

第五章 构造数据类型



本章内容

- 概述
- 枚举类型
- 数组类型
- 结构类型
- 联合类型
- 指针类型
- 引用类型



构造数据类型

- 有些数据不适合用基本数据类型来表示。
- 语言往往提供了由基本数据类型来构造新类型的手段。
- 构造数据类型属于用户自定义数据类型。

枚举类型 (1)

- 用户自定义的简单数据类型。
- 类型定义格式
 - `enum <枚举类型名> {<枚举值表>;`
 - 例如：`enum Day{SUN, MON, TUE, WED}`
`enum Month{JAN, FEB, MAR}`
- `bool`类型可看成C++语言提供的枚举类型：
`enum bool { false, true };`

枚举类型 (2)

枚举值

- 每一个枚举值都对应一个整数。
- 默认情况下，第一个枚举值对应常量值0，其它的值
为前一个值加1。
- 定义时，可给枚举值指定对应的整数值。

例如：

```
enum Day {SUN=7,MON=1,TUE,WED,THU,FRI,SAT};
```

枚举类型 (1)

变量定义格式

- <枚举类型名> <枚举类型变量名>;
- enum <枚举类型名> <枚举类型变量名>; (C写法)
- enum <枚举类型名> {<枚举值表>} <枚举类型变量名>;

例如: enum Day {SUN, MON, TUE} d1, d2;

- enum {<枚举值表>} <枚举类型变量名>;

例如: enum {V1, V2, V3} v1, v2;



枚举类型运算 (1)

○ 赋值

- 只在相应枚举类型的值集中取值。

例如：

```
Day day;  
day = SUN; //OK  
day = 1; //Error  
day = RED; //Error
```

- 相同枚举类型之间可进行赋值操作，

例如：

```
Day d1,d2;  
d2 = d1;
```



枚举类型运算 (2)

赋值

- 可把枚举值赋值给整型变量，但不能把整型数赋值给枚举类型的变量

- 例如：

```
int a;
```

```
a = d1; //OK
```

```
d1 = a; //Error
```

```
d1 = (Day)a; //OK, 但不安全
```


枚举类型运算 (3)

比较运算

- 转化为枚举值所对应的整数之间的比较。
- 例如, `MON < TUE` (结果为`true`)

算术运算

- 运算时, 枚举值将转换成对应的整型值。
- 对枚举类型进行算术运算的结果类型为算术类型。
- 例如:

`Day d;`

`d = d+1; //Error, 因为d+1的结果为int类型。`

`d = (Day)(d+1) //OK`

枚举类型运算 (4)

- 不能对枚举类型变量直接进行输入

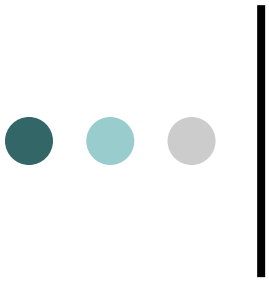
- 例如,

```
Day d;
```

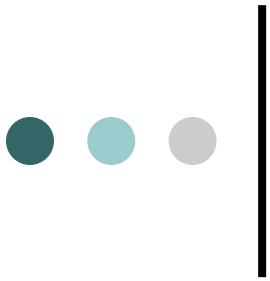
```
cin >> d; //Error
```

```
cout << d; //Ok
```

- 通过输入整数值把该整型值转化为枚举值。



```
#include <iostream>
using namespace std;
int main()
{   Day d;
    int i;
    cin >> i;
    switch (i)
    {   case 0: d = SUN;           break;
        case 1: d = MON;         break;
        case 2: d = TUE;         break;
        case 3: d = WED;         break;
        case 4: d = THU;         break;
        case 5: d = FRI;         break;
        case 6: d = SAT;         break;
        default: cout << "Input Error!" << endl; exit(-1);
    }
}
```



.....

switch (d)

```
{    case SUN:      cout << "SUN" << endl;      break;
    case MON:      cout << "MON" << endl;      break;
    case TUE:      cout << "TUE" << endl;      break;
    case WED:      cout << "WED" << endl;      break;
    case THU:      cout << "THU" << endl;      break;
    case FRI:      cout << "FRI" << endl; break;
    case SAT:      cout << "SAT" << endl;      break;
}
```

return 0;

}



数组类型

- 如何表示类似于向量和矩阵的复合数据？
- C++提供了数组类型来表示上述的数据：
 - 数组类型是一种由固定多个同类型的元素按一定次序所构成的复合数据类型。
 - 数组类型是一种用户自定义的数据类型。



一维数组的定义

- 表示由固定多个同类型的具有线性次序关系的数据所构成的复合数据类型。
- 定义格式：
 - `<元素类型> <一维数组变量名> [<元素个数>;`
例如： `int a[10];`
 - `typedef <元素类型> <一维数组类型名> [<元素个数>;`
例如： `typedef int A[10];`
`A a;`
 - `<元素个数>`为整型常量表达式

一维数组的操作 (1)

访问一维数组元素

- `<一维数组变量名>[<下标>]`
- `<下标>`为整型表达式，第一个元素的下标为0

例如： `int a[10];` //数组a

`a[0]、a[1]、...、a[9]` //数组元素

- C++语言不对数组元素下标越界进行检查。程序员必须仔细处置这个问题！

一维数组的操作 (2)

- 可把数组的每个元素看成是独立的变量。
- 不能对两个数组进行整体赋值，需要通过元素来进行：

例如： `int a[10],b[10];`

.....

`a = b; //Error`

`for (int i=0; i<10; i++) a[i] = b[i]; //OK`

用一维数组实现求第n个费波那契(Fibonacci)数

```
#include <iostream>
using namespace std;
int main()
{   const int MAX_N=40;
    int fibs[MAX_N];
    int n,i;
    cout << "请输入n(1-" << MAX_N << "):";
    cin >> n;
    if (n > MAX_N)
    {   cout << "n太大! 应不大于" << MAX_N << endl;
        return -1;
    }
    fibs[0] = fibs[1] = 1;
    for (i=2; i<n; i++) //计算费波那契数
        fibs[i] = fibs[i-1] + fibs[i-2];
    cout << "第" << n << "个费波那契数是: " << fibs[n-1] << endl;
    return 0;
}
```

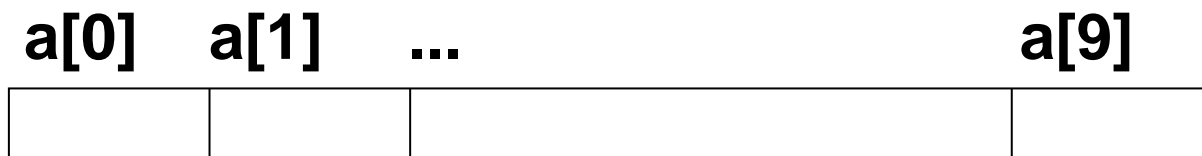
一维数组的初始化

- 用花括号把元素的初始值括起来。
 - 例如： `int a[10]={1,2,3,4,5,6,7,8,9,10};`
- 初始化表中的值可以少于数组元素个数，不足部分的数组元素初始化成0。
 - 例如： `int b[10]={1,2,3,4};` //b[4]~b[9]为0
- 如果每个元素都进行了初始化，则数组元素个数可以省略。
 - 例如： `int c[]={1,2,3};` //因含着c由三个元素构成

一维数组的存储分配

- 编译程序将会在内存中给其分配连续的存储空间来存储数组元素。

- 例如： `int a[10];` 其内存空间分配如下：



- 所占内存空间大小用sizeof操作符来计算。

- 例如：

```
int a[10];
```

```
cout << sizeof(a); //输出数组a所占的内存字节数。
```

向函数传递一维数组 (1)

- 形参为不带数组大小的一维数组定义以及数组元素的个数。

- 例如：

```
int max(int x[], int num)
{ int i,j;
  j = 0;
  for (i=1; i<num; i++)
    if (x[i] > x[j]) j = i;
  return j;
}
```

向函数传递一维数组 (2)

- 实参为一维数组变量的名以及数组元素的个数

- 例如： `int a[10],b[20],index_max;`

.....

