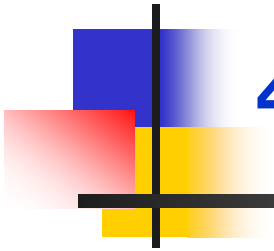
第四章 函数、内联函数、函数重载与默认参数

面向对象程序设计（C++）

第四章 函数和作用域



4.1 函数的定义和声明

- 函数：本质上是一段参数化的共享代码。

4.1.1 函数的定义

- <类型><函数名>(<参数表>) {〈若干条语句〉}

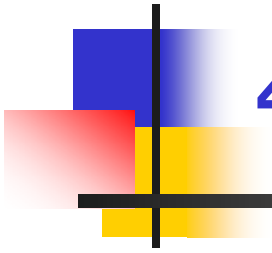
```
#include <iostream.h>

    double sum(double x, double y)
    {return x+y;}

    void main ( )
    {  double a,b;
      cout << “Input double a and b:”
       cin >>a>>b;
       double sum = sum(a,b);
       cout <<“sum=”<<sum<<endl;
    }
```

4.1.2 函数的声明

- “一遍扫描”的需要。原则：
  定义在先，调用在后，调用前可以不必声明；
  定义在后，调用在前，调用前必须声明。

- <类型> <函数名>(<参数表>);

```
int add(int,int);
void main()
{ int i=10,j=20;
     …;
   j=add(i ,j);
                …;
}
int add(int a,int b){   return(a+b);      }
```



4.2 函数的调用与参数传递

函数调用语句中的函数名经编译后变成函数的入口地址！

4.2.1 函数的调用方法

- 语法：〈函数名〉（〈实参表〉）

```
c=sum(a,b);
swap(&a,&b);
```

- 关键：“虚实结合”的问题！即：

实在参数的值如何传递给形式参数?
对形式参数的值的改变是否影响实在参数?

4.2.1.1 函数的调用和返回序列

- 调用序列:
  1. 分配函数局部空间(活动记录)
  2. 参数传递
  3. 保存返回地址
  4. 控制转移
  5. 执行函数

- 返回序列:
  1. 返回值传递
  2. 控制转移
  3. 返回主调函数

4.2.2 参数传递

实参 ←——→ 形参

- 值传递：把实际参数的值复制给形式参数，也就是说，**形参和实参是两个不同的内存空间**。函数所处理的仅仅是实际参数的一个拷贝。

- 引用传递：把实际参数的地址传递给形式参数，也就是说，**形参和实参实际上是同一个内存空间**。函数通过对形式参数的处理来达到对实际参数进行加工的目的。

4.2.2.1 传值调用-传值

```
#include <iostream.h>
void swap1(int x, int y)
{    int temp;
     temp = x;
     x = y;
     y = temp;
     cout<<“x=”<<x<<“y=”<<y<<endl;
}
void main ( )
{
     int a=5,b=9;
     swap1(a,b);
     cout << “a=”<<a<<“b=”<<b<<endl;
}
```

执行过程

(0) **a:** 5

b: 9

(1)

x: 5

y: 9

(2)

x: 9

y: 5

temp: 5

(3) 退出函数

x, y, temp 均被撤销

(4) **a, b** 未改变

4.2.2.2 传值调用-传指针(地址值)

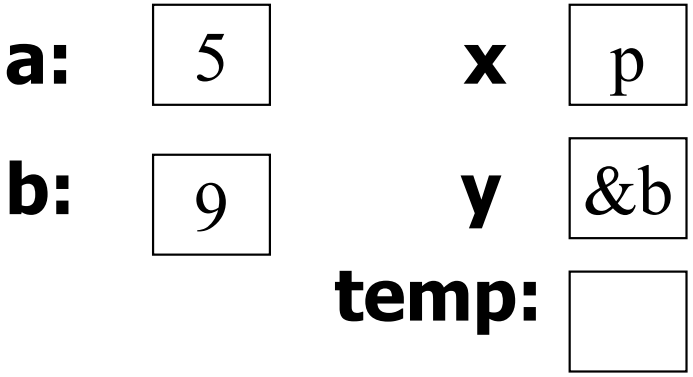
```
void swap2(int *x, int *y)
{ int temp;
   temp = *x;
   *x = *y;
    *y = temp;
    cout <<“x =”<<*x<“,”<<“y=”<<*y;
}

void main(){
    int a=10,b=20;
    swap2(&a,&b);
}
```

