的实现 */

/* 查找符合指定条件的元素的后继元素 */

Status sqliist::NextElem(ElemType cur_e, ElemType &next_e)

{

ElemType *p = elem;

int i = 1;

/* 循环比较整个线性表(不含尾元素) */

while(i<length && *p!=cur_e) {

i++;

p++;

}

if (i>=length) //未找到 (最后一个元素未比较)

return ERROR;

next_e = *++p; //取后继元素的值

return OK;

}

形参是ElemType &next_e
因此 next_e=*++p;
不需要 *next_e

/* linear_list_sq.cpp 的实现 */

Status sqlist::ListInsert(int i, ElemType e)

{ ElemType *p, *q; //如果是算法，一般可以省略，程序不能

if (i<1 || i>length+1) //合理范围是 1..length+1

return ERROR;

/* 空间已满则扩大空间 */

if (length >= listsize) {

ElemType *newbase;

newbase = new ElemType[listsize+LISTINCREMENT];

if (!newbase)

return OVERFLOW;

/* newbase大小为listsize+LISTINCREMENT，复制前listsize个 */
memcpy(newbase, elem, listsize*sizeof(ElemType));

delete elem;

elem = newbase;

listsize += LISTINCREMENT;

//length暂时不变

}

q = &(elem[i-1]); //第i个元素，即新的插入位置

/* 从最后（length放在[length-1]中）开始到第i个元素依次后移 */

for (p=&(elem[length-1]); p>=q; --p)

*(p+1) = *p;

/* 插入新元素，长度+1 */

*q = e;

length ++;

return OK;

}

```
/* linear_list_sq.cpp 的实现 */
/* 删除指定位置的元素，并将被删除元素的值放入e中返回 */
Status sqlist::ListDelete(int i, ElemType &e)
{
    ElemType *p, *q; //如果是算法，一般可以省略，程序不能
    if (i<1 || i>length) //合理范围是 1..length
        return ERROR;

    p = &(elem[i-1]); //指向第i个元素
    e = *p; //取第i个元素的值放入e中
    q = &(elem[length-1]); //指向最后一个元素
    //也可以 q = L->elem+L->length-1

    /* 从第i+1到最后，依次前移一格 */
    for (++p; p<=q; ++p)
        *(p-1) = *p;
    length --; //长度-1
    return OK;
}
```

`/* linear_list_sq.cpp 的实现 */`

`/* 遍历线性表 */`

```
Status sqliist::ListTraverse(Status (*visit)(ElemType e))
{
    extern int line_count; //main中定义的换行计数器(与算法无关)
    ElemType *p = elem;
    int      i = 1;

    line_count = 0;           //计数器恢复初始值(与算法无关)
    while(i<=length && (*visit)(*p++)==TRUE)
        i++;
    if (i<=length)
        return ERROR;

    cout << endl; //最后打印一个换行，只是为了好看，与算法无关
    return OK;
}
```


/* linear_list_sq_main.cpp 中 MyCompare和MyVisit的实现 */

```
Status MyCompare(ElemType e1, ElemType e2)
```

```
{
```

```
    if (e1==e2)
```

```
        return TRUE;
```

```
    else
```

```
        return FALSE;
```

```
}
```

```
Status MyVisit(ElemType e)
```

```
{
```

```
    cout << setw(3) << e << "->";    //此句输出
```

```
    /* 每输出20个，打印一个换行 */
```

```
    if ((++line_count)%20 == 0)
```

```
        cout << endl;
```

```
    return OK;
```

```
}
```

§ 2. 线性表

2.3. 线性表的链式表示和实现

2.3.2. 线性表链式表示的基本操作的实现

2.3.2.2. C++语言版

★ 程序的组成

- `linear_list_L.h` : 头文件
- `linear_list_L.cpp` : 具体实现
- `linear_list_L_main.cpp` : 使用（测试）示例

★ `ElemType => int` （带头结点）

/ linear_list_L.h 的组成 */*

```
#define TRUE      1
#define FALSE    0
#define OK       1
#define ERROR    0
#define INFEASIBLE -1
#define OVERFLOW -2
```

```
typedef int Status;
```