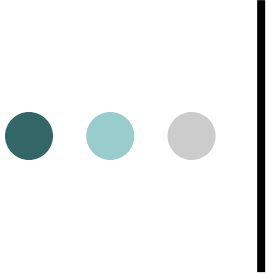
```
index_max = max(a,10);
```

```
cout << a[index_max] << index_max << endl;
```

```
index_max = max(b,20);
```

```
cout << b[index_max] << index_max << endl;
```

- 数组作为函数参数传递时实际传递的是数组的首地址，函数的形参数组不再分配内存空间，它共享实参数组的内存空间。



字符数组（字符串）

- C++语言本身没有提供字符串类型。
- 用元素类型为char的一维数组（字符数组）来表示字符串类型。
 - 例如：char s[10];
 - 用字符数组存储字符串时，通常在最后一个字符的后面放置字符串结束标记：'\0'。
 - 定义时，元素个数应比实际能够存储的字符串最大长度多一个。



编写函数把由数字构成的字符串转换成整型数

```
#include <iostream>
#include <cstring>
using namespace std;
int str_to_int(char str[])
{ int n=0;
    for (int i=0; str[i] != '\0'; i++)
        n = n*10+(str[i]-'0');
    return n;
}
```



字符数组的初始化

- 有下面几种形式

- `char s[10]={'h','e','l','l','o','\0'};`
- `char s[10]="hello";`
- `char s[10]="hello";`
- `char s[]="hello";`

- 除了第一种形式，其它形式的初始化都会在最后—一个字符的后面自动加上' \0'，而对于第一种形式，程序中必须显式地加上' \0'。



标准库中的字符串处理函数

○ 计算字符串的长度

```
int strlen( const char s[] );
```

○ 字符串复制

```
char *strcpy( char dst[], const char src[] );
```

```
char *strncpy( char dst[], const char src[], int n );
```

○ 字符串拼接

```
char *strcat( char dst[], const char src[] );
```

```
char *strncat( char dst[], const char src[], int n );
```

标准库中的字符串处理函数

字符串比较

```
int strcmp( const char s1[], const char s2[] );
```

```
int strncmp( const char s1[], const char s2[], int n );
```

从字符串到数值类型转换的函数

```
double atof( const char s[] );
```

```
int atoi( const char s[] );
```

```
long atol( const char s[] );
```

```
string -> const char s[] 使用 *.c_str()
```



把从键盘输入的字符串逆向输出

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{   const int MAX_LEN=100;
    char str[MAX_LEN];
    cin >> str; //输入
    int len = strlen(str); //str中的字符个数
    for (int i=0,j=len-1; i<len/2; i++,j--)
    {   char temp;
        temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }
    cout << str << endl;
    return 0;
}
```



二维数组

- 表示由固定多个同类型的具有行列结构的数据所构成的复合数据。
- 所表示的是一种具有两维结构的数据，第一维称为二维数组的行，第二维称为二维数组的列。二维数组的每个元素由其所在的行和列唯一确定。

二维数组的定义

定义格式:

- `<元素类型> <二维数组变量>[<行数>][<列数>];`

例如: `int a[10][5];`

- `typedef <元素类型> <二维数组类型名>[<行数>][<列数>];`

例如: `typedef int A[10][5];`

`A a;`

- `<行数>`和`<列数>`为整型常量表达式

二维数组的操作 (1)

访问二维数组元素：

- `<二维数组变量名>[<下标1>][<下标2>]`
- `<下标1>`和`<下标2>`为整型表达式，均从0开始。

- 例如： `int a[10][5];`

`a[0][0]`、`a[0][1]`、...、`a[9][0]`、...、`a[9][4]`

以行为单位访问：

- 例如： `int a[10][5];`

`a[0]`、`a[1]`、...、`a[9]`

每个都是一维数组，代表二维数组中的一行



二维数组的操作 (2)

- 对二维数组的操作通常是通过其元素来进行。

- 例如: `int a[10][5],sum=0;`

.....

```
for (int i=0; i<10; i++)
```

```
    for (int j=0; j<5; j++)
```

```
        sum += a[i][j];
```

- 例如: `int a[10][5];`

其内存空间分配如下:

[illegible]

向函数传递二维数组 (1)

- 形参为不带数组行数的二维数组定义及其行数。

- 例如： `int sum(int x[][5], int lin)` //接收lin行、5列的二维数组

```
{ int s=0;
  for (int i=0; i<lin; i++)
    for (int j=0; j<5; j++)
      s += x[i][j];
  return s; }
```

- 参数传递时传递的是数组的首地址。二维数组形参的列数必须要写，否则，无法计算`x[i][j]`的内存地址。

- $x[i][j]$ 的地址 = x 的首地址 + $i \times \text{列数} + j$

向函数传递二维数组 (2)

- 实参为二维数组变量 (列数要与形参相同) 的名和行数。

- 例如: `int a[10][5], b[20][5];`

.....

```
cout << "a的和为: " << sum(a,10) << endl;
```

```
cout << "b的和为: " << sum(b,20) << endl;
```

- 又如: `int c[40][20];`

.....

```
sum(c,40); //Error
```



二维数组降为一维数组处理

```
int sum(int x[], int num)
{   int s=0;
    for (int i=0; i<num; i++) s += x[i];
    return s;
}
```

.....

```
int a[10][5], b[20][5], c[40][20];
```

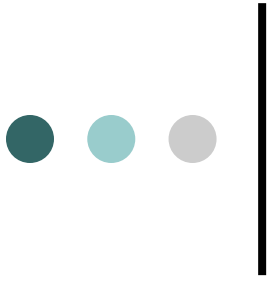
.....

```
cout << sum(a[0], 10*5) << endl;
cout << sum(b[0], 20*5) << endl;
cout << sum(c[0], 40*20) << endl;
```



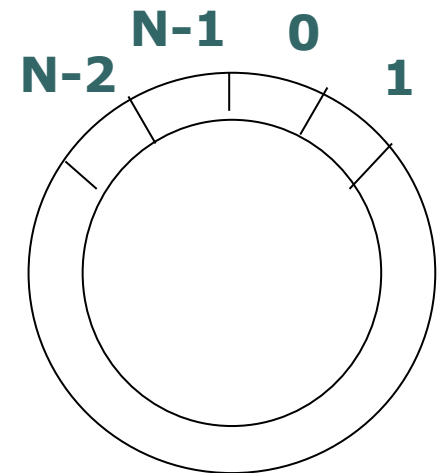
求解约瑟夫 (Josephus) 问题

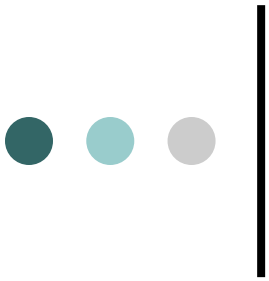
- 约瑟夫问题：有 N 个小孩围坐成一圈，从某个小孩开始顺时针报数，报到 M 的小孩从圈子离开，然后，从下一个小孩开始重新报数，每报到 M ，相应的小孩从圈子离开，问哪一个小孩最后离开圈子的？



//数组bool in_circle[N]; in_circle[i]为true表示编号为i的小孩在圈里
//变量num_of_children_remained表示圈中剩下的小孩数目
//变量count来对成功的报数进行计数
//变量index表示要报数小孩的下标

```
#include <iostream>
using namespace std;
const int N=20, M=5;
int main()
{  bool in_circle[N];
   int num_of_children_remained, index;
   //初始化数组in_circle。
   for (index=0; index<N; index++)
       in_circle[index] = true;
```





//开始报数