执行过程

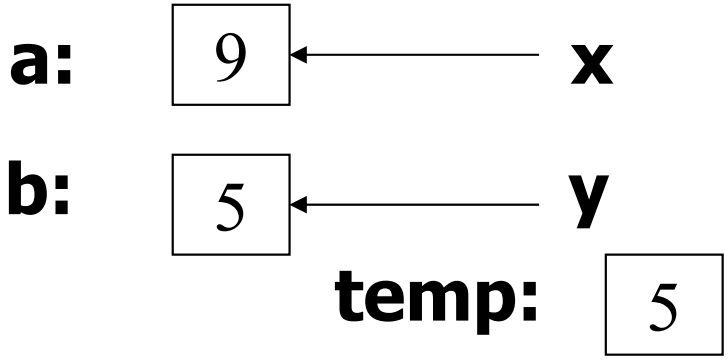
(0) **a:** 5

b: 9

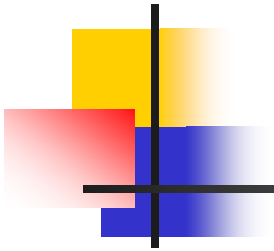
(1)



(2)



(3) **a, b的值被交换**



传值 vs 传指针

- 本质上都是值传递；

- 适用于：
  - 当需要将一个类的大的对象作为参数传递给函数时；
  - 当参数值必须被修改后返回。

思考题1

```
void GetMemory(char *p)
{
    p = new char*(100);
}

int Test(void)
{
    char *str = NULL;
    GetMemory(str);
    strcpy(str, "hello world");
    return 0;
}
```

请问运行Test函数会有什么样的结果?

答：程序崩溃。

p只是str的一个副本。因为GetMemory并不能传递动态内存，Test函数中的 str一直都是 NULL。strcpy(str, “hello world”);将使程序崩溃。

思考题2

```
char *GetMemory(void)
{

    return p;
}
void Test(void)
{
    char *str = NULL;
    str = GetMemory();
    cout<<str;
}
```

请问运行Test函数会有什么样的结果?

答：可能是乱码。

因为GetMemory返回的是指向“栈内存”的指针，该指针的地址不是 NULL，但其原先的内容已经被清除，新内容不可知。

4.2.2.3 引用传递

函数定义：

- **形参声明为引用量；**
- **函数体直接对引用量操作；**

函数调用：

- **形式上象值传递；实际上是引用传递；**
- **形参是实参的别名，并没有另外开辟空间。**
- **对形参的修改直接影响到实参。**

```
void swap3(int &v1,int &v2)
{
    int temp = v2;  v2 =v1; v1 = temp;
}
void main(){
  int a=5,b=9;    swap3(a,b);
}
```

优点：

- 形式参数只是实参的别名，因此函数调用时不需要为它分配空间；尤其是传递大的对象。

- 在函数内对参数值的改变不再作用于局部拷贝，而是直接针对实参；但是相对于指针传递而言，引用传递的函数体定义和函数调用的形式更为简洁。

```
void swap3(int *v1,int *v2){
    int temp = *v2; *v2 =*v1;  *v1 = temp;
}
```

```
void swap3(int &v1,int &v2){
    int temp = v2;    v2 =v1;      v1 = temp;
}
```