P. 10 预定义常量和类型

无变化

```
class LinkList;    //提前声明，因为定义友元要用到
```

```
class LNode {
    protected:
        ElemType data;    //数据域
        LNode *next;      //指针域
    public:
```

```
        friend class LinkList;
```

定义为LinkList的友元，可以任意访问

//不定义任何函数，相当于struct LNode

```
};
```

/ linear_list_L.h 的组成 */*

```
class LinkList {  
    protected:  
        LNode *head;    //头指针  
    public:  
        LinkList();      //构造函数, 替代InitList  
        ~LinkList();     //析构函数, 替代DestroyList  
        Status ClearList();  
        Status ListEmpty();  
        int ListLength();  
        Status GetElem(int i, ElemType &e);  
        int LocateElem(ElemType e,  
                        Status (*compare)(ElemType e1, ElemType e2));  
        Status PriorElem(ElemType cur_e, ElemType &pre_e);  
        Status NextElem(ElemType cur_e, ElemType &next_e);  
        Status ListInsert(int i, ElemType e);  
        Status ListDelete(int i, ElemType &e);  
        Status ListTraverse(Status (*visit)(ElemType e));  
};
```

除构造/析构函数名外, 所有
与顺序表的C++定义完全相同

```
/* linear_list_L.cpp 的实现 */
#include <iostream>
#include <cstdlib>
#include "linear_list_L.h" //形式定义

using namespace std;

/* 构造函数（初始化线性表） */
LinkList::LinkList()
{
    /* 申请头结点空间，赋值给头指针 */
    head = new LNode;
    if (head == NULL)
        exit(OVERFLOW);

    head->next = NULL;
}
```

```
/* linear_list_L.cpp 的实现 */
```

```
/* 析构函数（删除线性表） */
```

```
LinkedList::~LinkedList()
```

```
{
```

```
    LNode *q, *p = head;
```

```
    /* 从头结点开始依次释放（含头结点） */
```

```
    while(p) {
```

```
        q=p->next; //抓住链表的下一个结点
```

```
        delete p;
```

```
        p=q;
```

```
    }
```

```
    head = NULL; //头指针置NULL
```

```
}
```

```
/* linear_list_L.cpp 的实现 */
```

```
/* 清除线性表（保留头结点） */
```

```
Status LinkList::ClearList()
```

```
{
```

```
    LNode *q, *p = head->next;
```

```
    /* 从首元结点开始依次释放 */
```

```
    while(p) {
```

```
        q = p->next;    //抓住链表的下一个结点
```

```
        delete p;
```

```
        p = q;
```

```
    }
```

```
    head->next = NULL; //头结点的next域置NULL
```

```
    return OK;
```

```
}
```

```
/* linear_list_L.cpp 的实现 */
```

```
/* 判断是否为空表 */
```

```
Status LinkList::ListEmpty()
```

```
{
```

```
    /* 判断头结点的next域即可 */
```

```
    if (head->next==NULL)
```

```
        return TRUE;
```

```
    else
```

```
        return FALSE;
```

```
}
```



```
/* linear_list_L.cpp 的实现 */
```

```
/* 求表的长度 */
```

```
int LinkList::ListLength()
```

```
{
```

```
    LNode *p = head->next; //指向首元结点
```

```
    int    len = 0;
```

```
    /* 循环整个链表，进行计数 */
```

```
    while(p) {
```

```
        p = p->next;
```

```
        len++;
```

```
    }
```

```
    return len;
```

```
}
```

```
/* linear_list_L.cpp 的实现 */
```

```
/* 取表中元素 */
```

```
Status LinkList::GetElem(int i, ElemType &e)
```

```
{   LNode *p = head->next; //指向首元结点
```

```
    int pos = 1;           //初始位置为1
```

```
/* 链表不为NULL 且 未到第i个元素 */
```

```
while(p!=NULL && pos<i) {
```

```
    p=p->next;
```

```
    pos++;
```

```
}
```

```
if (!p || pos>i)
```

```
    return ERROR;
```

```
e = p->data;
```

```
return OK;
```

```
}
```