```
index = N-1; //保证从编号为0的小孩开始报数,  
num_of_children_remained = N; //报数前的圈子中小孩个数  
while (num_of_children_remained > 1)  
{ int count = 0;  
  while (count < M) //对成功的报数进行计数, 直到M。  
  { index = (index+1)%N; //计算要报数的小孩的编号。  
    if (in_circle[index]) count++; //如果编号为index的  
                                //小孩在圈子中, 该报数为成功的报数。  
  }  
  in_circle[index] = false; //小孩离开圈子。  
  num_of_children_remained--; //圈中小孩数减1。  
}
```



```
//找最后一个小孩
```

```
for (index=0; index<N; index++)  
    if (in_circle[index]) break;
```

```
cout << "The winner is No." << index << ".\n";  
return 0;
```

```
}
```



对 n 个数进行排序P142

- 选择排序
- 冒泡排序
- 快速排序



结构(struct)类型

- 表示由固定多个类型可以不同的元素所构成的复合数据，它是一种用户自定义类型。
- 类型定义格式：
 - `struct <结构类型名> {<成员表>;`
 - <成员表>为成员类型说明，可以是任意的C++类型（`void`和本结构除外）。
 - 结构成员之间在逻辑上没有先后次序关系
 - 定义次序会影响成员的存储安排



- 例如：

```
struct Student
{ int no;
  char name[20];
  Sex sex;
  Date birth_date;
  char birth_place[40];
  Major major;
};
```

结构类型变量的定义

○ 变量定义格式:

- <结构类型名> <结构类型变量名>;
- struct <结构类型名> <结构类型变量名>;
- struct <结构类型名> {<成员表>} <结构类型变量名>;
- struct {<成员表>} <结构类型变量名>;
- 例如: struct
 { int x;
 double y;
 } a,b;



结构类型的操作

○ 访问结构成员

- **<结构类型变量>.<结构成员名>**，可看作是**独立变量**
- 例如： `Student st;`
`st.name, st.no,`

○ 赋值

- **可以进行整体结构赋值**
- 不同的结构类型之间不能相互赋值



结构类型的初始化

○ 注意：

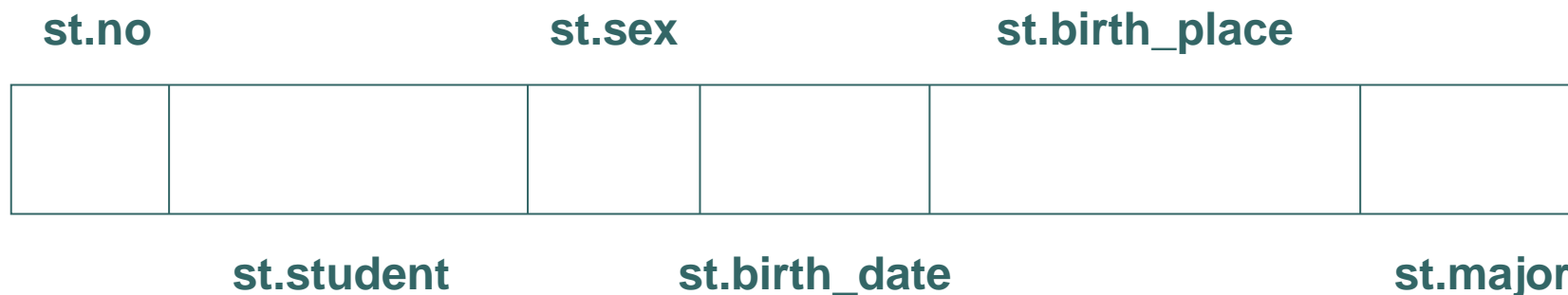
- 定义结构类型时，不能初始化
- 定义结构类型的变量时，可以进行初始化
- 例如

```
Student some_student = {2, "李四", FEMALE,  
{1970,12,20}, "北京", MATHEMATICS}
```

结构类型的存储

- 结构类型的变量在内存中占用一块连续的存储空间，其各个元素依它们在结构类型中的定义依次存储在这块内存空间中。

- 例如：





向函数传递结构数据

- 默认传递方式为**值传递**。有时效率不高，考虑指针或者引用来实现。
- 可作为函数的返回值类型

联合(union)类型

- 用于实现用一种类型表示多种类型的数据。
 - 例如：

```
union A
{
    int i;
    char c;
    double d;
};
```
- 所有成员占有同一块内存空间，该内存空间的大小为其最大成员所需内存空间的大小。
 - 例如：

```
A a;
cout << sizeof(a); //输出8
```


联合类型的赋值 (1)

- 在程序运行的**不同时刻**中，可以给一个联合类型的变量赋予不同类型的数据。
 - 例如： `A a;`
`a.i = 12;` //以下把a当作int型来用
`a.c = 'X';` //以下把a当作char型来用
`a.d = 12.0;` //以下把a当作double型来用
- 对于联合类型的变量，程序将会分阶段地把它作为不同的类型来使用，而**不会同时把它作为几种类型来用**。

联合类型的赋值 (2)

- 当给一个联合类型的变量赋了一个某种类型的值之后，如果以另外一种类型来使用这个值，将得不到原来的值。
 - 例如：

```
a.i = 12;
```

```
cout << a.d; //输出什么呢?
```
- 可以对联合类型的数据进行整体赋值，以及把联合类型的数据传给函数和作为函数的返回值
 - 按照其占有的整个空间进行赋值，而不是按照某个成员进行赋值。



指针类型

- 指针是内存地址的抽象表示，一个指针代表一个内存地址。
- 形式上是一个无符号整数，但是概念上不同于无符号整数
 - 指针对应某个内存单元，它关联到某个变量或者函数
 - 无符号整数的有些运算对指针没有意义
 - 一个内存地址可能属于不同指针类型，这要取决于内存单元存储的是何种类型的程序实体。

指针变量的定义

- 指针变量的定义格式:

- `typedef <类型> *<指针类型名>;`
`<指针类型名> <指针变量名>;`

例如: `typedef int *Pointer;`

`Pointer p;`

- `<类型> *<指针变量名>;`

例如: `int *p;`

- 指针变量拥有自己的内存空间, 存储另一个数据的内存地址

指针变量 指向的数据





指针类型的基本操作

- 取地址操作
- 间接访问操作
- 赋值操作
- 指针运算
- 指针的输出



取地址操作符

- 通过取地址操作符&来获得变量地址

例如：int x

... &x ... //取变量x的地址

间接访问操作

- 通过间接访问操作符*来访问变量

例如：int x; int *p;

p = &x;

*p = 1; //等价于x = 1;

- 访问结构体变量

- (*<指针变量>).<结构成员> 或者 <指针变量>-><结构成员>

例如：struct A

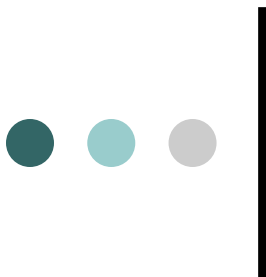
```
{ int i; double d;      char ch; };
```

```
A a;
```

```
A *p=&a;
```

```
cout << (*p).i << endl;
```

```
cout << p->d << endl; //输出a.i和a.d
```



```
int *p;  
int x;  
x = 1;  
p = &x;  
*p = 2;
```

执行操作: "x = 1;"前, (假设120和124分别代表变量x和p的内存地址)

	x		p
120:	?	124:	?

执行操作: "x = 1;"后:

	x		p
120:	1	124:	?

执行操作: "p = &x;" 后:

	x		p
120:	1	124:	120

执行操作: "*p = 2;"后,

	x		p
120:	2	124:	120

取地址操作符

- 只能把定义指针变量时所指定类型的变量的地址赋给它

例如 `int x,*p,*p1;`

`double y,*q;`

`p = &x; //OK, p指向x。`

`q = &y; //OK, q指向y。`

`p = &y; //Error, 类型不一致。`

`q = &x; //Error, 类型不一致。`

`p1 = p; //OK, p1指向p所指向的变量。`

`p1 = q; //Error, 类型不一致。`

`p = 0; //OK, 使得p不指向任何变量。`

`p = 120; //Error, 120为int型。`

`p = (int *)120; //OK, 不建议使用。`

指针的运算 (1)

- 加上或者减去一个整数
 - 运算结果为同类型指针
 - 实际加（或减）值由指针所指数据类型决定

例如： `int x;`

`int *p;`

`p = &x + 2; //x的地址加上sizeof(int)*2`

- 常用于访问数组元素

例如： `int a[10]; int *p; p = &a[0]; //或 p = a;`

访问元素： `a[0]、a[1]、...、a[9]`

`*p、*(p+1)、...、*(p+9)`

`p[0]、p[1]、...、p[9]`



指针的运算 (2)

- 两个同类型指针相减
 - 结果为整数型，其大小由指针所指类型来定
 - 计算两个指针之间有多少个元素

例如： `int a[10];`

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

`int *q = &a[3];`

`cout << q-p << endl; //输出3`



指针的运算 (3)

○ 两个同类型的指针比较

- 比较它们所对应的内存地址的大小。

例如：

```
int a[10],sum,*p,*q;
```

.....