思考题

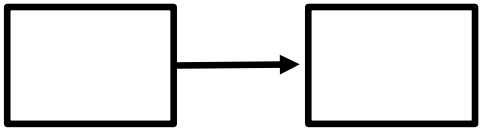
判断对错：

```
int& f()
{
        int i;
        return i;
}


int f()
{
        int i;
        return i;
}
```

思考题

判断对错：

```
int& f()
{
        int i;
        return i;
}


int f()
{
        int i;
        return i;
}
```



4.2.2.5 使用数组作函数参数

三种情形：

1、形参和实参都用数组；
2、形参和实参都用对应数组的指针；
3、实参用数组名，形参用引用

例：

```
int a[8] = {1,3,5,7,9,11,13};

void fun(int b[ ],int n){
   for (int i =0;i<n-1;i++){
        b[7]+=b[i];
   }
}

int main ( ){
      int m =8;
      fun(a,m);
      cout <<a[7]<<endl;
}
```

例：

```
int a[8] = {1,3,5,7,9,11,13};

void fun(int b[],int n){
   for (int i =0;i<n-1;i++){
        b[7]+=b[i];
   }
}

int main ( ){
      int m =8;
      fun(a,m);
      cout <<a[7]<<endl;
}
```

49

例：

```
int a[8] ={1,3,5,7,9,11,13};

void fun( int *pa, int n){
   for (int i= 0;i<n-1;i++){
      *(pa + 7) += *(pa + i);
      pa[7]+=pa[i];
   }
}

int main( ){
    int m =8;
    fun(a,m);
    cout << a[7]<<endl;
}
```

例：

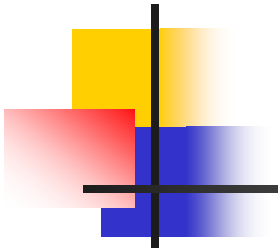
```
int a[8] ={1,3,5,7,9,11,13};

void fun( int *pa, int n){
   for (int i= 0;i<n-1;i++){
      *(pa + 7) += *(pa + i);
   }
}

int main( ){
    int m =8;
    fun(a,m);
    cout << a[7]<<endl;
}
```

49

4.2.3 函数参数的求值顺序

```
#include <iostream>
using namespace std;

int add_int(int x, int y){
      return x+y;
}

int main ( ){
    int x(4),y(6);
    ++x;
    int z = add_int(x,x+y);
    cout <<z<<endl;
}
```

若从右向左：z为15
若从左向右：z为16



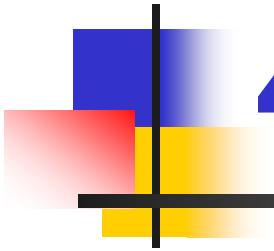
```
#include <iostream>
using namespace std;

int add_int(int x, int y){
      return x+y;
}

int main ( ){
    int x(4),y(6);
    int z = add_int(x++,x+y);
    cout <<z<<endl;
}
```

14

4.2.4 返回语句实现过程(略)

- Return (表达式);

1. 求〈表达式〉的值
2. 如果〈表达式〉的类型与函数类型不相同，将表达式的类型**自动转换**为函数的类型。(注意：这种转换是**强制性的**，可能出现信息丢失的情况)
3. 将求出的表达式值返回给调用函数作为调用函数的值;
4. 将控制权由被调用函数返回主调用函数，执行主调用函数下一条语句。



4.3 作用域

变量的可见范围

4.3.1 作用域

- 作用域：标识符的可见范围
- 作用域的种类：
  - -程序级（全局级　　）
  - -文件级（.cpp　　）
  - -函数级（func(){..}）
  - -块级　（　{…}　　）

4.3.2 作用域的屏蔽规则

```
Class A{
   int a;
};

int main(){

int a=1;

A a1,a2,a3;
a1.a;
a2.a;
a3.a;

}
```

在某个作用范内定义的标识符在该范围内的子范围内可以定义定义新的同名标识符。这时原定义的标识符在子范围内是不可见的，但是它还是存在的，只是在子范围内由于出现了同名的标识符而被暂时隐藏起来。过了子范围后，它又是可见的。