形参是ElemType &e
因此 e=p->data;
不需要 *e

```
/* linear_list_L.cpp 的实现 */
```

```
/* 查找符合指定条件的元素 */
```

```
int LinkList::LocateElem(ElemType e, Status (*compare)(ElemType e1, ElemType e2))
```

```
{
```

```
    LNode *p = head->next; //指向首元结点
```

```
    int pos = 1;           //初始位置为1
```

```
    /* 循环整个链表 */
```

```
    while(p && (*compare)(e, p->data)==FALSE) {
```

```
        p=p->next;
```

```
        pos++;
```

```
    }
```

```
    return p ? pos:0;
```

```
}
```

/* linear_list_L.cpp 的实现 */

/* 查找符合指定条件的元素的前驱元素 */

Status LinkList::PriorElem(ElemType cur_e, ElemType &pre_e)

{

 LNode *p = head->next; //指向首元结点

 if (p==NULL) //空表直接返回

 return ERROR;

 /* 从第2个结点开始循环整个链表(如果比较相等, 保证有前驱) */

 while(p->next && p->next->data != cur_e)

 p = p->next;

 if (p->next==NULL) //未找到

 return ERROR;

 pre_e = p->data;

 return OK;

}

形参是ElemType &pre_e
因此 pre_e = p->data;
不需要 *pre_e

```
/* linear_list_L.cpp 的实现 */
```

```
/* 查找符合指定条件的元素的后继元素 */
```

```
Status LinkList::NextElem(ElemType cur_e, ElemType &next_e)
```

```
{
```

```
    LNode *p = head->next;  //首元结点
```

```
    if (p==NULL)             //空表直接返回
```

```
        return ERROR;
```

```
    /* 有后继结点且当前结点值不等时继续 */
```

```
    while(p->next && p->data!=cur_e)
```

```
        p = p->next;
```

```
    if (p->next==NULL)
```

```
        return ERROR;
```

```
    next_e = p->next->data;
```

```
    return OK;
```

```
}
```

```
/* linear_list_L.cpp 的实现 */
/* 在指定位置前插入一个新元素 */
Status LinkList::ListInsert(int i, ElemType e)
{
    LNode *s, *p = head;    //p指向头结点
    int pos = 0;
    /* 寻找第i-1个结点 */
    while(p && pos<i-1) {
        p=p->next;
        pos++;
    }
    if (p==NULL || pos>i-1) //i值非法则返回
        return ERROR;
    //执行到此表示找到指定位置，p指向第i-1个结点
    s = new LNode;
    if (s==NULL)
        return OVERFLOW;

    s->data = e;           //新结点数据域赋值
    s->next = p->next;      //新结点的next是第i个
    p->next = s;           //第i-1个的next是新结点
    return OK;
}
```

```
/* linear_list_L.cpp 的实现 */
/* 删除指定位置的元素，并将被删除元素的值放入e中返回 */
Status LinkList::ListDelete(int i, ElemType &e)
{
    LNode *q, *p = head;    //p指向头结点
    int pos = 0;
    /* 寻找第i个结点 (p->next是第i个结点) */
    while(p->next && pos<i-1) {
        p=p->next;
        pos++;
    }
    if (p->next==NULL || pos>i-1)    //i值非法则返回
        return ERROR;
    //执行到此表示找到了第i个结点，此时p指向第i-1个结点
    q = p->next;    //q指向第i个结点
    p->next = q->next; //第i-1个结点的next域指向第i+1个

    e = q->data;    //取第i个结点的值
    delete q;    //释放第i个结点

    return OK;
}
```

/ linear_list_L.cpp 的实现 */*

```
Status LinkList::ListTraverse(Status (*visit)(ElemType e))
{
    extern int line_count; //main中定义的换行计数器(与算法无关)
    LNode *p = head->next;

    line_count = 0;          //计数器恢复初始值(与算法无关)
    while(p && (*visit)(p->data)==TRUE)
        p=p->next;

    if (p)
        return ERROR;

    cout << endl; //最后打印一个换行，只是为了好看，与算法无关
    return OK;
}
```