```
while (p <= q)
```

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

```
  p++;
```

```
}
```

指针的输出

通过 `cout<<` 来实现

- 当输出字符指针 (`char*`) 时, 输出指向的字符串
- 进行强制转换(`void *`), 输出指针值

例如: `int x=1;`

```
int *p=&x;
```

```
cout << p; //输出p的值(x的地址, 16进制)
```

```
cout << *p; //输出x的值
```

```
char str[]="ABCD";
```

```
char *q=&str[0];
```

```
cout << q; //输出q指向的字符串ABCD
```

```
cout << *q; //输出q指向的字符A
```

```
cout << (void *)q //输出字符串"ABCD"的内存首地址
```



指针作为形参类型 (1)

```
void swap(int x, int y)
{   int t=x;
    x = y;
    y = t;
}
```

```
int main()
{   int a=0,b=1;
    swap(a,b);
    cout << "a=" << a << ",b=" << b << endl; //输出: a=0,b=1
    return 0;
}
```



指针作为形参类型 (2)

```
void swap(int *px, int *py)
{  int t=*px;
   *px = *py;
   *py = t;
}
```

```
int main()
{  int a=0,b=1;
   swap(&a,&b);
   cout << "a=" << a << ",b=" << b << endl; //输出: a=1,b=0
   return 0;
}
```

再论数组作为参数传递

- 在C++中，数组参数的默认传递方式是把实参数组的首地址传给函数。

下面两种方式函数定义等价：

```
int max(int x[], int num)
{ .....
  ... x[i] ...
  .....
}
```

```
int max(int *x, int num)
{ .....
  ... *(x+i) ... 或者 x[i]
  .....
}
```




向函数传递大型的结构类型数据

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

```
void f(A *p)
{ .....
  ... p->no .... //或者, (*p).no
  ... p->name ... //或者, (*p).name
  .....
}
```

```
int main()
{ A a;
  f(&a); //把结构变量的地址传给函数f。 }
```

指向常量的指针 (1)

- 指针作为形参
 - 提高参数传递效率
 - 通过形参改变实参的值
- 只需要提高效率，把形参定义为指向常量的指针。

例如：void f(const int *p,int num)//或 void f(const int p[],int num)

```
{ .....  
    p[i] = 1; //Error, 不能改变p所指向的数据。  
    ..... }
```

void g(const A *p) //A为一个结构类型

```
{ .....  
    p->no = ... //Error, 不能改变p所指向的数据。  
}
```

指向常量的指针 (2)

- `const int *p;` // 指针p指向常量
 `int y;`
 `p = &y;` // OK 允许指向变量
 `*p = 1;` // Error 不允许通过指针p改变变量y
 `y = 1;` // OK 变量y本身可以改变
- `const int x = 0;` // x是一个常量
 `int *p;`
 `p = &x;` // Error



指向常量的指针 (3)

- `int x, y;`

- `int *const p = &x; // 指针p是常量`

- `*p = 1; // OK, *p是一个变量`

- `p = &y; // Error, p是一个常量, 不能改变`

- `const int x = 0, y=1;`

- `const int * const p = &x; // p是指向常量的指针常量`

- `*p = 1; // Error`

- `p = &y; // Error`



指针作为函数返回值类型 (1)

- 函数的返回值类型可以是指针类型。

例如: `int *max(const int x[], int num)`

```
{    int max_index=0;
    for (int i=1; i<num; i++)
        if (x[i] > x[max_index])
            max_index = i;
    return (int *)&x[max_index];
```

```
}
```

指针作为函数返回值类型 (1)

- 函数的返回值类型可以是指针类型。

例如: `int *max(const int x[], int num)`

```
{    int max_index=0;
    for (int i=1; i<num; i++)
        if (x[i] > x[max_index])
            max_index = i;
    return (int *)&x[max_index];
    // &x[max_index] 相当于 const int * p;
    // int * p = &x[max_index]; error!
}
```

指针作为函数返回值类型 (2)

- 不能把局部变量的地址作为指针返回给调用者。

例如

```
int *f()          int *g()
{ int i=0;        { int j=1;
  return &i;      return &j;
}
```

```
int main()
{ int *p=f();
  int *q=g();
  int x=*p+*q;
  cout << x << endl; //输出什么?
}
```



指针与动态变量

- 在程序运行中，由程序根据需从堆区申请内存所创建的变量。
 - 动态变量的创建
 - 动态变量的撤销

动态变量的创建 (1)

○ 三种方式

- new <类型名>

例如: `int *p; p = new int;`

- new <类型名> [<整形表达式>]

例如: `int *p; int n; p = new int[n];`

`int (*q)[20]; int n; q = new int[n][20];`

- (类型 *) malloc(unsigned int size)

例如: `int *p; double *q; int n;`

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

`q = (double *)malloc(sizeof(double)*n);`

动态变量的创建 (2)

○ new与malloc区别

- new自动计算分配空间大小, malloc需要指定
- new自动返回相应类型指针, malloc需要强制转换
- new会去调用对象类的构造函数, malloc则不会

○ 如果没有足够内存供分配, 则产生bad_alloc异常, 或者返回null指针

○ 动态变量没有名字, 对动态变量的访问需要通过指向动态变量的指针变量来进行。

例如 `int *p,*q;`

`p = new int; ...*p... //访问动态变量`

`q = new int[10]; ...*(q+3)或q[3]...//访问动态数组元素`

动态变量的撤销 (1)

- 需要由程序显式地撤销，三种方式

- delete <指针变量>

例如: `int *p = new int; ...`
`delete p;`

- delete [] <指针变量>

例如: `int *p = new int[20];`
`delete []p;`

- void free(void *p)

例如: `int *p = (int *)malloc(sizeof(int));`
`int *q = (int *)malloc(sizeof(int) * 20);`
`.....`
`free(p); free(q);`

动态变量的撤销 (2)

- delete与free的区别

- 如果p指向的是对象（或对象数组），则delete p（或delete []p）**会去调用对象类的析构函数**，而free(p)则否。

- 用delete和free**只能撤消动态变量！**

例如：int x,*p;

p = &x;

delete p; //Error

“内存泄漏” 问题

- 没有撤消动态对象，而把指向它的指针变量指向了别处或指向它的指针变量的生存期结束了，导致这个动态变量存在但不可访问。

例如： **int x,*p;**

p = new int[10];

p = &x; //动态数组就不能再访问!



“悬浮指针” 问题

- 用delete或free撤消动态变量后，C++编译程序一般不会对指向它的指针变量的值赋为0，这时就会出现一个“悬浮指针”（dangling pointer），指向一个无效空间。

例如：

```
int *p;  
p = new int[10];  
.....  
delete []p;  
.....  
*p = 1;
```

动态数组

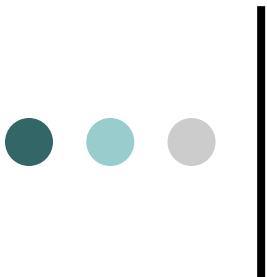
- 对于程序运行前无法确定元素个数的数组问题，一种解决办法是采用**动态数组**，即在程序运行过程中创建数组

