南京大学 计算机科学与技术系
Department of Computer Science & Technology, NJU

Chapter 8’

结构之补充内容

刘奇志

- **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）：

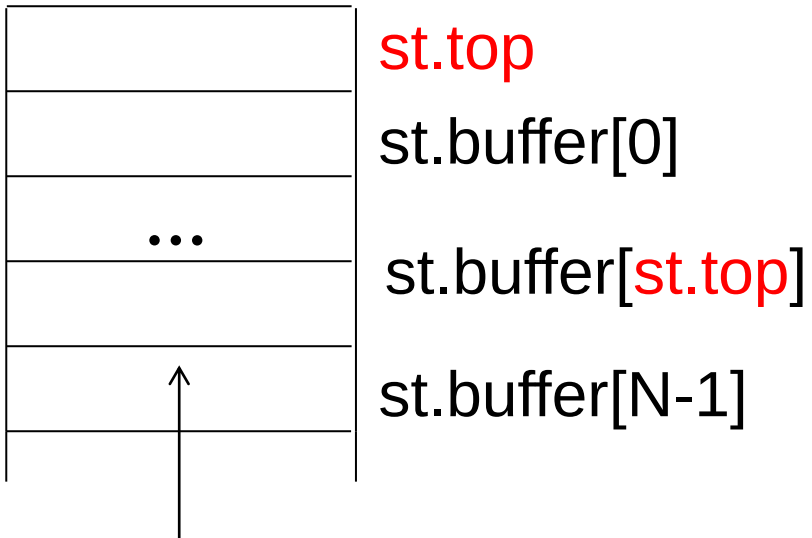
第（ i /20 ）个节点：

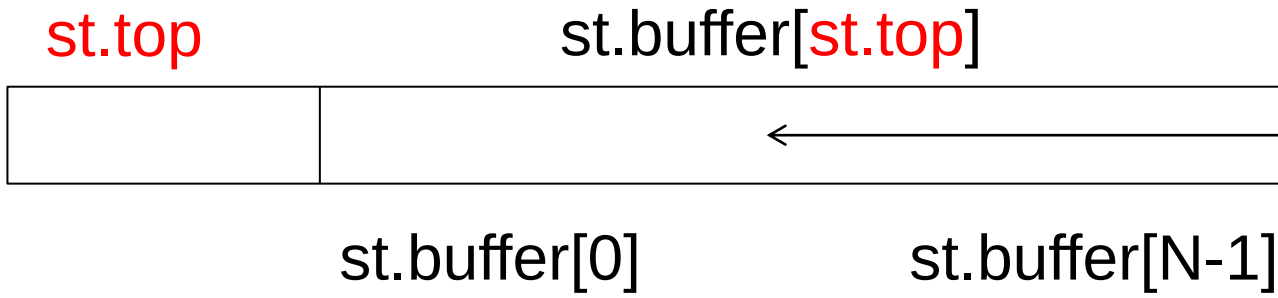
第（ p -> data[i%20] ）个元素

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;
}
```

```
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;
}
```

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

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

- **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;
      }
```

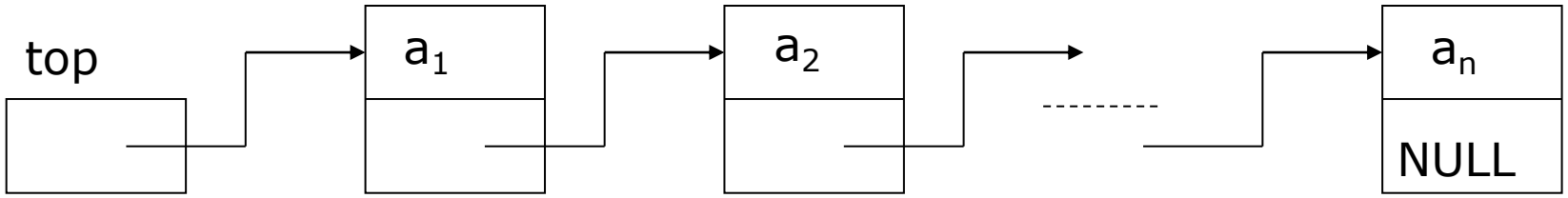
```
int main()
{     Stack st;          //会自动调用st.Stack()初始化，不允许 st.top = -1;
      st.Push(12);       //不允许或 st.top++; 或 st.buffer[st.top] = 12;
      int x;
      st.Pop(x);         //不允许st.Fun();
      …
      return 0;
}
```

