

Data Structures and Algorithms II

Lecture 13: Inheritance and ADTs

Revision

- OOP and Inheritance.
- A simple “labelled stack” example!
- Using inheritance for ADTs.

1

Object Oriented Programming (OOP)

- Using inheritance in this way makes it relatively easy to *re-use* code in different applications, where minor variations of a basic approach are required.
- Libraries of generally useful datatypes (objects) are made available, but the programmer can easily define special customised versions with slightly different behaviour.
- Key concepts in object oriented programming are encapsulation and inheritance.
 - Encapsulation has been met before for ADTs - the internals of the data representation should be hidden, and all communication with the object should be via the specified methods.
 - Inheritance is where methods and data of parent (or ancestor) classes are used by instances of a given class.

3

Object Oriented Programming (OOP)

- Object-oriented programming can be viewed as an extension of the idea of abstract datatypes.
- Abstract datatype involves a data structure with a set of defined operations on that data structure.
- Object oriented programming adds inheritance – a given datatype (say, t1) may be a *subtype* of another (say, t2), and so access the operations and data of that other type.
- The *derived class* t2 can have additional data/methods..
- and if an operation (or data field) is defined for both t1 and t2, the version for t1 *overrides* the version for t2, and is the one used.
- We can create this slightly different, more specialised datatypes *without knowing how the first is implemented*.

2

Terminology

- Datatypes = *classes*; variables of given type = *instances* of the class.
- Operations = *methods*; subtypes = *subclasses*.
- Set of subclass relations defines a *hierarchy* (= tree structure of classes).
- Normal tree terminology used to refer to things (e.g., *parent* class).
- Where the methods or data of a parent class are used by an object we say that it *inherits* the methods or data.

4

Creating a Derived Class

To declare a new class to be derived (ie, a subclass of) another one we simply define our class as (say):

```
class DerivedClass : BaseClass
{ ...
```

If we do the above then BaseClass is a *private* base class. Which means that an instance of the derived class won't have access to the methods of the base class (they'll be *private* methods in the derived class).

Usually we want instances of the derived class to be able to use the methods of the base class, so we usually say:

```
class DerivedClass : public BaseClass
{ ...
```

The derived class still can't get directly at the base classes private data. It inherits the values, but can only access them via public methods.

5

Example: Named Stack

```
class NamedStack : public Stack {
public:
    NamedStack(char* name);
    ~NamedStack();
    char* GetName();
private:
    char* stkname;
};
```

Now namedstack has all methods of stack, so we could have:

```
NamedStack mystack('alison');

mystack.Pop();
cout << mystack.GetName();
```

7

Example: Named Stack

- Suppose we have defined and implemented (as a class) a stack ADT, with operations IsEmpty, Push, etc.
- Now we want a 'named stack' ADT, which also has a name or label associated with it. How would we create this?
- Obviously all the above operations are still relevant, we just need an additional datafield to store the name, and methods to access/modify the name. Say a `getName` method. And a new constructor that takes a name as an argument, allowing e.g.:

```
NamedStack s("a useful stack");
```

6

Named Stack Constructor

```
NamedStack :: NamedStack(char* name)
{
    stkname = new char[strlen(name) + 1];
    strcpy(stkname, name);
}
```

Above is a version that COPIES the name into the private datafield, allocating memory for it. It uses functions from `string.h`.

Could alternatively:

```
NamedStack :: NamedStack(char* name)
{
    stkname = name;
}
```

Consider the difference in these if you did:

```
char name[20];
cin >> name;
NamedStack ns1(name);
cin >> name;
NamedStack ns2(name);
```

8

Constructors

- In the example the (unnamed) stack constructor will automatically be invoked.
- But if your base class constructor has arguments, and these are different to your derived class constructor, you have to tell it which base class constructor to use, e.g. (using example from DS&A 1:

```
LJar::LJar(char* jlabel, int n) : Jar(n)
{
    ... // code for LJar constructor
}
```

This tells it to call the Jar constructor with argument n.

9

Example2: Stack derived from List

A very different example of using inheritance in ADTs is the following.

Suppose we have a general IntList class:

```
class IntList
{
public:
    IntList();
    void AccessFirst();
    void AccessNext();
    logical AtANode();
    int GetValue();
    logical Insert(int NewValue);
    void Delete();
    ...
}
```

How could we exploit the work done in creating this, if we then want to create a specialised list such as a stack?

11

Remaining Named Stack Implementation

Getname function is fairly trivial:

```
char* NamedStack::GetName();
{
    return stkname;
}
```

Destructor requires a little more work (if constructor allocates memory and copies).

```
NamedStack::~NamedStack()
{
    delete [] stkname;
}
```

The above is the format to use when you have created an array using pointers, and want to delete it.

10

Stack derived from List

We can make Stack a derived class of IntList, but IntList will be *private* (so Stack objects can't use its methods):

```
class Stack: IntList {
public:
    Stack() {};
    ~Stack() {};
    void Push(int x);
    void Pop();
    int GetValue();
    logical IsEmpty();
private:
    // no extra private data members
};
```

The new Stack's *methods* can use the methods inherited from IntList. Make sure current node is always the first node.

```
int Stack::GetValue()
{
    return IntList::GetValue();
}
```

12

```

void Stack::Pop()
{
    IntList::Delete();
}

void Stack::Push(int x)
{
    IntList::Insert(x);
    IntList::AccessFirst();
}

```

Note how we explicitly access IntList's methods.
We can now do:

```

Stack s;
s.Push(1);

```

but NOT:

```

Stack s;
s.Insert(1);

```

as that is *private* to IntList.

13

Summary

- Inheritance can be used to create new ADTs from old!
- Can either make new derived class have old one as *public* base class, so new class can access all the old methods, or make it *private* and define new ops using the old ones.
- Could also use same method to create different versions of an ADT - Can write new methods that override old.

15

Example 3: Creating a modified ADT

Maybe you just want to modify slightly the way a couple of methods in an ADT work; May be possible to use inheritance in similar way. Silly example:

```

class MyStack : public Stack
{
    logical Push(int);
}

int MyStack::Push(int x)
{
    logical ok = Stack::Push(x);
    if (ok=True) cout << ``Done it!``;
    else cout << ``Sorry, couldn't``;
    return ok;
}

```

14