4.3.3 局部变量和全局变量

作用域：
函数级；
块级。

生命期：
自动类变量；
内部静态变量

作用域：
程序级；
文件级。

生命期：
外部变量；
外部静态变量；

全局变量与局部变量

```
#include <iostream.h>
int a(10);    // 全局变量     a=10;
void other( );
void f( )
{
        int a(3);     a=3
        register int b(5);      //寄存器、局部变
        static int c=0;           // 静态局部变量
        cout << “a =” << a << “,” << “b =”
                << b <<”,” << “c = ”<< c;
        a=3, b=5;
        other( );

        other( );
}
```

全局变量与局部变量(续)

```
int c=5;

void other( )
{
        int a(5);      a=5              //局部变量
        a += 10; a=15
        b+=20;  b=52
        cout << “a=” << a << “,”
                << “b=” << b << endl;
}
```

a=3,b=5;

a=15,b=32;

a=15,b=52

全局变量与局部变量(续)

```
void other( )
{
        int a(5);                    //局部变量
        static int b(12);      // 静态局部变量
        a += 10;
        b+=20;
        cout << “a=” << a << “,”
                << “b=” << b << endl;
}
```

```
a=3,b=5,c=0
 a=15,b=32
 a=15,b=52
```

4.3.4 内部静态变量

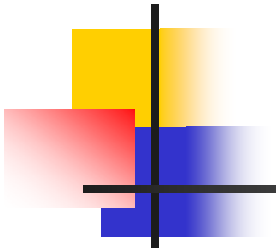
- 初始化一次;
- 作用域是局部于函数的;
- 生命期是全局的（程序级）；

```
#include <iostream.h>
traceGcd(int v1, int v2)
{
    static int depth = 1;
    cout <<“depth#”<<depth++<<“\n”;
    if (v2 = = 0)
            return v1;
    return traceGcd(v2, v1%v2);
}
```

4.3.5 寄存器变量

在函数中频繁使用的局部变量可以定义为寄存器变量(用register修饰)，只要还有寄存器可用，编译器会尽可能将其放入寄存器中，提高执行速度。

```
for (register int i= 0; i < s2; i++){
                    ia[i] = i;
}
```



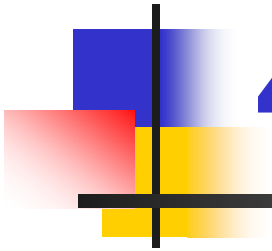
4.3.6

```
namespace lib1{     int   a;     }

namespace lib2{     int   a;     }

lib1::a
```

小结

- 同名的标识符可在不同域中重复出现，分别具有不同含义和用途，而不会引发任何不良效果。

```
void swap(int *ia, int, int){ia=};
void sort(int *ia, int sz){ia=};
void putValue(int *ia, int sz){ ia=};
int ia[ ] = {4,7,0,9,2,5,8,3,6,0};
const sz = 10;
main( )
{    int i, j;
     //…
     swap(ia,i,j);
     sort(ia,sz);
     putValue(ia,sz);
}
```



4.4 内联函数[注]

对宏替换的改进！

4.4.1 内联函数（inline function)

- 内联函数：告诉编译器，在编译时用函数代码的“拷贝替换函数调用。

```
inline int power_int(int x){
    return   x*x;
}

void main( )
{
    for (int i = 1;i<=10; i++) {
        int p=power_int(i);     //编译时换成 (i*i)
        cout <<i<<“*”<<i<<“=”<<p;
    }
}
```

说明：

- 优点：
  - -- **提高代码效率**
  - -- **保持源代码可读性及易维护性；**

- 限制：
  - -- **在内联函数内不允许用循环语句和开关语句；**
  - -- **内联函数的定义必须出现在每一个调用该函数的源文件之中；(由于要进行源代码替换)**
  - -- **编译器不保证所有被定义为inline 的函数编译成内联函数。Eg:递归函数**