- 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;
      }
}
```

类

- 对象是程序的基本计算单位，是数据及其操作的封装体，对象的特征则由相应的类来描述。
- 在C++中，类是一种用户构造出来的数据类型，构造形式如下：

```
class <类名>
{
    <数据成员描述>
    <成员函数描述>
};
```

- 类成员标识符的作用域为整个类构造范围

例：一个日期类的构造

```
class Date
{ public:
    void set(int y, int m, int d);
    //声明
    …
};

void Date::set(int y, int m, int d)
{ year = y;
  month = m;
  day = d;
} //写在外面（需用类名受限，区别于全局函数）
```

```
class Date
{ public:
    void set(int y, int m, int d)
    {   year = y;
        month = m;
        day = d;
    }
    bool is_leap_year()
    {   return (year%4 == 0 && year%100 != 0) || (year%400==0);
    }
    void print()
    {   cout << year << "." << month << "." <<day;
    }
    //成员函数，对数据成员所能实施的操作，写在类构造的里面，建议编译器按内联函数处理
  private:
    int year, month, day;
     //数据成员
};
```

成员

- **数据成员**
  - 是对类的对象所包含的数据描述。
  - 可以是常量和变量。
  - 类型可以是另一个类（成员对象）。
  - 描述格式与结构成员的描述格式相同。

```
class A
{  ......
public:
   void f();
   int f(int i);
   double f(double d);
   ......
};
```

- **成员函数**
  - 是对类构造中的数据成员所能实施的操作描述。
  - 类成员函数名可以重载（析构函数除外）。
  - C++也允许在结构（struct）和联合（union）中定义成员函数，但访问控制不受限制。

类成员的访问控制

- 在C++的类构造中，可以用访问控制修饰符public，private或protected来描述在类的外部对类成员的访问限制。默认访问控制是private。

```
class A
{
    int m;          //只能在本类和友元的代码中访问
  public:
    int x;          //访问不受限制
    void f();       //访问不受限制
  private:
    int y;          //只能在本类和友元的代码中访问
    void g();       //只能在本类和友元的代码中访问
  protected:
    int z;          //只能在本类、派生类和友元的代码中访问
    void h();       //只能在本类、派生类和友元的代码中访问
};
```

不属于该类成员，但可以访问类的私有成员，加 friend 修饰。在一定程度上破坏了类的封装性和隐藏性，不过提高了程序的运行效率。

- 一般来说，类的数据成员和在类的内部使用的成员函数应该指定为private，只有提供给外界使用的成员函数才指定为public。
- 具有public访问控制的成员构成了类与外界的一种接口（interface）。在一个类的外部只能访问该类接口中的成员。
- protected类成员访问控制具有特殊的作用（在派生类中使用）。

对象

- **类属于类型范畴的程序实体，它一般存在于静态的程序（运行前的程序）中。**
- **而动态的（运行中）的对象式程序则是由对象构成的。**
  - 对象在程序运行时创建。
  - 程序的执行是通过对象之间相互发送消息来实现的。
  - 当对象接收到一条消息后，它将调用对象类中定义的某个成员函数来处理这条消息。

对象的定义

- 在程序中定义一个类型为类的变量，其格式与普通变量的定义相同。

```
class A
{ public:
      void f();
      void g();
  private:
      int x, y;
}
......
A a1;          //创建一个A类的对象
A a2[100];     //创建100个A类对象
```

- 通过对象名来标识和访问
- 分为：全局对象、局部对象和成员对象

对象的创建（动态对象）

- 在程序运行时刻，用new操作来创建对象，用delete操作来撤消对象。对象通过指针来标识和访问。
  - 单个动态对象的创建与撤消

```
A *p;
p = new A;  //创建一个A类的动态对象
… *p …      //或 p->...，通过p访问动态对象
delete p;   //撤消p所指向的动态对象
```

  - 动态对象数组的创建与撤消

```
A *q;
q = new A[100];    //创建一个动态对象数组
...q[i]...         //或 *(q+i)，访问动态对象数组中的第i个对象
delete []q;        //撤消q所指向的动态对象数组
```

对象的操作

- 对于创建的一个对象，需要通过调用对象类中定义的某个public成员函数（也称向它发送消息）来操作。