```
int n;  
cin >> n;  
p = new int[n];  
for(int i=0; i<n; i++) cin >> p[i];  
sort(p, n);  
delete []p;
```

- 如果输入时还是不能确定这些数的个数呢？比如最后一个输入以-1作为结束标记。

- 对输入的若干个数进行排序，在输入时，先输入各个数，最后输入一个结束标记（如：-1），这时，可以按以下方式实现：

```
const int INCREMENT=10;
int max_len=20, count=0, n, *p=new int[max_len];
cin >> n;
while (n != -1)
{ if (count >= max_len)
    { max_len += INCREMENT;
      int *q=new int[max_len];
      for (int i=0; i<count; i++) q[i] = p[i];
      delete []p;
      p = q;
    }
    p[count] = n;
    count++;
    cin >> n;
}
sort(p,count);
.....
delete []p;
```

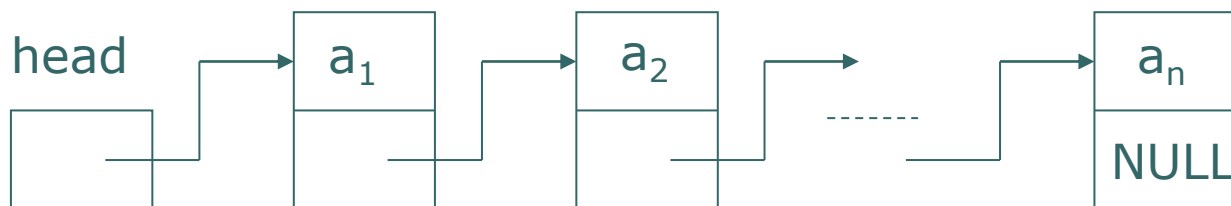
- 上面的实现方法虽然可行，但是
 - 当数组空间不够时，它需要重新申请空间、进行数据转移以及释放原有的空间，这样做比较麻烦并且效率有时不高。
 - 当需要在数组中增加或删除元素时，还将会面临数组元素的大量移动问题。
- 链表可以避免数组的上述问题。



动态变量的应用--链表

- 链表用于表示由若干个（**个数不定**）**同类型**的元素所构成的具有**线性结构**的复合数据。
- 链表中的每一个元素除了本身的数据外，**还包含一个（或多个）指针**，它（们）指向链表中下一个（和其它）元素。
- 如果每个元素只包含一个指针，则称为单链表，否则称为多链表。
- 上述的定义隐含着链表元素**在内存中不必存放在连续的空间内**。

单链表



○ 结点类型

```
struct Node
```

```
{ int content; //代表结点的数据
```

```
    Node *next; //代表后一个结点的地址
```

```
};
```

○ 表头指针变量

```
Node *head=NULL; // 头指针变量定义，初始状态下为空值。  
                // NULL在cstdio中定义
```

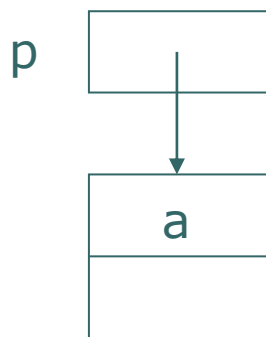
在链表中插入结点 (1)

- 首先产生新结点

`Node *p=new Node;` //产生一个动态变量来表示新结点。

`p->content = a;` //把a赋给新结点中表示结点值的成员。

- 图示为：



在链表中插入结点 (2)

- 如果链表为空（创建第一个结点时）：

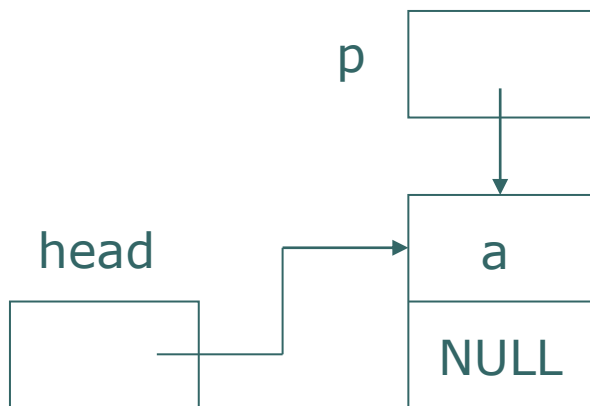
```
if (head == NULL)
```