类内实现的函数自动内联！

4.4.2 内联函数 vs 宏替换

- 内联函数由编译器处理，而宏由预处理器处理；

  - 编译器会对内联函数调用的实参进行类型检查；

  - 宏可能会有副作用，而内联函数不会有副作用。

```
#define abs(n)    ((n)<0 ? –(n):(n)) ;
i=1;
j= abs(--i);   // 实际上j= ( (--i)<0 ? –(--i) : ( -- i) ) ;

#define power(n)  (n*n)
k=power(1+2);  //   实际上k=(1+2*1+2)
```

4.4.3 内联函数 vs 直接代码比较

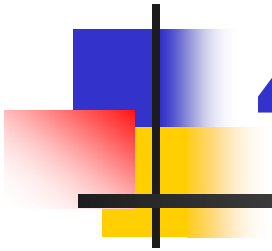
- **可读性强**

  max(a,b)　　*vs*　　(a>b?a:b);

- **复用性好**

  max(a,b)　　vs　　(a>b?a:b);
  max(c,d)　　(c>d?c:d);

- **便于修改，易于维护**



4.5 默认参数[注]

C++函数的高级机制之一

4.5.1 设置函数参数的默认值

```
// DefaultArgument.h
typedef struct time{
    int h;              // hour
    int m;              // minute
    int s;              // second
} *TIME, CLOCK;

int setclock(TIME t, int hour=12,int minute=0,int
second=0){
    t->h = (hour <= 23 && hour >=0)? hour : 12;
    t->m=(minute<=59 &&minute>=0)? minute:0;
    t->s=(second<=59 && second >=0)?second:0;
}
```

```
// DefaultArgument.cpp
#include <iostream>
using namespace std;

void main(){
    CLOCK t1,t2,t3,t4,t5;

     setclock(&t1,12,0,0);       // t1=12:00:00
     setclock(&t2,7);           // t2=07:00:00
     setclock(&t3,14,20);        // t3=14:20:00
}
```

说明

- 若有实际参数值，则缺省值无效；否则，参数值就采用缺省值；

```
setclock(&t1);              // t1=12:00:00
setclock(&t2, 7);           // t2=07:00:00
setclock(&t3,14,20);        // t3=14:20:00
```

- 带有缺省值的参数必须全部放置在参数的最后，即在带有缺省值的参数的右边不再出现无缺省值的参数；

```
int setclock(TIME t,int minute,int second=12,
int hour) //非法
 { … }
```

说明(续)

- 函数的参数既可以在定义又可以在声明中指明，一旦定义了缺省参数，就不能再定义它，但可以添加一个或多个缺省参数。

```
void f (int a, int b, int c =0);    // 定义缺省参数a
void f(int a, int b=1, int c=0);    // 增加缺省参数b
void f(int a, int b=2, int c=1);    // 错误，企图重定义b
                                    // 和c的缺省参数
```

4.5.2 占位符参数

- 函数声明时，参数可以没有标识符，称这种参数为“占位符参数”，或称“哑元”。

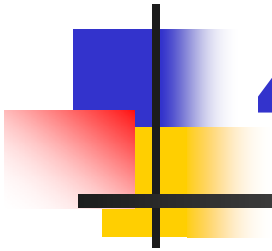
```
// 声明
 void f(int x, int = 0 , float = 1.1);

 // 定义
 void f(int x, int y, float flt)
 {x, flt, y}

 // 调用时必须为占位符参数提供一个值
 f(1)   ;
 f(1,2,3.0)
```

优点：增强代码的可维护性

将来修改函数功能需要增加一个形式参数时，可以利用该哑元，保证函数的接口不发生变化，从而不需要修改程序中的函数调用。例如：

```
long min(  long a, long b,  long )
{
      return (a>b?b:a);
}

Const int M=1;

void main( ){
        long  a = 100, b=200;
        long  c = min ( a, b, M);
}
```

```
long min(  long a, long b,  long c ) { …};
void main( ){
        long a = 100, b=200;
        long  c = min ( a, b, MAXNUMBER);
        long  d = min(a, b, c);
}
```