```
class A
{   int x;
  public:
    void f() { ..... }
};

int main()
{ A a;              //创建 A 类的一个局部对象 a
  a.f();            //调用 A 类的成员函数 f 对对象 a 进行操作
  A *p = new A;//创建 A 类的一个动态对象，p 指向之
  p -> f();         //调用 A 类的成员函数 f 对 p 所指向的对象进行操作
  delete p;
  return 0;
}
```

通过对象来访问类的成员时要受类成员访问控制的限制

```
class A
{  public:
      void f()
      { ..... //允许访问: x, y, f, g, h
      }
   private:
      int x;
      void g()
      { ..... //允许访问: x, y, f, g, h
      }
   protected:
      int y;
      void h();
};
```

```
void func() //全局函数
{   A a;
    a.f();    // √
    a.x = 1; // X
    a.g();    // X
    a.y = 1; // X
    a.h();    // X
    ......
}
```

```
void A::h() //成员函数
{ ......     //允许访问: x, y, f, g, h
    A a;
    ...      //能a.x, a.y, a.g或a.h吗?
}            //可以! 因为是在成员函数中
```

- 可以对同类对象进行赋值

```
Date yesterday, today, some_day;
…
some_day = yesterday;          //把yesterday的数据成员分别赋给some_day的相应数据成员
```

- 取对象地址

```
Date *p_date;
p_date = &today;               //把today的地址赋值给对象指针p_date
```

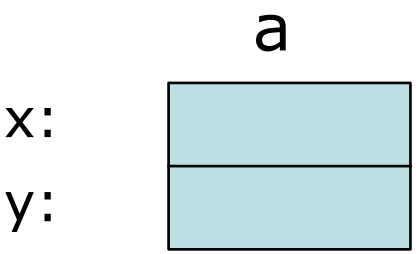
- 把对象作为实参传给函数以及作为函数的返回值

```
Date f(Date d)
{ Date x;
    ......
    return x;
}
some_day = f(yesterday);    //调用函数f，yesterday作为实参；返回值赋给some_day
```

对象的初始化--构造函数（ constructor ）

- 当一个对象被定义或创建时，它将获得一块存储空间，该存储空间用于存储对象的数据成员。
- 在使用对象前，需要对对象存储空间中的数据成员进行初始化。 C++提供了一种对象初始化的机制：**构造函数**（constructor）。构造函数是类的特殊成员函数，它的名字与类名相同、无返回值类型。创建对象时，构造函数会自动被调用来初始化对象的数据成员。例如：

```
class A
{   int x, y;
public:
    A() { x = 0;   y = 0;   } //构造函数
   ......
};
.......
A a; //创建对象a，即为a分配内存空间，然后调用A类的构造函数A()
```



- **构造函数可以重载，其中，不带参数的（或所有参数都有默认值的）构造函数被称为默认构造函数。例如：**

```
class A
{   int x,y;
  public:
    A() //默认构造函数
    { x = y = 0;
    }
    A(int x1)
    {   x = x1; y = 0;
    }
    A(int x1,int y1)
    {   x = x1; y = y1;
    }
    ......
};
```

- 在创建对象时，可以显式地指定调用对象类的某个构造函数进行初始化。
- 如果没有指定调用何种构造函数，则调用默认构造函数初始化。

```
class A
{      ......
  public:
       A();
       A(int i);
       A(char *p);
};
......
A a1;                  //调用默认构造函数。也可写成: A a1=A(); 但不能写成: A a1();
A a2(1);               //调用A(int i)。也可写成: A a2=A(1); 或 A a2=1;
A a3("abcd");          //调A(char *)。也可写成: A a3=A("abcd"); 或 A a3="abcd";
A a[4];                //调用对象a[0]、a[1]、a[2]、a[3]的默认构造函数
A b[5]={A(), A(1), A("abcd"), 2, "xyz"};
  //调用b[0]的A()、b[1]的A(int)、b[2]的A(char *)、b[3]的A(int)和b[4]的A(char *)
```

```
A *p1 = new A;                  //调用默认构造函数
A *p2 = new A(2);               //调用A(int i)
A *p3 = new A("xyz");           //调用A(char *)
A *p4 = new A[20];              //创建动态对象数组时只能调用各对象的默认构造函数
```

- **对象创建后，不能再调用构造函数!**