```
{ head = p;           //头指针指向新结点。
```

```
  p->next = NULL; //把新结点的next成员置为NULL。
```

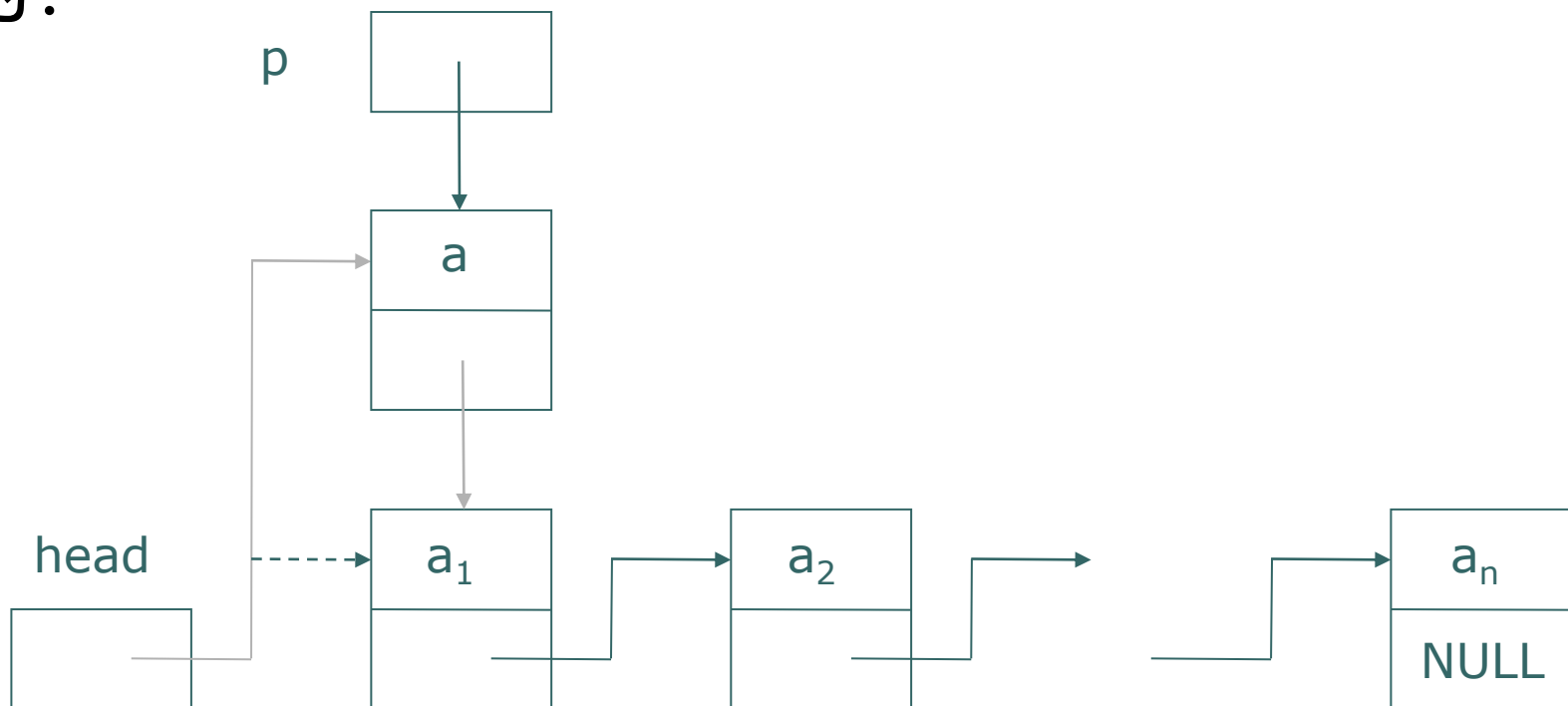
```
}
```

- 图示为：



在链表中插入结点 (3)

- 如果新结点插在表头，则进行下面的操作：
 $p \rightarrow \text{next} = \text{head}$; //新结点的下一结点为链表原来的第一个结点。
 $\text{head} = p$; //表头指针指向新结点。
- 图示为：



在链表中插入结点 (4)

- 如果新结点插在表尾，则进行下面的操作：

Node *q=head; //q指向第一个结点

while (q->next != NULL) //循环查找最后一个结点

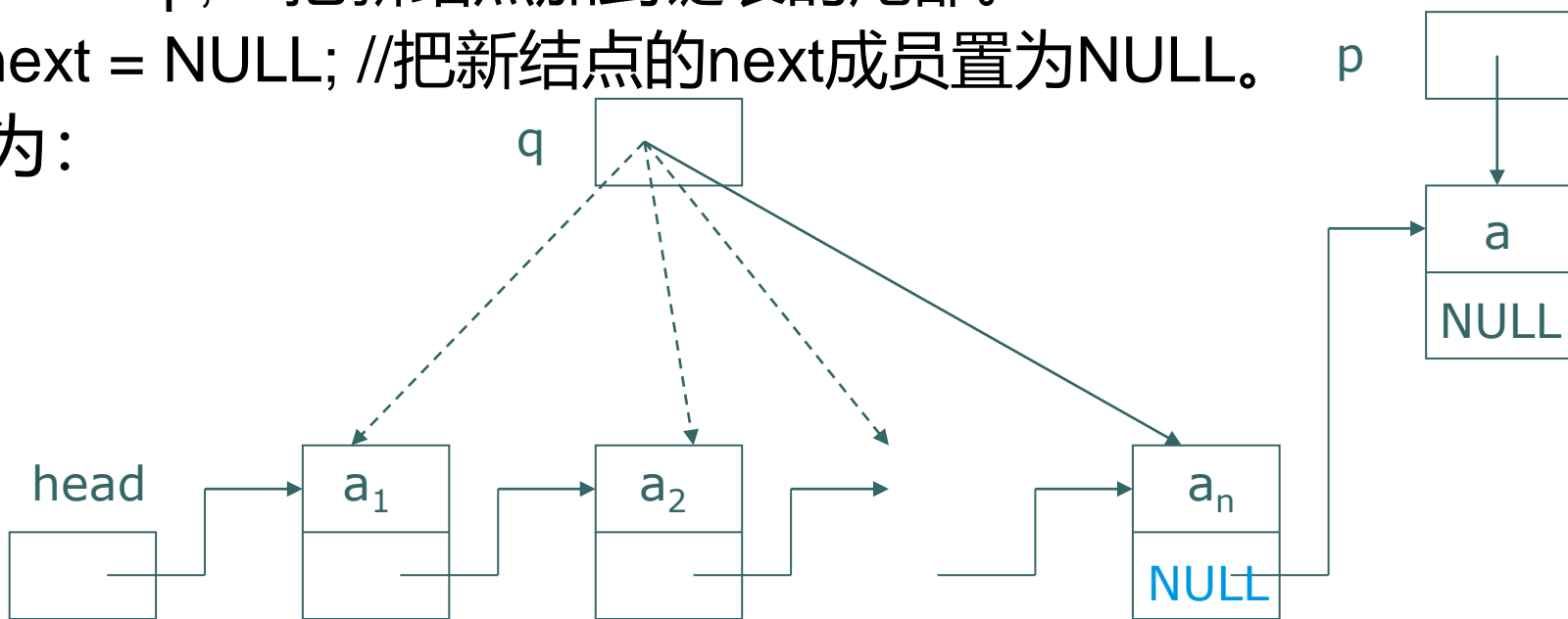
q = q->next;

//循环结束后，q指向链表最后一个结点

q->next = p; //把新结点加到链表的尾部。

p->next = NULL; //把新结点的next成员置为NULL。

- 图示为：



在链表中插入结点 (5)

- 如果新结点插在链表中第 i ($i > 0$) 个结点 (a_i) 的后面:

//查找第 i 个结点

Node *q=head; //q指向第一个结点。

int j=1;

//循环查找第 i 个结点。

while (j < i && q->next != NULL)

{ q = q->next;

j++;

}

在链表中插入结点 (6)

//循环结束时, q或者指向第i个结点, 或者指向最后一个结点
(结点数不够i时)。

if (j == i) //q指向第i个结点。

{ p->next = q->next; //把新结点的下一个结点指定为q
//所指向结点的下一个结点。

q->next = p; //把新结点指定为q所指向结点的下一个结点。

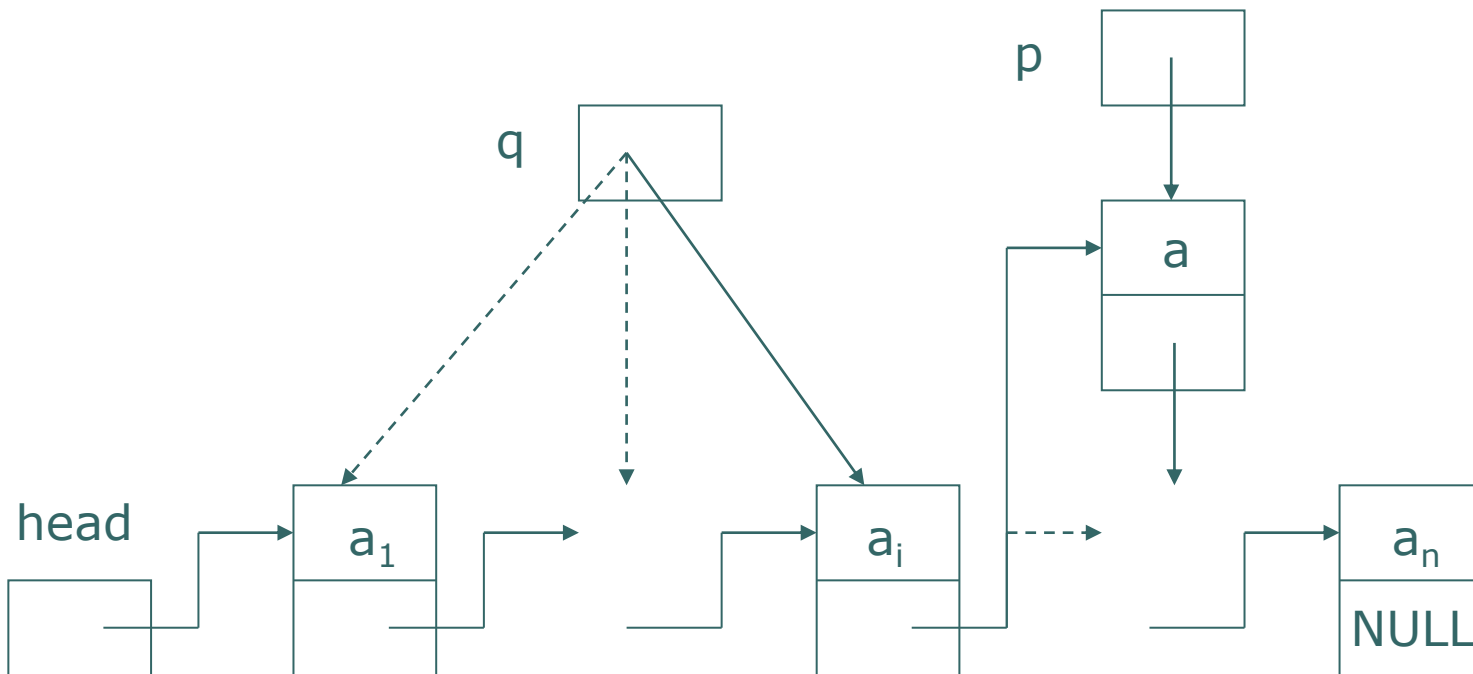
}

else //链表中没有第i个结点。

cout << "没有第" << i << "个结点\n";

在链表中插入结点 (7)

○ 图示为：



在链表中删除一个结点 (1)

- 假设链表不为空，即 **head != NULL**

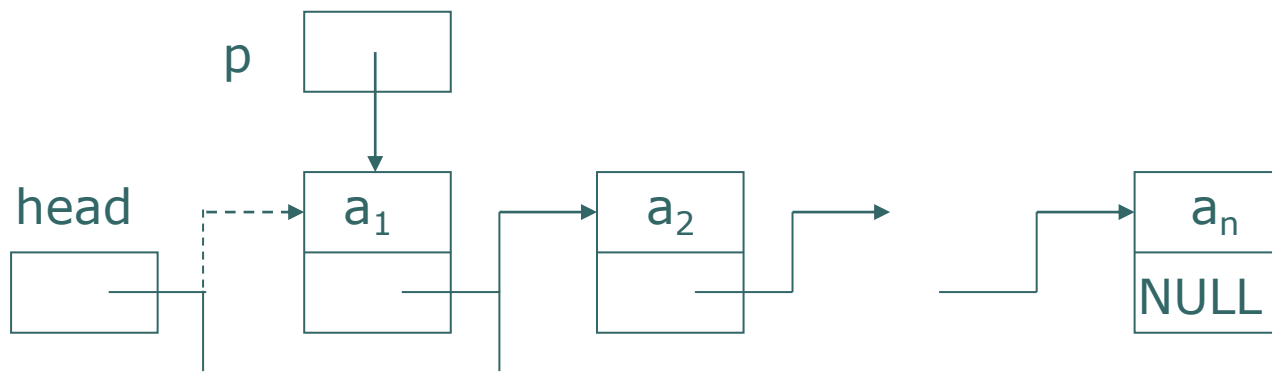
- 如果删除链表中第一个结点：

Node *p=head; //p指向第一结点。

head = head->next; //头指针指向第一结点的下一结点。

delete p; //归还删除结点的空间。

- 图示为：





在链表中删除一个结点 (2)

- 如果删除链表的最后一个结点:

```
Node *q1=NULL,*q2=head;// 循环查找最后一个结点,  
                        // q2指向它, q1指向它的前一个结点。
```

```
while (q2->next != NULL)
```

```
{ q1 = q2;
```

```
  q2 = q2->next;
```

```
}
```

```
if (q1 != NULL) //存在倒数第二个结点。
```

```
    q1->next = NULL; //倒数第二个结点的next置为NULL。
```

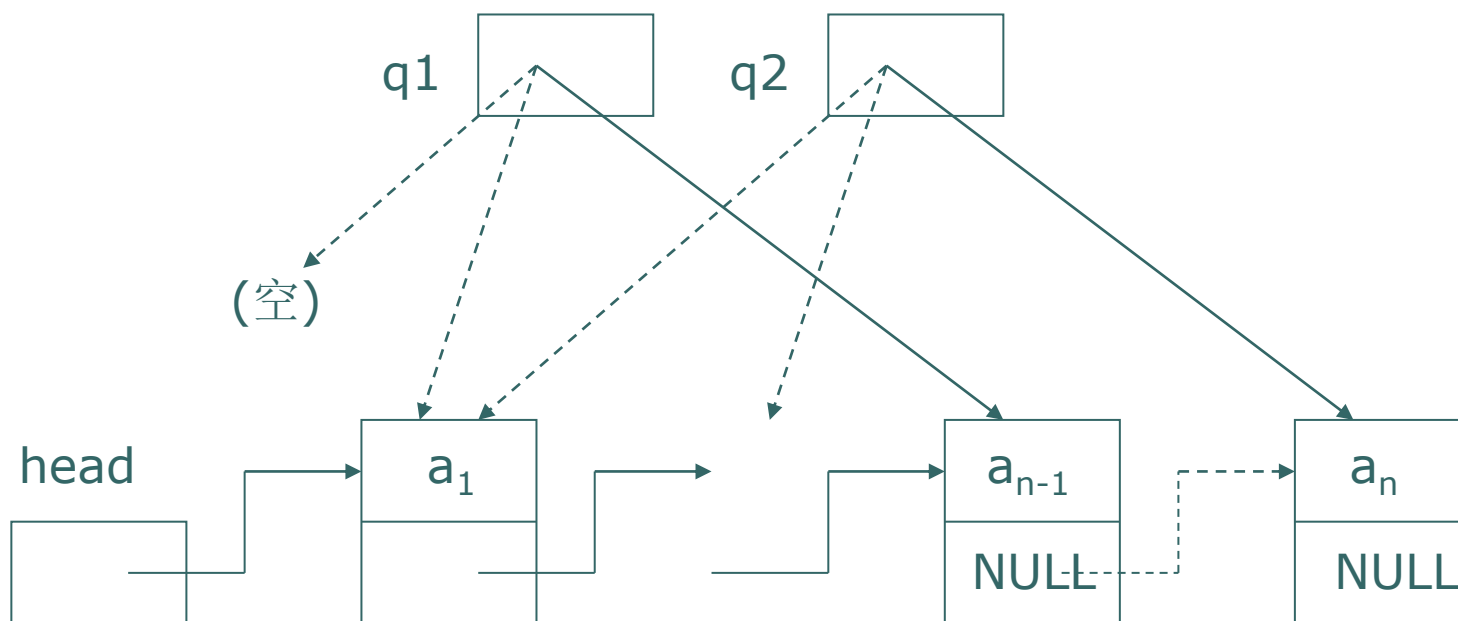
```
else //链表中只有一个结点。
```

```
    head = NULL; //把头指针置为NULL。
```

```
delete q2; //归还删除结点的空间。
```

在链表中删除一个结点 (3)

○ 图示为：



在链表中删除一个结点 (4)

- 如果删除链表中第 i ($i > 0$) 个结点 a_i :

```
if (i == 1) //要删除的结点是链表的第一个结点。
{
    Node *p=head; //p指向首结点。
    head = head->next; //head指向首结点的下一个结点。
    delete p; //归还删除结点的空间。
}
```

```
else //要删除的结点不是链表的第一个结点。
{
    Node *p=head; //p指向第一个结点。
    int j=1;
    //循环查找第i-1个结点。
    while (j < i-1 && p->next != NULL)
    {
        p = p->next;
        j++;
    }
    if (p->next != NULL) //链表中存在第i个结点。
    {
        Node *q=p->next; //q指向第i个结点。
        p->next = q->next; //把第i-1个结点的下一个结点
                           //设为第i个结点的下一个结点。
        delete q; //归还第i个结点的空间。
    }
    else //链表中没有第i个结点。
        cout << "没有第" << i << "个结点\n";
}
```



在链表中检索某个值a

```
int index=0;//用于记住结点的序号
```

```
//从第一个结点开始遍历链表的每个结点查找值为a的结点。
```

```
for (Node *p=head; p!=NULL; p=p->next)
{  index++; //记住结点的序号，下面输出时需要。
    if (p->content == a)
        break;
}
```

```
if (p != NULL) //找到了
```

```
    cout << "第" << index << "个结点的值为：" << a << endl;
```

```
else //未找到
```

```
    cout << "没有找到值为" << a << "的结点\n";
```


对输入的若干个数进行排序，在输入时，先输入各个数，最后输入一个结束标记（如：-1）