4.6 函数重载[注]

优于占位符，实际上是多个同名函数

4.6.1 重载的概念

- 重载：把多个功能类似的函数定义成同名函数

```
void main()
{
        int a, b, c;
        cout << “max(a,b)=”<<max(a,b)<<endl;
        cout << “max(a,b,c)=”<<max(a,b,c)<<endl;

}
```

4.6.2 重载的实现

核心问题：

- 编译器能够确定应执行哪一个函数。
  （由于是静态关联）

编译器根据什么来区分调用哪个函数？

- 函数名：相同，无法区分；
- 返回值类型：语句中不出现返回值类型，也无法区分；
- 参数：参数的个数和类型不同，可以区分。

4.6.2.1 参数类型不同的重载函数

```
#include <iostream.h>
int add(int,int);
double add(doule,double);

void main( ){
    cout << add(5,10) << endl;
    cout << add(5.1,10.5) << endl;
}

int add(int x, int y)   {  return x
                                  }

double add(double a, double b) { return a+b;   }
```

调用
add(int,int)

调用
add(double,double)

4.6.2.2 参数个数不同的重载函数

```
#include <iostream.h>
int min(int a, int b);
int min(int a, int b, int c
int min(int a, int b, int c
void main( )
{
    cout << min(13,5,4,9)<< endl;
    cout << min(-2,8,0);
}
int min(int a, int b)
{
    return a<b ? a : b;
}
```

调用
min(int,int,int,int)

调用
min(int,int,int)

```
int min(int a, int b, int c)
{
    int t = min(a,b);
    return min(t,c);
}

int min(int a, int b, int c, int d)
{
    int t1 = min(a,b);
    int t2 = min(c,d);
    return min(t1,t2);
}
```

4.6.2.3 参数次序不同的重载函数

```
char * get(char *s, char ch);
char * get(char ch, char *s);

// 其实是参数类型不同的一种特例
```

4.6.3 重载函数的“二义性”错误

- 返回值类型不同，参数相同

```
int    print(int);
double   print(int);

void main()
{
      int a=10, b=20;
      print(a);    // 二义性
      print(b);    // 二义性
}
```

调用哪个print???

4.6.3 重载函数的“二义性”错误(续)

- 仅用了const或引用造成的参数类型不同

```
int print(const int &);
int print(int );

void main()
{
    int a=10, b=20;
    print(a);    // 二义性
    print(b);    // 二义性
}
```

调用哪个print???

4.6.3 重载函数的“二义性”错误(续)

- 使用了修饰符造成的的参数类型不同的函数，可能引起二义：

```
void print(unsigned int );
void print(int);

int main(){
    print(1l);  // 二义性
    print(1u) ; // OK,无二义
    return 0;
}
```

1l 既可转换成unsigned int，又可转换成int。

调用哪个print???

4.6.3 重载函数的“二义性”错误(续)

- **缺省参数引起的二义性：**

```
void print(int a , int b=1);
void print(int a);

void main(){
      print(1);
}
```

调用哪个print???
到底是一个参数的print，还是取缺省值b为1的print.

4.6.4 编译器如何处理重载函数?

