数据抽象
——对象与类

(陈家骏老师ppt)

- 数据抽象与封装
- 类和对象
- 对象的初始化和消亡前处理

数据抽象与封装

- 数据抽象
  - 数据的使用者只需要知道对数据所能实施的操作以及这些操作之间的关系，而不必知道数据的具体表示。
- 数据封装
  - 指把数据及其操作作为一个整体来进行实现。
  - 数据的具体表示对使用者是不可见的，对数据的访问只能通过封装体所提供的对外接口（操作）来完成。
- 数据抽象与封装是面向对象程序设计的基础。

抽象数据类型及面向对象概念

抽象数据类型：

- 由用户定义，用以表示应用问题的数据模型
- 由基本的数据类型组成, 并包括一组相关的服务（或称操作）
- 信息隐蔽和数据封装，使用与实现相分离
- 抽象数据类型可用（D, S, P）三元组表示，其中，D 是数据元素的集合（简称数据对象），S 是 D 上的关系集合，P 是对 D 的基本操作集合。

例：自然数的抽象数据类型定义

ADT ***NaturalNumber*** **is**

objects:

一个整数的有序子集合,它开始于0,

结束于机器能表示的最大整数(*MaxInt*)。

Function: 对于所有的 $x, y \in NaturalNumber$;

False, *True* $\in$ *Boolean*, +、 -、 <、 ==、 =等都是可用的服务。

Zero( ) : NaturalNumber 返回自然数0

IsZero(x) :Boolean if (*x*==0) 返回*True*

else 返回*False*

Add (x, y) : NaturalNumber : if (*x*+*y*<=*MaxInt*)

返回 *x*+*y*

else 返回*MaxInt*

Equal (x, y) :Boolean if (*x*==*y*) 返回*True*

else 返回 *False*

Successor (x) : NaturalNumber if (*x*==*MaxInt*)

返回 *x*

else 返回 *x*+1

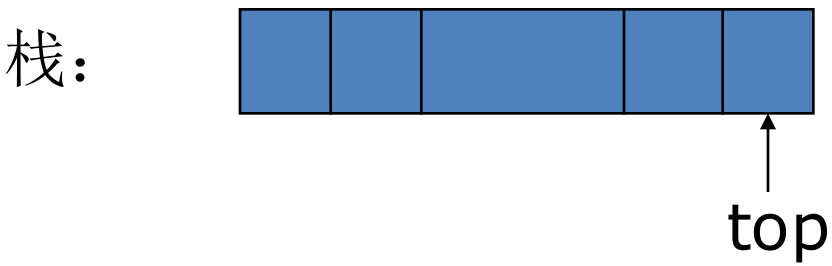
Subtract (x, y) :NaturalNumber` if (*x* < *y*) 返回 0

else 返回 *x* - *y*

end *NaturalNumber*

例：“栈”数据的表示与操作

- 栈是一种由若干个具有线性次序的元素所构成的复合数据。对栈只能实施两种操作：
  - 进栈（push）：往栈中增加一个元素
  - 退栈（pop）：从栈中删除一个元素
  - 上述两个操作必须在栈的同一端（称为栈顶，top）进行。后进先出（Last In First Out，简称LIFO）是栈的一个重要性质。
    - push(...);...pop(...);...;push(x);pop(y);
    - x == y



“栈”数据的表示与操作
——非数据抽象和封装途径

- 定义栈数据类型

```
const int STACK_SIZE=100;
struct Stack
{   int top;
    int buffer[STACK_SIZE];
};
```

- 直接操作栈数据

```
Stack st; //定义栈数据
int x;
//对st进行初始化。
st.top = -1;
//把12放进栈。
st.top++;
st.buffer[st.top] = 12;
//把栈顶元素退栈并存入变量x。
x = st.buffer[st.top];
st.top--;
```

- 存在的问题
  - 必需知道数据的表示
  - 数据表示发生变化将影响操作
  - 不安全

- 通过过程抽象操作栈数据

```
bool push(Stack &s, int i)
{   if (s.top == STACK_SIZE-1)
    { cout << “Stack is overflow.\n”;
      return false;
    }
    else
    { s.top++; s.buffer[s.top] = i;
      return true;
    }
}
bool pop(Stack &s, int &i)
{   if (s.top == -1)
    { cout <<“Stack is empty.\n”;
      return false;
    }
    else
    { i = s.buffer[s.top]; s.top--;
      return true;
    }
}
```

```
void init(Stack &s)
{   s.top = -1;
}
Stack st; //定义栈数据
int x;
init(st); //对st进行初始化。
push(st,12); //把12放进栈。
pop(st,x); //把栈顶元素退栈并存入变量x。
```