§ 2. 线性表

2.5. 算法转换为程序过程中的问题与技巧(补充)

2.5.3. 相同算法中程序适应不同的数据类型

2.5.3.1. C语言对不同数据类型的适应方法

例：线性表的顺序实现(六种数据类型)

方法一：用六个程序实现，每个程序仅不同数据类型的处理细节不同

方法二：用一个程序实现，用条件编译来区分不同数据类型的处理细节

2.5.3.2. C++语言的处理方法

例：线性表的顺序实现(六种数据类型)

方法一：同C方法一（六个不同的程序）

方法二：同C方法二（一个程序，改宏定义分别编译）

方法三：函数模板及类模板

★ 用类模板实现同一程序适应不同的数据类型

/ linear_list_sq.h 的组成 */*

```
#define TRUE          1
#define FALSE         0
#define OK            1
#define ERROR         0
#define INFEASIBLE    -1
#define OVERFLOW      -2
```

以ElemType为int的C++程序为基准，
比较两者的不同

```
typedef int Status;
```

```
#define LIST_INIT_SIZE 100    //初始大小，可按需修改
#define LISTINCREMENT 10     //空间不够时每次
```

```
template <class ElemType> //声明模板, 虚类型ElemType
```

/* linear_list_sq.h 的组成 */

```
class sqliist {  
    protected:  
        ElemType *elem;  
        int length;  
        int listsize;  
    public:  
        sqliist();    //构造函数, 替代InitList  
        ~sqliist();   //析构函数, 替代DestroyList  
        Status  ClearList();  
        Status  ListEmpty();  
        int     ListLength();  
        Status  GetElem(int i, ElemType &e);  
        int     LocateElem(ElemType e,  
                           Status (*compare)(ElemType e1, ElemType e2));  
        Status  PriorElem(ElemType cur_e, ElemType &pre_e);  
        Status  NextElem(ElemType cur_e, ElemType &next_e);  
        Status  ListInsert(int i, ElemType e);  
        Status  ListDelete(int i, ElemType &e);  
        Status  ListTraverse(Status (*visit)(ElemType e));  
};
```

不变

/ linear_list_sq.c 的实现 */*

```
#include <iostream>
#include <stdlib.h>           //malloc/realloc函数
#include <unistd.h>           //exit函数
#include "linear_list_sq.h"   //形式定义
using namespace std;
```

/ 构造函数，替代 InitList */*

```
template <class ElemType>
sqlist<ElemType>::sqlist()
{
```

原来: `sqlist::sqlist()`

```
    elem = new ElemType[LIST_INIT_SIZE];
    if (elem == NULL)
        exit(OVERFLOW);
    length = 0;
    listsize = LIST_INIT_SIZE;
```

```
}
```

函数的实现过程
没有任何变化
所有函数均如此

/* linear_list_sq.c 的实现 */

其它函数略

/* 遍历线性表 */

Status sqlist::ListTraverse(Status (*visit)(ElemType e))

{

实现过程略

}

/* 遍历线性表 */

template <class ElemType>

Status sqlist<ElemType>::ListTraverse(Status (*visit)(ElemType e))