- C的函数编译后通常为 _cdecl_*funcname*();

  如：int max(int,int)　编译后为：
  int _cdecl_*max*()(0040260);

- 为了支持重载，C++的函数编译后通常为 _cppdecl_*funcname*_*parameter*_...();

  如：int max(int,int);
  int max(int,int,int) 编译后为：
  int _cppdecl_*max*_*int*_*int*()(0080330) ;
  int _cppdecl_*max*_*int*_*int*_*int*()(0100280) ;

- 关于extern “C”声明：

强制C++编译器按C的函数命名规则去连接相应的目标代码。

```
extern “c”
max(a,b);
```

编译为：

```
call _cdecl_max()(0040260);
```

补充：标准模板库

- 容器定义在C++标准模板库中
- std::vector是一种常用的容器，功能与数组相似

```
#include <vector>

int  main(){
      std::vector<float> v(3),w(3); //类模板
      v[0]=1;v[1]=2;v[2]=3;
      w[0]=7;w[1]=8;w[2]=9;
}
```

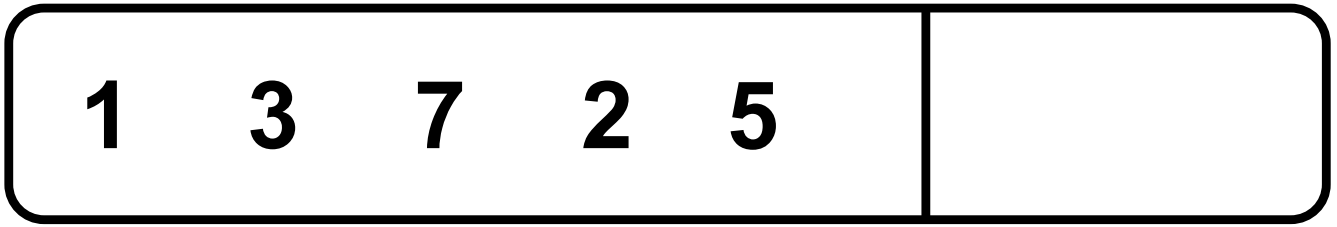
- 标准模板库（STL）提供标准容器和算法
- 迭代器（Iterator）用于连接容器和算法

```
std::vector<int> l={3,5,9,7};     //C++11

for (vector<int>::iterator it=l.begin();it!=l.end();++it)
{
    int i=*it;
    std::cout<<i<<std::endl;
}
```

与数组遍历比较?

```
std::vector<int> l={3,5,9,7};

for (auto it=begin(l),e=end(l);it!=e;++it)
{
    int i=*it;
    std::cout<<i<<std::endl;
}
```

