DATA TYPES AND ABSTRACTION

Hunyadi Ioan Daniel, Popa M. Emil, Musan Mircea
Department of Informatics
University “Lucian Blaga” of Sibiu
5-7 Dr. Ioan Ratiu Street, Sibiu
ROMANIA

Abstract: In designing object-oriented programs, one of the primary concerns of the programmer is to develop an appropriate collection of abstractions for the application at hand, and then to define suitable abstract data types to represent those abstractions. In so doing, the programmer must be conscious of the fact that defining an abstract data type requires the specification of both a set of values and a set of operations on those values. In this paper I defined a C++ class hierarchy which is used to represent the basic repertoire of abstract data types.

Key-Words: abstraction, class hierarchy, container, data type, object, polymorphism.

1 Introduction

A variable in a procedural programming language such as Pascal, C and C++, is an abstraction. The abstraction comprises a number of attributes: name, address, value, lifetime, scope, type, and size. Each attribute has an associated value.

Unfortunately, the terminology can be somewhat confusing: The word “value” has two different meanings: in one instance it denotes one of the attributes and in the other it denotes the quantity assigned to an attribute.

The name of a variable is the textual label used to refer to that variable in the text of the source program. The address of a variable denotes its location in memory. The value attribute is the quantity which that variable represents. The lifetime of a variable is the interval of time during the execution of the program in which the variable is said to exist. The scope of a variable is the set of statements in the text of the source program in which the variable is said to be visible. The type of a variable denotes the set of values which can be assigned to the value attribute and the set of operations which can be performed on the variable. Finally, the size attribute denotes the amount of storage required to represent the variable.

The process of assigning a value to an attribute is called binding. When a value is assigned to an attribute, that attribute is said to be bound to the value. Depending on the semantics of the programming language, and on the attribute in question, the binding may be done statically by the compiler or dynamically at run-time.

2 Design Patterns

An experienced programmer is in a sense like a concert musician: she has mastered a certain repertoire of pieces which she is prepared to play at any time. For the programmer, the repertoire comprises a set of abstract data types with which she is familiar and which she is able to use in her programs as the need arises.

The repertoire of basic abstract data types has been designed as a hierarchy of C++ classes. I will present an overview of the class hierarchy.

The C++ class hierarchy which is used to represent the basic repertoire of abstract data types is shown in figure 1. Two kinds of classes are shown in figure 1: abstract C++ classes, which look like this

```
Abstract Class
```

and concrete C++ classes, which look like this

```
Concrete Class
```

Lines in the figure indicate derivation; base classes always appear to the left of derived classes.

![Figure 1: Object Class Hierarchy](image-url)
An abstract class is a class which specifies an interface only. It is not possible create object instances of abstract classes. In C++ an abstract class typically has one or more pure virtual member functions. A pure virtual member function declares an interface only - there is no implementation defined. In effect, the interface specifies the set of operations without specifying the implementation.

An abstract class is intended to be used by the base class from which other classes are derived. Declaring the member functions virtual makes it possible to access the implementations provided by the derived classes through the base-class interface. Consequently, we don't need to know how a particular object instance is implemented, nor do we need to know of which derived class it is an instance.

This design pattern uses the idea of polymorphism. Polymorphism literally means "having many forms". The essential idea is that a single, common abstraction is used to define the set of values and the set of operations - the abstract data type. This interface is embodied in the C++ abstract class definition. Then, various different implementations (many forms) of the abstract data type can be made. This is done in C++ by deriving concrete class instances from the abstract base class.

### 2.1. Objects

The abstract class at the top of the class hierarchy is called Object. With the exception of the Ownership class, all the other classes in the hierarchy are derived from this class. Program 1 gives the declaration of the Object class. Altogether only six member functions are declared: the destructor, IsNull, Hash, Put, Compare and CompareTo.

Program 1: Object Class Definition

class Object
{
protected:
virtual int CompareTo(Object const&) const = 0;
public:
virtual ~Object();
virtual bool IsNull() const;
virtual int Compare(Object const&) const;
virtual HashValue Hash() const = 0;
virtual void Put(ostream &s) const = 0;
};

The declaration of the NullObject class is given in Program 2. The NullObject class is a concrete class derived from the Object abstract base class. The NullObject class is also a singleton class. It is a class of which there will only ever be exactly one instance. The one instance is in fact a static member variable of the class itself. In order to ensure that no other instances can be created, the default constructor is declared to be a private member function. The static member function Instance returns a reference to the one and only NullObject instance.

Program 2: NullObject Class Definition

class NullObject: public Object
{
static NullObject instance;
    NullObject();
protected:
    int CompareTo(Object const&) const;
public:
    bool IsNull() const;
    HashValue Hash() const;
    void Put(ostream &s) const;
    static NullObject & Instance();
};

One of the design goals of the C++ language is to treat user-defined data types and the built-in data types as equally as possible. However, the built-in data types of C++ do have one shortcoming with respect to user-defined ones. A built-in data type cannot be used as the base class from which other classes are derived.