{

实现过程略

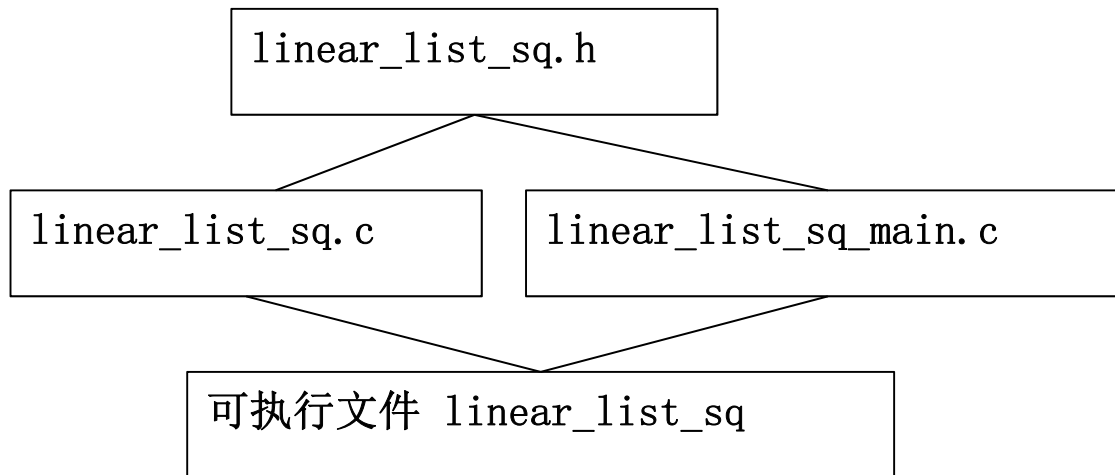
}

`/* linear_list_sq_main.c 的实现 */`

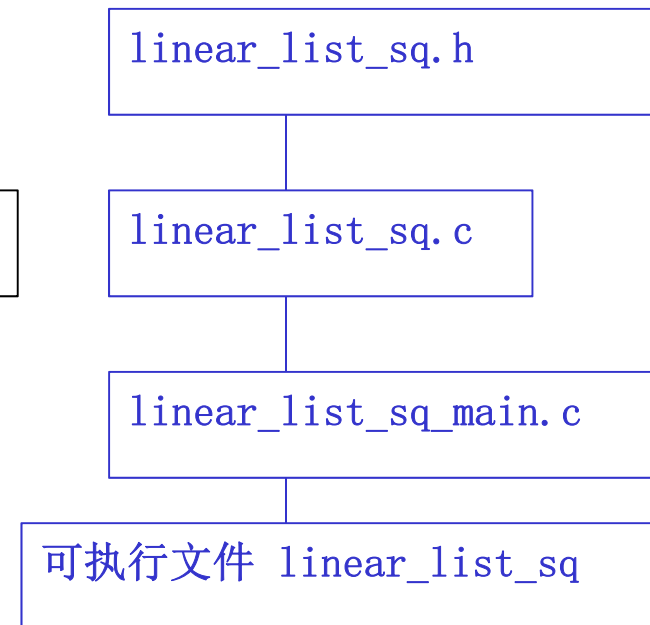
```
#include <iostream>
#include <iomanip>
#include "linear_list_sq.cpp"
```

不能是linear_list_sq.h
因为类模板规定定义和实现
必须在同一个文件中

```
using namespace std;
```



工程文件中包含 2个源程序 + 1个头文件



自行研究工程文件中应该如何包含?

/* linear_list_sq_main.c 中 MyCompare和MyVisit的实现 */

```
template <class ElemType>
```

```
Status MyCompare(ElemType e1, ElemType e2)
```

```
{
```

具体实现略

```
}
```

```
template <class ElemType>
```

```
Status MyVisit(ElemType e)
```

```
{
```

具体实现略

```
}
```

虽然不是类的成员函数
也可以用类模板

/* linear_list_sq_main.c 中 main函数 */

//此处不能是template <class ElemType>, 类模板不适用于main函数

int main()

{

sqlist<int> L; //类模板实例化

int e1, e2; //不能是ElemType e1, e2, 只能具体类型

//从此处往下, 无任何变化

int i, pos;

cout << "空表=" << ((L.ListEmpty() == TRUE) ? "TRUE" : "FALSE") << endl;

cout << "表长=" << L.ListLength() << endl;

后续略

}

换为 sqlist <long> L;
sqlist <short> L;
均可以
下面 int e1, e2;
同步变化即可

§ 2. 线性表

2.5. 算法转换为程序过程中的问题与技巧(补充)

2.5.3. 相同算法中程序适应不同的数据类型

2.5.3.1. C语言对不同数据类型的适应方法

2.5.3.2. C++语言的处理方法

例：线性表的顺序实现(六种数据类型)