```
struct Node
```

```
{ int content; //代表结点的数据  
  Node *next; //代表后一个结点的地址  
};
```

```
extern Node *input(); //输入数据，建立链表，返回链表的  
  头指针
```

```
extern void sort(Node *h); //排序
```

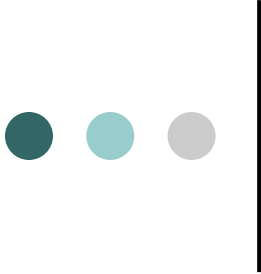
```
extern void output(Node *h); //输出数据
```

```
extern void remove(Node *h); //删除链表
```

```
int main()
```

```
{ Node *head;  
  head = input();  
  sort(head);  
  output(head);  
  remove(head);  
  return 0;
```

```
}
```



```
#include <iostream>
#include <cstdio>
Node *input()
{ Node *head=NULL,*tail;
  int x;
  std::cin >> x;
  while (x != -1)
  { Node *p=new Node;
    p->content = x;
    p->next = NULL;
    if (head == NULL)
      head = p;
    else
      tail->next = p;
    tail = p;
    std::cin >> x;
  }
  return head;
}
```



```
void sort(Node *h)
```

```
{ if (h == NULL) return;
```

```
  for (Node *p1=h; p1->next != NULL; p1 = p1->next)
```

```
  { Node *p_min=p1;
```

```
    for (Node *p2=p1->next; p2 != NULL; p2=p2->next)
```

```
      if (p2->content < p_min->content) p_min = p2;
```

```
    if (p_min != p1)
```

```
    { int temp = p1->content;
```

```
      p1->content = p_min->content;
```

```
      p_min->content = temp;
```

```
    }
```

```
  }
```

```
}
```

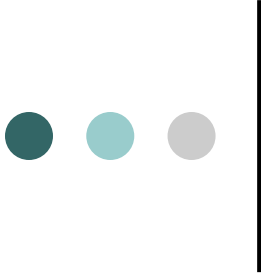
```
void output(Node *h)
```

```
{ for (Node *p=h; p!=NULL; p=p->next)
```

```
  std::cout << p->content << ',';
```

```
  std::cout << endl;
```

```
}
```



```
void remove(Node *h)
{ while (h != NULL)
  { Node *p=h;
    h = h->next;
    delete p;
  }
}
```



指针与数组 (1)

○ 一维数组的首地址

```
int a[10];
```

- 通过数组首元素来获得

例如: `int *p;`

`p = &a[0];`或`p = a;` //一维数组名表示第一个元素的地址

`p++;` //加: `sizeof(int)`

- 通过整个数组获得。

例如: `int (*q)[10];` //等价于: `typedef int A[10]; A *q;`

`q = &a;` //整个数组的地址

`q++;` //加: `10×sizeof(int)`

指针与数组 (2)

二维数组的首地址

```
int b[5][10];
```

- 通过第一行、第一列元素来获得。

例如: `int *p;`

`p = &b[0][0];` 或 `p = b[0];` `p++;` //加sizeof(int)

- 通过第一行的一维数组获得。

例如: `int (*q)[10];`

`q = &b[0];` 或 `q = b;` //二维数组名表示第一行的地址。

`q++;` //加10×sizeof(int)

- 通过整个数组获得。

例如: `int (*r)[5][10];`

`r = &b;` `r++;` //加5×10×sizeof(int)



函数main的参数

- 定义格式为：

- `int main(int argc, char *argv[]);`

- argc表示传给函数main的参数的个数，

- argv是一个一维数组，其每个元素为一个指向字符串的指针。

- 例如：“copy file1 file2”执行程序copy时，将得到参数：

- argc: 3

- argv[0]: "copy"

- argv[1]: "file1"

- argv[2]: "file2"



函数指针

- C++中可以定义一个指针变量，使其指向函数。

例如： `double (*fp)(int);` //fp是一个指向函数的指针变量

- 用取地址操作符&来获得函数的内存地址，或直接用函数名来表示。

例如： `double f(int x) { ... }`

`fp = &f;` //或, `fp = f;`

- 通过函数指针调用函数可采用下面的形式：

例如： `(*fp)(10);` 或 `fp(10);`



例：编写一个程序，根据输入的要求
执行在一个函数表中定义的某个函数。

```
#include <iostream>
#include <cmath>
using namespace std;
const int MAX_LEN=8;
typedef double (*FP)(double);
FP func_list[MAX_LEN]={sin,cos,tan,asin,acos,atan,log,log10};
int main()
{   int index;
    double x;
    do //循环以获得正确的输入
    {   cout << "请输入要计算的函数(0:sin 1:cos 2:tan 3:asin\n"
        << "4:acos 5: atan 6:log 7:log10):";
        cin >> index;
    } while (index < 0 || index > 7);
    cout << "请输入参数: ";
    cin >> x;
    cout << "结果为: " << (*func_list[index])(x) << endl;
    return 0;
}
```

向函数传递函数

- 调用一个函数时可把一个函数作为参数传给被调用函数。被调用函数的形参定义为一个函数指针类型，调用时的实参为一个函数的地址。

例如：

```
int f(int);    int g(int);
int func(int (*fp)(int x)) //参数为一个函数指针类型
{  int i;
  (*fp)(i)... //或fp(i); 调用形参fp所指向的函数
}
int main()
{  func(&f)... //或func(f); 调用函数func, 把f作为参数传给它
  func(&g)... //或func(g); 调用函数func, 把g作为参数传给它
}
```

C++的引用类型 (1)

- 引用类型用于给一个变量取一个别名。

例如：int x=0;

```
int &y=x; //y为引用类型的变量
```

```
cout << x << ',' << y << endl; //结果为： 0,0
```

```
y = 1;
```

```
cout << x << ',' << y << endl; //结果为： 1,1
```

- 对引用类型变量的访问实际访问的是被引用的变量，效果与通过指针间接访问另一个变量相同。

C++的引用类型 (2)

○ 需要注意下面几点:

- 定义引用类型变量时, 应在变量名**加上符号 “&”**, 以区别于普通变量。

例如: `int &y=x;`

- 定义引用变量时**必须要有初始化**, 并且引用变量和被引用变量应具有相同的类型。

例如: `int x; int &y=x;`

- 引用类型的变量定义之后, 它**不能再引用其它变量**。

例如: `int x1,x2;`

`int &y=x1;`

.....

`y = &x2; //Error`

引用类型作为函数的参数类型

- 引用类型作为参数类型，实现指针类型参数的效果。

例如： `#include <iostream>`

```
using namespace std;
```

```
void swap(int &x, int &y)
```

```
{ int t;
```

```
  t = x;  x = y;  y = t;
```

```
}
```

```
int main()
```

```
{ int a=0,b=1;
```

```
  cout << a << ',' << b << endl; //结果为： 0,1
```

```
  swap(a,b);
```

```
  cout << a << ',' << b << endl; //结果为： 1,0
```

```
  return 0;
```

```
}
```



常量的引用

- 通过把形参定义成对常量的引用，可以防止在函数中通过形参改变实参的值。

例如：struct A

```
{ int i;  
  ..... };
```

```
void f(const A &x)  
{ x.i = 1; //Error  
  ..... }
```

```
int main()  
{ A a;  
  f(a);  
}
```



引用类型与指针类型的区别

- 语法上，引用是采用**直接访问**形式，而指针则需要采用**间接访问**形式。
- 在作为函数参数类型时，形式上，引用类型参数的**实参是变量的名字**，而指针类型参数的实参是**变量的地址**。
- 除了在定义时指定的被引用变量外，引用类型变量**不能再引用其它变量**；而指针变量定义后可以**指向其它的变量**。

```
void f(int *const p)
{ .....
  int m;
  p = &m; //Error,
  ... *p ... //通过p只能访问实参
}
```

```
void f(int *p)
{ .....
  int m;
  p = &m; //OK,
  ... *p ... //通过p可以访问实参以外的数据
}
```

```
void g(int &x)
{ int m;
  .....
  x = &m; //Error
  x = m; //把实参的值改成m, 即通过x只能访问实参
}
```

```
int main() { int a; f(&a); g(a); }
```