The usual design pattern for dealing with this deficiency is to put instances of the built-in types inside a wrapper class and to use the wrapped objects in place of the built-in types. Program 3 illustrates this idea. The class Wrapper<T> is a concrete class which is derived from the abstract base class Object. It is also a generic class, the purpose of which is to encapsulate an object of type T. Each instance of the Wrapper<T> class contains a single member variable of type T called datum.

Program 3: Wrapper<T> Class Definition

template <class T>
class Wrapper: public Object
{
protected:
    T datum;
    int CompareTo(Object const&) const;
public:
    Wrapper();
    Wrapper(T const &);
    Wrapper& operator = (T const &);
    operator T const &() const;
    HashValue Hash() const;
    void Put(ostream &s) const;
};
Program 1 uses the Wrapper<T> class defined above to define the Int, Char, Double, and String object classes which simply encapsulate variables of type int, char, double, and string, respectively.

```cpp
typedef Wrapper<int> Int;
typedef Wrapper<char> Char;
typedef Wrapper<double> Double;
typedef Wrapper<std::string> String;
```

These declarations require the existence of suitable ::Compare and ::Hash functions. The Int class definition requires the existence of a Compare(int) and a Hash(int) function.

### 2.2. Containers

The Container class abstracts the notion of a container: an object that holds within it other objects. A container is itself an object. Therefore, the Container class is derived from the Object class. A consequence of this is that containers can be held in other containers!

The declaration of the Container class is given in Program 4. Notice that the Container declaration uses multiple inheritances. It is derived from two base classes, Object and Ownership. We have already seen the definition of the Object class. The Ownership class will be defined.

#### Program 4: Container Class Definition

```cpp
class Container: public virtual Object, public virtual Ownership
{
protected:
    unsigned int count;
    Container();
public:
    virtual unsigned int Count() const;
    virtual bool IsEmpty() const;
    virtual bool IsFull() const;
    virtual HashValue Hash() const;
    virtual Iterator& NewIterator() const;
    virtual void Put(ostream &); const;
    virtual void Purge(); = 0;
    virtual void Accept(Visitor &); const;
};
```

The Container class is an abstract base class. It is intended to be used as the base class from which concrete container realizations are derived. The Container class public interface comprises eight virtual member functions - Count, IsEmpty, IsFull, Hash, Put, NewIterator, Purge and Accept. Default implementations are provided for the first six; the last two are pure virtual member functions.

A single member variable, count, is declared. This variable is used to keep track of the number of objects held in the container. The count field is set initially to zero. It is the responsibility of the derived class to update this field as required.

The Count member function is an accessory that returns the number of items contained in the container. The IsEmpty and IsFull functions are Boolean-valued accessors which indicate whether a given container is empty or full, respectively. The Hash function is used in conjunction with hash tables and the Put function inserts a human-readable representation of the container and its contents in a specified output stream. The purpose of the Purge function is to remove all the objects from a container, making it empty.

The notion of a container as an object which contains other objects was introduced. Conspicuous by their absence from the interface are member functions for putting objects into the container and for taking objects out of the container. As it turns out, the particular forms of the functions required depend on the type of container implemented. Therefore, we have left the specification of those functions for the classes which are derived from the Container base class.

Another reason for leaving these matters unspecified, is that we have not yet defined what it means for one object to be contained within another! We have two options: Direct Containment when an object is put into a container, a copy of that object is made in the container, and Indirect Containment when an object is put into a container, a pointer to that object is kept in the container.

The main advantage of using direct containment is its simplicity. It is easy to understand and easy to implement. However, it does suffer some problems. First, if the objects which are to be put into a container are large, if they occupy a large amount of memory space, the copying of the objects is likely to be both time-consuming and space-consuming. Second, an object cannot be contained in more than one container at a time. Third, a container cannot contain itself.

The indirect containment approach addresses these three concerns. By keeping a pointer to the contained object rather than a copy of the contained object, it is not necessary to copy the object when it is put into the container. This saves time and space. By keeping pointers to the contained objects, it is possible to put a pointer to the same object into several different containers. Finally, it is possible to
put the pointer to a container into the container itself. The indirect containment approach is not without disadvantages. The principal disadvantage being that for every access to a contained object it is necessary to dereference a pointer. This adds a constant overhead to such accesses.

2.3. Visitors

The Container class described in the preceding section interacts closely with the Visitor class shown in Program 5. In particular, the Accept member function of the Container class takes as its lone argument a reference to a Visitor.

Program 5: Visitor Class Definition
class Visitor
{
 public:
  virtual void Visit (object&)=0;
  virtual bool IsDone () const
  { return false; }
};

But what is a visitor? A shown in Program 5, a visitor is an object that has the two member functions Visit and IsDone. Of these, the Visit function is the most interesting. The Visit function takes as its lone argument a reference to an Object instance.