方法一：同C方法一（六个不同的程序）

方法二：同C方法二（一个程序，改宏定义分别编译）

方法三：函数模板及类模板

- ★ 用类模板实现同一程序适应不同的数据类型

- ★ 参数个数、实现方式必须完全相同，否则不适用

 - int/long/short 适用

 - int/double 不适用

- ★ 参数个数、实现方式必须完全相同，否则不适用

- ★ 类模板的定义和实现必须放在同一头文件中

§ 2. 线性表

2.5.3.2. C++语言的处理方法

例：线性表的顺序实现(六种数据类型)

方法三：函数模板及类模板

讨论1：能否用类模板实现 `int/double` 的统一处理？

● 比较时有差别 `e1==e2` `fabs(e1-e2)<1e-6`

● 其它无差别

```
class double1 {  
    protected:  
        double x;  
    public:  
        int operator==(double1 &d1)  
        {   if (fabs(x-d1.x) < 1e-6)  
                return 1;  
            else  
                return 0;  
        }  
        //其它需要的成员函数  
}
```

`int/double1` 能统一处理

§ 2. 线性表

2.5.3.2. C++语言的处理方法

例：线性表的顺序实现(六种数据类型)

方法三：函数模板及类模板

讨论2：能否用类模板实现

int/double/char数组/char指针/struct student/struct student *

的统一处理？

- char数组：比较strcmp / 赋值strcpy
- char指针：比较strcmp / 赋值strcpy / 申请释放
- student：比较仅学号 / 赋值memcpy
- student*：比较仅学号 / 赋值memcpy / 申请释放

答：(1) char数组和指针都没必要了，直接用string类，使strcpy/strcmp变为 s1=s2, s1==s2

(2) struct student 改为 class student, 重载赋值、比较、输入输出等运算符即可

使 memcpy 变为 stu1=stu2 / 而 stu1==stu2 仅比较学号

继续讨论：student和student*在细节处理上还是有所不同，如何统一？

例：GetElem中 memcpy(e, &(L.elem[i-1]), sizeof(ElemType));
 memcpy(*e, L.elem[i-1], sizeof(ET));

答：两者在实际应用没有区别，目前的讨论仅限于程序内部处理机制，因此只选一种即可

继续讨论：选哪种？

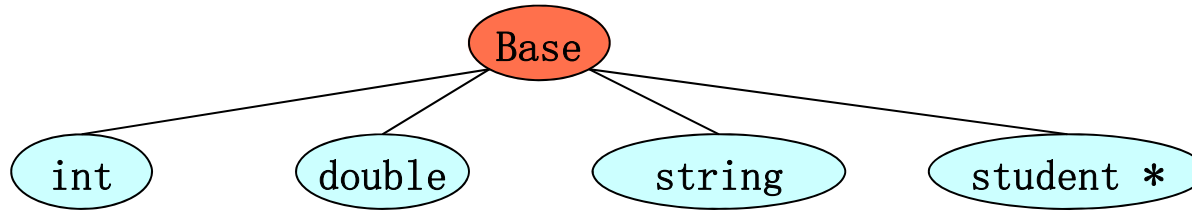
答：指针方式，内存使用更灵活

§ 2. 线性表

2.5.3.2. C++语言的处理方法

例：线性表的顺序实现(六种数据类型)

方法四：多态性与虚函数(是否可以适应多数据类型?)



```
class sqliist_base {};  
class sqliist_int : public sqliist_base {};  
class sqliist_double : public sqliist_base {};  
class sqliist_string : public sqliist_base {};  
class sqliist_student_p : public sqliist_base {};
```

右侧例：无参函数适合、带参函数不适合)

思考：可不可以通过什么方法使合适？

(可能需要自行补充额外的C++的知识)

```
sqliist_base *p;  
sqliist_int Li;  
sqliist_double Ld;  
p = &Li;  
p->length();  
p = &Ld;  
p->length();
```

适合

```
sqliist_base *p;  
sqliist_int Li;  
int i=1, e;  
p = &Li;  
p->GetElem(i, e);
```

不适合