```
A a;
......
a.A(1); //X
```

成员初始化表

- 对于常量和引用数据成员，不能在说明它们时初始化，也不能采用赋值操作在构造函数中对它们初始化。例如：

```
class A
{  int x;
   const int y = 1;        //X
   int& z = x;             //X
 public:
   A()
   {     x = 0;            //√
         y = 1;            //X
         z = x;            //X
   }
};
```

- 可以在构造函数的函数头和函数体之间加入一个成员初始化表来对常量和引用数据成员进行初始化。例如：

```
class A
{  int x;
   const int y;
   int& z;
 public:
   A(): z(x), y(1)   //成员初始化表
   {     x = 0;
   }
};
```

析构函数（Destructors）

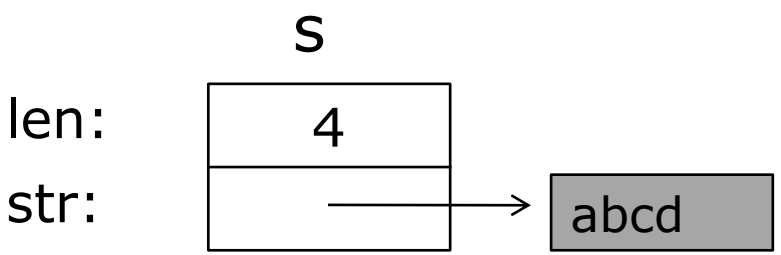
- 在类中可以定义一个特殊的成员函数：析构函数，它的名字为 ~<类名>，没有返回类型、不带参数、不能被重载。

```
class A
{    ......
  public:
     ......
     ~A();          //析构函数
};
```

- 一个对象消亡时，系统在收回它的内存空间之前，将会自动调用析构函数。
- 可以在析构函数中完成对象被删除前的一些清理工作（如：归还对象额外申请的内存空间等）。

```
class String
{   int len;
    char *str;
public:
    String(char *s)
    {       len = strlen(s);
            str = new char[len+1]; //额外申请的空间
            strcpy(str, s);
    }
    ~String()
    {       delete[]str;                    //归还额外申请的空间
            str = NULL;
    }
};
void f()
{   String s("abcd"); //调用s的构造函数
    ......
} //调用s的析构函数
```

系统为对象 s 分配的内存空间只包含 len 和 str（指针）本身所需的空间，str 所指向的空间不由系统分配，而是由对象自己申请！归还空间也是由系统和对象自己分别完成。



- 再例如：用链表实现的栈类，由于链表中的结点是对象创建后自己申请的，因此需要对象自己来归还它们！

```
class Stack
{  public:
       Stack() { top = NULL; }
       ~Stack() //析构函数
       { while(top != NULL)
          {  Node *p=top;
             top = top->next;
             delete p;
          }
       }
       void push(int i);
       void pop(int& i);
   private:
       struct Node
       {      int content;
              Node *next;
       } *top;
```

- **析构函数可以显式调用，这时并不是让对象消亡，而是归还对象额外申请的资源。例如，**

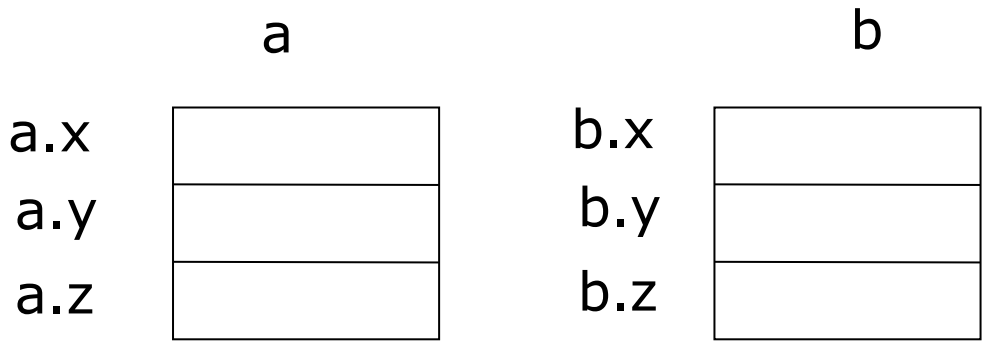
```
String s1("abcd");
……
s1.~String();  //只归还对象申请的资源，对象并未消亡!
… s1 …         //仍然可以使用对象s1
```

- **再如，**

