Introduction to Object-Oriented Programming - Part II

Andreas Savva
University of Cyprus
30 January 2008
C++: Object-Oriented Programming

• Classes and Objects
• Template classes
• Operator Overloading
• Inheritance
• Polymorphism
The Evolution of The Notion of Object

• In C, a **struct** models what a thing has/is (i.e., the data, also called the *characteristics*), but not what it does (its *behavior*, represented by functions).

• The functions are outside and separate from structs.

• In C++, the characteristics and behavior are integrated into a single structure, called *object*.

• The data type of an object is *the class* of the object.

• The packaging of the data and the functions into a class type is called *data encapsulation*. 
Example: A Basic Stack
(Using C **`structs`**)

```c
struct stack {
    int data[100];
    int top;
} S;

int isEmpty(stack S){
    return S.top==0?1:0;
}

int pop(stack S){
    assert(top>0);
    return S.data[--S.top];
}

void push(stack S, int a){
    assert(top<100);
    S.data[top]=a;  S.top++;
}
```
Problems with structs

• Need a different struct for each different data type to be pushed onto or popped off the stack
• Overflow (although it can be fixed in C, using dynamic data allocation)
• The overall structure does not convey a tight coupling between the stack data and the stack operations
How struct Becomes in C++
(1st step: put the functions inside)

```cpp
struct stack {
    int data[100];
    int top;
    void push(int a); // implement outside
    int pop(void);   // implement outside
    bool isEmpty(void); // implement outside
};
```
## 2\textsuperscript{nd} Step: Implement the Functions

```cpp
void stack::push(int a) {
    assert(top<100);
    data[top]=a;
    top++;
}

bool stack::isEmpty() {
    return top==0?true:false;
}

int stack::pop() {
    assert(top>0);
    int x=data[--top];
    return x;
}
```
# How to Use the New Data Type

<table>
<thead>
<tr>
<th>Statements</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack S; S.top=0; S.push(12); S.push(20); S.push(30); <strong>int</strong> x=S.pop(); cout&lt;&lt;x&lt;&lt;endl; cout&lt;S.pop()&lt;&lt;endl; cout&lt;S.isEmpty()&lt;&lt;endl; cout&lt;S.pop()&lt;&lt;endl; cout&lt;S.isEmpty()&lt;&lt;endl</td>
<td>30 20 <strong>false</strong> 12 <strong>true</strong></td>
</tr>
</tbody>
</table>
Definition of Classes and Objects (revisited)

• A new data type defined with **struct** (where the data and functions are together) is called a **class**.
  – Example: the data type stack is a class

• Indeed, C++ has a new reserved word, **class**, that is synonymous to **struct** (except for a minor difference explained later)

• Any variable of the type defined by **struct** or **class** is called an **object** or **instance** of that class
  – Example: In the declaration: **stack S;**, S is an object of the stack class.
Definition of Members

• The declared variables inside structs are called the *data fields* in C.

• In C++, the declared variables and functions inside structs/classes are called *members*:
  – *member variables*
  – *member functions* (also called *methods*)
Data Hiding: Context

• Note that users of a stack object can access directly the members data[] and top
• For example, a user can do: \( S\.top \ -= 3; \)
• But notice that the 3 stack operations are all the users need. They do need to access top directly
• Also, this free access can cause problems:
  – If a user is not careful or is malicious, such a direct access to \( S\.top \) can invalidate the stack operations’ outcome
  – It also limits the implementer from evolving/modifying the implementation of the data type
Data Hiding: Private and Public sections

- C++, and other object-oriented programming languages, allow the programmer to designate certain members of a class as private, and other members as public.
- Private members cannot be accessed from outside the class, while public members can.
- Private members are hidden (thus the term data hiding)
The Stack with Data Hiding

**struct** stack {  // or **class** stack {
    private:
    int data[100];
    int top;
    public:
    void push(int a);
    int pop(void);
    bool isEmpty(void);
};

Now, observe which access is legal and which is illegal (will not compile):

stack S;
S.top = 10;  //illegal
S.data[17]=22;  // illegal
S.push(5);  //legal
S.isEmpty();  //legal
S.pop();  // legal
Initialization of Variables