- 存在的问题
  - 数据类型的定义与操作的定义是分开的，二者之间没有显式的联系，push、pop在形式上与下面的函数没有区别：
    - void f(Stack &s);
  - 数据表示仍然是公开的，可以不通过push、pop来操作st，这样就可能破坏st栈的性质：
    - 在数据表示上直接操作
    - 通过函数f来操作

“栈”数据的表示与操作
——数据抽象和封装途径

- 定义栈数据类型

```
const int STACK_SIZE=100;
class Stack
{
public:
    Stack() { top = -1; }
   bool push(int i);
   bool pop(int &i);
private:
   int top;
    int buffer[STACK_SIZE];
};
```

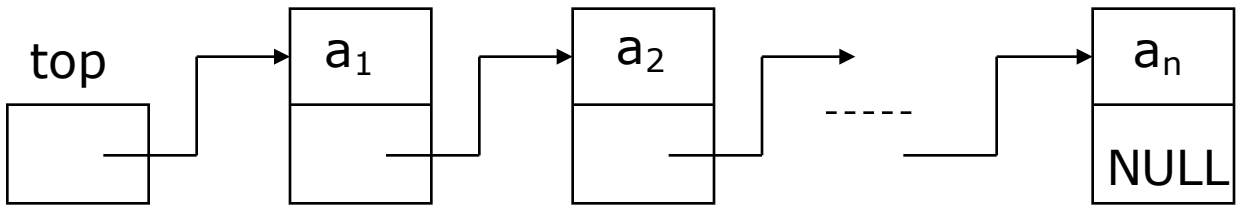
```
bool Stack::push(int i)
{    if (top == STACK_SIZE-1)
     {    cout << “Stack is overflow.\n”;
          return false;
     }
     else
     {    top++;  buffer[top] = i;
          return true;
     }
}
bool Stack::pop(int &i)
{    if (top == -1)
     {    cout << “Stack is empty.\n”;
          return false;
     }
     else
     {    i = buffer[top]; top--;
          return true;
     }
}
```

- 使用栈类型数据

```
Stack st;  //会自动地去调用st.Stack()对st进行初始化。
int x;
st.push(12);  //把12放进栈st。
st.pop(x);  //把栈顶元素退栈并存入变量x。
st.top = -1;  //Error
st.top++;  //Error
st.buffer[st.top] = 12;  //Error
st.f(); //Error
```

“栈”类的另一种实现
——用链表实现

```
class Stack
{   public: //对外的接口
     Stack() { top = NULL; }
     bool push(int i);
     bool pop(int& i);
    private:
     struct Node
     { int content;
       Node *next;
     } *top;
};
```



```
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;
     }
}
```

- 栈类型数据的实现变化了，但对使用者没有影响！

```
Stack st;
int x;
st.push(12);
st.pop(x);
```

类

- 对象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，类是一种用户自定义类型，定义形式如下：

  class <类名>{<成员描述>};

  其中，类的成员包括：
  – 数据成员
  – 成员函数

例：一个日期类的定义

```
class Date
{    public:
        void set(int y, int m, int d) //成员函数
        {       year = y;
                month = m;
                day = d;
        }
        bool is_leap_year() //成员函数
        {       return (year%4 == 0 && year%100 != 0) ||
                                                (year%400==0);
        }
        void print() //成员函数
        {       cout << year << "." << month << "." <<day;
        }
    private:
        int year,month,day; //数据成员
};
```

数据成员

- 数据成员指类的对象所包含的数据，它们可以是常量和变量。例如：

```
class Date //类定义
{    ......
   private:
     int year,month,day;  //数据成员说明
};
```

成员函数

- 成员函数描述了对类定义中的数据成员所能实施的操作。
- 成员函数的定义可以放在类定义中，例如：

```
class A
{     ...
   void f() {...} //建议编译器按内联函数处理。
};
```

- 成员函数的定义也可以放在类定义外，例如：

```
class A
{     ...
   void f(); //声明
};
void A::f() { ... } //需要用类名受限，区别于全局函数。
```

- 成员函数名是可以重载的（析构函数除外），它遵循一般函数名的重载规则。例如：

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

类成员的访问控制

- 在C++的类定义中，可以用访问控制修饰符public，private或protected来描述对类成员的访问限制。默认访问控制是private。 例如：

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

- 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）。操作一个类的对象时，只能通过访问对象类中的public成员来实现。
- protected类成员访问控制具有特殊的作用（继承，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。
- 而对象则存在于动态的程序（运行中的程序）中。
  - 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。

对象的创建和标识

- 直接方式
  - 通过在程序中定义一个类型为类的变量来实现的，其格式与普通变量的定义相同。例如：

```
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。
```

  - 分为：全局对象、局部对象和成员对象。
  - 对象通过对象名来标识和访问。

- 间接方式（动态对象）
  - 在程序运行时刻，通过new操作来创建对象，用delete操作来撤消（使之消亡）。
  - 动态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 … //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。
……//通过q访问动态对象数组
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成员函数来操作。例如：

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