```
Stack st;
… st.push …
……
… st.pop …
st.~Stack();  //清空栈st
… st.push …   //仍然可以使用栈st
……
… st.pop …
```

this指针：指向正在被某个成员函数操作的对象

```
class A
{ public:
      void g(int i) { x = i; }
      ..... //其它成员函数
  private:
      int x, y, z;
};
A a, b;
```




```
a.g(1);  //在函数 g 中，怎么找到对象 a 的 x 呢?
b.g(2);  //在函数 g 中，怎么找到对象 b 的 x 呢?
```

通过隐藏的 this 指针

- **实际上，类的每一个成员函数（静态成员函数除外）都有一个隐藏的形参this，其类型为该类对象的指针；在成员函数中对类数据成员的访问是通过 this 来进行的。**
  - A类的成员函数：`void g(int i) { x = i; }`

    实际上是：`void g(A *const this, int i) { this -> x = i; }`
  - 对于成员函数调用：`a.g(1);` 编译器将会把它编译成：`g(&a, 1);`
  - 对于成员函数调用：`b.g(2);` 编译器将会把它编译成：`g(&b, 2);`

静态成员函数只能访问类的静态成员（属于一个类的不同对象的共享数据）。
静态成员函数可以通过对象来访问外，也可以直接通过类来访问。加static。

- 一般情况下，类的成员函数中不必显式使用 this 指针来访问对象的成员，编译器会自动加上。
- 如果成员函数中要把 this 所指向的对象作为整体来操作（如：取对象的地址），则需要显式地使用 this 指针。例如：

```
void func(A *p);
class A
{ int x;
 public:
   void g(int i) { x = i; func(this); }
     ......
};
.......
A a, b;
a.g(1); //要求在g中调用func(&a)
b.g(2); //要求在g中调用func(&b)
```

拷贝构造函数

- 在创建一个对象时，若用另一个同类型的对象对其初始化，这时将会调用一个特殊的构造函数：拷贝构造函数。
- 拷贝构造函数的参数类型为本类的引用。例如：

```
class A
{ ......
  public:
    A();                  //默认构造函数
    A(const A& a);  //拷贝构造函数
};
```

- **在三种情况下，会自动调用类的拷贝构造函数：**
  - 初始化对象时，例如：

```
A a1;
A a2(a1); //也可写成：A a2 = a1; 或：A a2 = A(a1);
          //调用 A 的拷贝构造函数，用对象 a1 初始化对象 a2
```

  - 把对象作为值参数传给函数时，例如：

```
void f(A x);
A a;
f(a);  //调用 f 时将创建形参对象 x，并调用A的拷贝构造函数，用对象 a 初始化 x
```

  - 把对象作为函数的返回值时，例如：