• In C, initialization of variables is left to the user.
• For example, the member “top” of stack S is not initialized. All calls to push() or pop() cause errors.
• C++ considers initialization too important to leave to the users’ unreliable memory to remember to initialize.
• Even if users remember, observe that:
  – In the definition of stack, it is illegal to write: `int top=0;`
  – If top is private, it is illegal to write: `stack S; S.top = 0;`
  – And we don’t want to have top public (as we saw)
Initialization and Constructors

• C++ requires that every class have a *constructor*
• The constructor is responsible for initializing objects automatically whenever they are declared.
• The programmer has the option to provide his/her own constructors, which get called whenever a new object is declared.
• Otherwise, the compiler provides a default constructor.
• But what is a constructor?
Constructors (Contd.)

• The constructor is like any other method (i.e., member function) in that
  – it takes arguments
  – it can be overloaded
  – its arguments can be defaulted
• But with one big difference: It has no return value, not even `void`.
• One other very important characteristic: the constructor’s name is the same as the class name
Constructor of the Stack Class

class stack {
    private:
        int data[100];
        int top;
    public:
        stack(); // the constructor
        void push(int a);
        ... // the other functions
    };

    // constructor: initializes
    // top to 0.
    stack::stack() {
        top = 0;
    }

    // the implementation of
    // the other functions is
    // the same as before
How to Create Objects Using Constructors

• Example: stack S();
• This creates S as a stack, and initializes its member “top” to 0.
• Now, we can start to call S.push(int), S.pop(), and S.isEmpty() without problems.
Constructors with Arguments & Constructor Overloading

• Suppose that when you create a stack, you want to “stuff” it with several specified values at the outset.

• This can be done by writing a constructor that take the specified values as input, and initializes the data[ ] to those values.

• This is shown next.
class stack {
    private:
        int data[100];
        int top;
    public:
        stack();
        stack(int a[], int n);
        // other functions
};

// constructor: initializes data to a
stack::stack(int a[], int n) {
    for (top = 0; top < n && top < 100; top++)
        data[top] = a[top];
}

// constructor: initializes top to 0.
stack::stack() {
    top = 0;
}
Stack Overflow and Solution

• So far, the stack example we have cannot hold more than 100 values. If we push more than that many, the stack overflows.

• To overcome this limit, we can use dynamic arrays (with `new`), and maintain a capacity indicator

• This is implemented next
class stack {
private :
   int *dataptr;
   int top;
   int capacity;
public:
   stack(int cap=100);
   stack(int a[], int n);
   int getCapacity() {
      return capacity;
   }
   void push(int b);
   int pop();
   bool isEmpty() {
      return top==0;
   }
};

//constructor: sets capacity to // max(2n,100); stores a[] in stack
stack::stack(int a[],int n) {
   capacity = (2*n>100)?2*n:100;
   dataptr = new int[capacity];
   for(top=0;top<n;top++)
      dataptr[top]=a[top];
}

// constructor: initializes top to 0, // and capacity to cap if provided // & >0, otherwise to 100.
stack::stack(int cap) {
   top=0;
   capacity = (cap>0)?cap:100;
   dataptr = new int[capacity];
}
```cpp
void stack::push(int b){
    if (top < capacity)
        dataptr[top++]=b;
    else{
        // double the capacity, copy
        // current contents to new array,
        // delete old array, and push b on
        // top of the new array
        capacity *=2;
        int *newptr = new int [capacity];
        for(int k=0;k<capacity/2;k++)
            newptr[k]=dataptr[k];
        delete [] dataptr;
        dataptr = newptr;
        dataptr[top++]=b;
    }
}
```

**Inline Functions:**
Note that the methods isEmpty() and getCapacity() are implemented inside the class. Such functions are called *inline functions*. Inline functions should be limited to very simple implementations.
One Last Elaboration on the Stack Class: Generic Data Type