- vector容器的内存规划



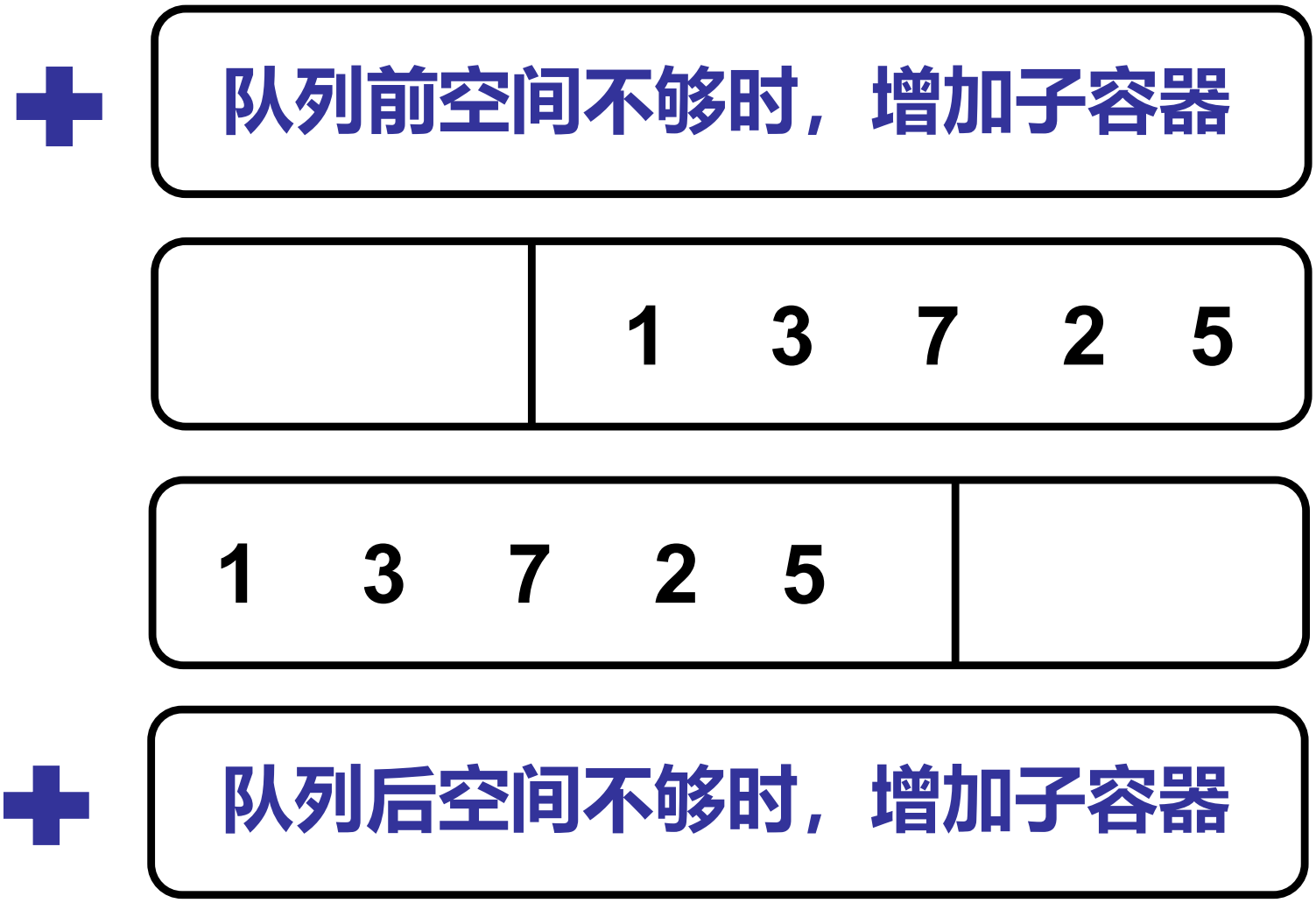
- vector添加元素时要么很快（直接在尾部多余空间添加）要么很慢（分配新空间，全部元素复制，再添加）

```
std::vector<int> v;

for (int i=0;i<100;++i)
{
    v.push_back(i);
}
```

- 测试以下代码，了解vector容器的基本内置操作

```
#include  <iostream>
#include  <vector>
#include  <algorithm>
using namespace std;

int main(){
      vector<int> v={3,4,7,9};
      auto  it=find(v.begin(),v.end(),4);
      cout<<*it<<*(it+1)<<endl;
      auto it2=v.insert(it+1,5);
      v.erase(v.begin());
      cout<<v.size()<<v.capacity()<<endl;
      v.shrink_to_fit();
      v.push_back(7);
      for (auto i:v)
            cout<<i<<endl;
}
```

- deque容器的内存规划



- deque首尾增删元素的速度几乎和vector一样快，且不需要复制

list容器（双向链表）

- 双向链接（iterator双向）
- 内存中离散分布，不支持随机读写
- 增删节点方便

```
int main(){
  list<int> l={3,4,7,9};
  auto it=find(begin(l),end(l),4),   it2=find(begin(l),end(l),7);
  l.erase(it);
  cout<<“it2 still points to”<<*it2<<endl;
}
```

- 为什么要用标准库？ 方便
- 大量预定义的数据结构和封装好的算法

```
std::vector<int> seq={3,4,7,9,5,7};
std::sort(seq.begin(),seq.end(),[](int x,int y){return x>y};)
```

Lambda函数

```
std::vector<int> some_list;
int total = 0;
for (int i = 0; i < 5; ++i) some_list.push_back(i);
std::for_each(begin(some_list), end(some_list),
[&total](int x){total += x;});
```