```
A f()
{  A a;
   ......
   return a; //创建一个临时对象，并调用A的拷贝构造函数，用对象 a 对其进行初始化
}
```

隐式拷贝构造函数

- **如果程序中没有为类提供拷贝构造函数，则编译器将会为其生成一个隐式拷贝构造函数。**
- **隐式拷贝构造函数将逐个成员拷贝初始化：**
  - 对于普通成员：它采用通常的初始化操作
  - 对于成员对象：则调用成员对象类的拷贝构造函数来实现成员对象的初始化

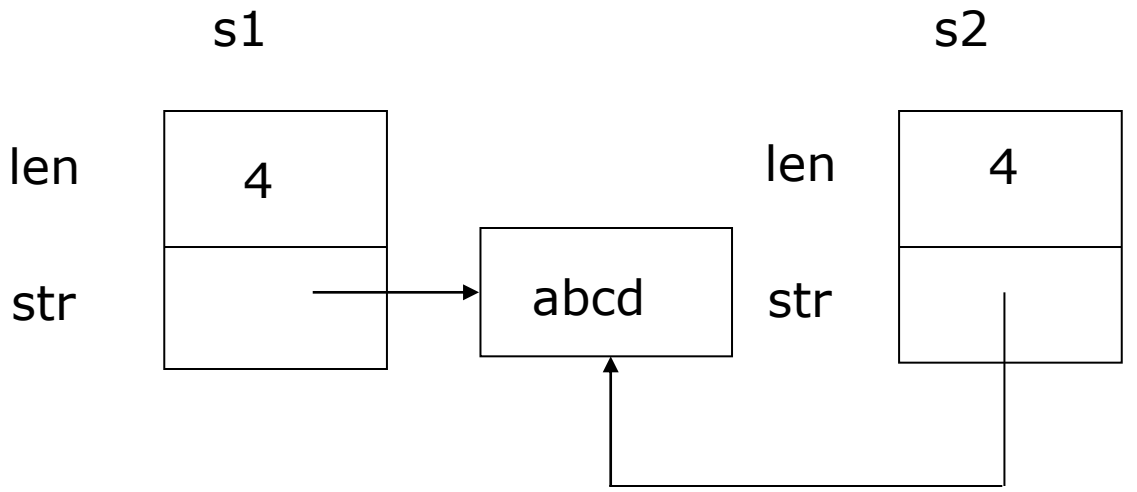
自定义拷贝构造函数*

- 一般情况下，编译器提供的隐式拷贝构造函数的行为足以满足要求，类中不需要自定义拷贝构造函数。
- 但在一些特殊情况下，必须要自定义拷贝构造函数，否则，将会产生设计者未意识到的严重的程序错误。

例如:

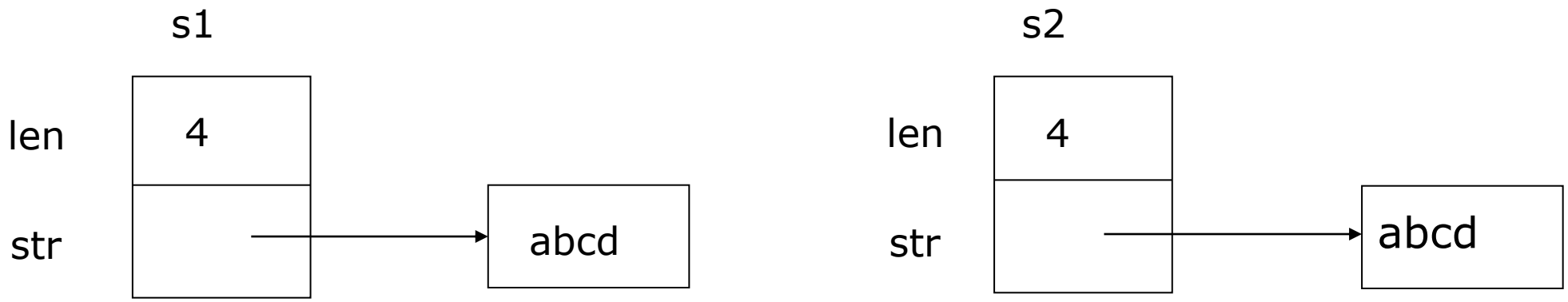
```
class String
{     int len;
      char *str;
  public:
      String(char *s)
      {   len = strlen(s);
          str = new char[len+1];
          strcpy(str, s);
      }
      ~A(){ delete []str; str = NULL;}
};
......
A s1("abcd");
A s2(s1);
```

系统提供的隐式拷贝构造函数将会使得 s1 和 s2 的成员指针 str 指向同一块内存区域!



- **解决上面问题的办法是在类 String 中显式定义一个拷贝构造函数**

```
String::String(const String& s)
{ len = s.len;
  str = new char[len+1];
  strcpy(str, s.str);
}
A s1("abcd");
A s2(s1);
```



操作符重载**

- C++语言本身没有提供复数类型，可通过构造一个类来实现：

```
class Complex  //复数类
{
public:
   Complex(double r, double i)      { real = r; imag = i; }
   void display()     { cout << real << '+' << imag << 'i';  }
     ......
private:
   double real;
   double imag;
};
```

- 如何实现两个复数（类型为Complex）相加操作?

- **可以为Complex类定义一个成员函数 add，例如：**

```
class Complex
{ public:
      Complex add(const Complex& x)
      {    Complex temp;
           temp.real = real + x.real;
           temp.imag = imag + x.imag;
           return temp;
      }
     ……
};
……
Complex a(1.0, 2.0), b(3.0, 4.0), c;
c = a.add(b);
```

- **这种操作形式不符合数学上的习惯：c = a + b;**

- **C++允许对已有的操作符进行重载，使得它们能对所构造类型（类）的对象进行操作。例如：**

```
class Complex
{ public:
    Complex operator + (const Complex& x) //重载操作符 +
    {       Complex temp;
            temp.real = real + x.real;
            temp.imag = imag + x.imag;
            return temp;
    }
     .......
};
……
Complex a(1.0, 2.0), b(3.0, 4.0), c;
c = a + b;
```

对象的赋值操作

- **同一个类的两个对象如何赋值?**