• The stack class allows us to push integers only
• Clearly, one may need a stack for floats or doubles or chars or even user-defined types
• Sure, one can design a different stack class for each different data type (cut and paste, and change the data type)
• But that is cumbersome, error-prone, and inefficient when it comes to making changes
• A better alternative: class templates
Class Templates

• Much as function templates allow argument types to be parameterized, class templates allow us to parameterize the types of:
  – member variables
  – arguments of member functions & constructors
  – return values of member functions

• The syntax is similar but somewhat more cumbersome
Class Templates Syntax

• For template class declaration:

  \[
  \text{template}<\text{class } T> \text{ class\_declaration;}
  \]

• For the implementation of the methods outside the class, the syntax is:

  \[
  \text{template}<\text{class } T> \text{ return\_type } className<T>::\text{methodName}(\text{parameter\_list})\{
    \ldots\
  \}
  \]

• For the implementation of the constructors outside the class, the syntax is:

  \[
  \text{template}<\text{class } T> \text{ className<T>:: className}(\text{parameter\_list})\{\ldots\}
  \]
template <class T> class stack {

private:
    T *dataptr;
    int top;
    int capacity;

public:
    stack(int cap=100);  // as before
    stack(T a[], int n); // new
    int getCapacity() {return capacity;}  
    void push(T b);
    T pop() {assert(top>0); return dataptr[--top];}
    bool isEmpty() {return top==0;}
};
//constructor: sets capacity to max(2n,100).
// It then initializes stack to a[].

_template<class T> stack<T>::stack(T a[],int n) {
  capacity = (2*n>100)?2*n:100;
  dataptr = new T[capacity];
  for(top=0; top<n; top++)
    dataptr[top] = a[top];
}

// constructor: initializes top to 0, and capacity to cap
// if provided & >0, otherwise to 100.
_template<class T> stack<T>::stack(int cap) {
  top=0;
  capacity = (cap>0)?cap:100;
  dataptr = new T[capacity];
}
template<class T> void stack<T>::push(T b) {
    if (top < capacity)
        dataptr[top++] = b;
    else {
        // double the capacity, copy
        // current contents to new array,
        // delete old array, and push b on
        // top of the new array
        capacity *= 2;
        T *newptr = new T[capacity];
        for (int k = 0; k < capacity/2; k++)
            newptr[k] = dataptr[k];
        delete [] dataptr;
        dataptr = newptr;
        dataptr[top++] = b;
    }
}
A Complete Program Using Template Stacks

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

// template stack definition goes here

int main(int argc, char *argv[]){
    stack<int> intS(5); // a stack of integers
    cout<<"intS capacity after construction = "<<intS.getCapacity()<<endl;
    int x[]={2,3,7,8,-10,14,5};
    for (int i=0; i<7; i++)
        intS.push(x[i]);
    cout<<"intS capacity after pushing 7 elements=
    Emptying intS: ";
    while (!intS.isEmpty())
        cout<<(intS.pop()<<"; ");
    cout<<endl;
}```
stack<\texttt{char *}> \texttt{stringS}(5); // a stack of strings
\texttt{stringS}.push("hi");
\texttt{stringS}.push("there");
cout<<"Emptying \texttt{stringS}: ";
\texttt{while} (!\texttt{stringS}.isEmpty())
    cout<<\texttt{stringS}.pop()<<"; ";
cout<<endl;

\texttt{double} \texttt{y[]}={3.14,9.8,1.42,12};
\texttt{stack<double>} \texttt{doubleS}(y,4); // a stack of doubles
\texttt{cout}<<"\texttt{doubleS} capacity=\"<<\texttt{doubleS}.getCapacity()<<endl;
\texttt{cout}<<"Emptying \texttt{doubleS}: ";
\texttt{while} (!\texttt{doubleS}.isEmpty())
    \texttt{cout}<<\texttt{doubleS}.pop()<<"; ";
cout<<endl;
The Output of the Last Program

intS capacity after construction = 5
intS capacity after pushing 7 elements=10
Emptying intS: 5; 14; -10; 8; 7; 3; 2;
Emptying stringS: there; hi;
doubleS capacity=100
Emptying doubleS: 12; 1.42; 9.8; 3.14;