```
A a, b;
......
a = b; //?
```

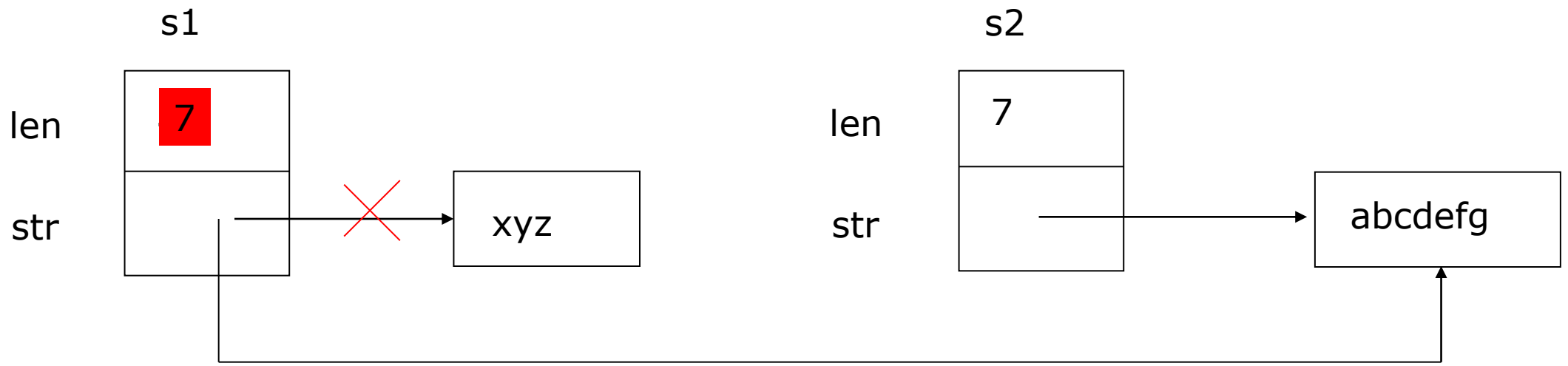
- **C++编译器会为每个类定义一个隐式的赋值操作，其行为是：逐个成员进行赋值操作（member-wise assignment）。**
  - 对于普通成员，它采用常规的赋值操作。
  - 对于成员对象，则调用该成员对象类的赋值操作进行成员对象的赋值操作。
- **隐式的赋值操作有时不能满足要求!**

- **例如，对下面 String 类的对象进行赋值操作时，系统提供的隐式赋值操作存在问题：**

```
class String
{     int len;
      char *str;
  public:
   String(char *s)
   {  len = strlen(s);
      str = new char[len+1];
      strcpy(str, s);
   }
   ~String() { delete []str; str = NULL; }
};
......
```

```
A s1("xyz"), s2("abcdefg");
.......
s1 = s2;
```

//赋值后，s1.str 原来所指向的**内存泄露**了，
//而且，s1 和 s2 互相干扰，s1 和 s2消 亡时，"abcdefg"所在的空间将会被**释放两次**!



- **解决上面问题的办法是自己定义赋值操作符重载函数:**

```
class String
{   ......
    String& operator = (const String& s)
    { if(&s == this) return *this;   //防止自身赋值
      delete []str;
      str = new char[s.len+1];
      strcpy(str, s.str);
        len = s.len;
      return *this;
    }
};
```

- **上面的返回值类型为什么是 String 的引用?**
  - 为了能进行下面的操作：(a = b) = c

- **一般来讲，需要自定义拷贝构造函数的类通常也需要自定义赋值操作符重载函数。**
- **注意：要区别下面两个 = 的不同含义。**

```
A a;
A b = a;        //初始化，等价于：A b(a);，调用拷贝构造函数
......
b = a;          //赋值，调用赋值操作符 = 的重载函数
```

数组元素访问操作符 [] 的重载

- 对于由具有线性关系的元素所构成的对象，可通过重载操作符 [ ] 来实现对其元素的访问。例如：

```
class String
{    int len;
     char *str;
  public:
       char& operator[](int i) { return str[i]; }
         ......
};
......
String s("abcd");
... s[2] ...                    //访问s中的'c'
```

Thanks !