The interaction between a container and a visitor goes like this. The container is passed a reference to a visitor by calling the container's Accept member function. The container “accepts” the visitor. What does a container do with a visitor? It calls the Visit member function of that visitor one-by-one for each object contained in the container.

The interaction between a Container and its Visitor are best understood by considering an example. The following code fragment gives the design framework for the implementation of the Accept function in some concrete class, say SomeContainer, which is derived from the abstract base class Container:

void SomeContainer::Accept (Visitor& visitor) const
  for each Object i in this container
    visitor.Visit (i);

The Accept function calls Visit for each object i in the container. Since the class Visitor is an abstract base class which does not provide an implementation for the Visit operation, what the visitor actually does with an object depends on the type of visitor used.

Suppose that we want to print all of the objects in the container. One way to do this is to create a PrintingVisitor which prints every object it visits, and then to pass the visitor to the container by calling the Accept member function. The following code shows how we can declare the PrintingVisitor class which prints an object on the standard output stream, cout.

class PrintingVisitor : public Visitor
{
 public:
  void Visit (object& object)
  { cout << object; }
};

Finally, given a container c that is an instance of a concrete container class SomeContainer which is derived from the abstract base class Container, we can call the Accept function as follows:

SomeContainer c;
PrintingVisitor v;
c.Accept (v);

The effect of this call is to call the Visit member function of the visitor for each object in the container.

As shown in Program 5, the Visitor class interface also includes the member function IsDone. The IsDone member function is an accessor which is used to determine whether a visitor has finished its work. The IsDone member function returns the boolean value true if the visitor “is done”.

The idea is this: Sometimes a visitor does not need to visit all the objects in a container. In some cases, the visitor may be finished its task after having visited only a some of the objects. The IsDone member function can be used by the container to terminate the Accept function like this:

void SomeContainer::Accept (Visitor& visitor) const
  for each Object i in this container
    if (visitor.IsDone ())
      return;
    visitor.Visit (i);

To illustrate the usefulness of IsDone, consider a visitor which visits the objects in a container with
the purpose of finding the first object that matches a
given object. Having found the first matching object
in the container, the visitor is done and does not
need to visit any more contained objects.

The following code fragment defines a visitor
which finds the first object in the container that
matches a given object.

```cpp
class MatchingVisitor : public Visitor
{
    Object const& target;
    Object* found;
public:
    MatchingVisitor(Object const & target):target(target), found(0) {}
    void Visit (Object& object)
    {
        if (found==0 && object==target)
            found = &object;
    }
    bool IsDone ()
    { return found != 0; }
};
```

The constructor of the `MatchingVisitor`
visitor takes a reference to an `Object` instance that
is the target of the search. I wish to find an object in
a container that matches the target. For each object
the `MatchingVisitor` visitor visits, it compares
that object with the target and makes `found`
point at that object if it matches. Clearly, the
`MatchingVisitor` visitor is done when the
`found` pointer is non-zero.

Suppose we have a container `c` that is an instance
of a concrete container class, `SomeContainer`, which is derived from the abstract base class
`Container`; and an object `x` that is an instance of
a concrete object class, `SomeObject`, which is
derived from the abstract base class `Object`. Then,
we can call use the `MatchingVisitor` visitor as follows:

```cpp
SomeContainer c;
SomeObject x;
MatchingVisitor v (x);
c.Accept (v);
```

The observant reader will have noticed in
Program 5 that the `Visit` member function of the
abstract `Iterator` class is `pure` virtual function whereas the `IsDone` function is not. It turns out
that it is convenient to define a default
implementation for the `IsDone` function that
always returns `false`.

### 2.4. Ownership of Contained Objects

A matter that is closely related to the containment
of objects is the `ownership` of objects. Ownership is
important because it is the owner of an object that
ensures that the object's destructor is called and that
any storage dynamically allocated for the object is
freed when the object is no longer needed.

The ownership of contained objects is clear when
direct containment is used. Since the container
makes a copy of any object put into the container, it
is the container that is responsible for deleting the
copy when it is longer needed. In particular, the
destructor for the container would normally delete
all contained objects.

If we assume that it is the responsibility of the
container to delete the contained objects, then we
must make sure that the only objects put into a
container are objects whose storage was dynamically
allocated. This is because as the owner of the object,
the container must delete the object when the time
comes to clean up. But, given a pointer to an object,
it is not possible for the container to know whether
the object it points to has been dynamically allocated
or whether it is actually a statically allocated global
variable or a stack allocated local variable.

Another consequence of assigning the ownership
of objects to a container is that things become
complicated when an object is put into two or more
different containers as well as when a given object is
put into the same container more than once. The
problem is that an object inserted in more than one
container has more than one owner. In order to
ensure that the object is only deleted once, we would
have to extract the pointer from all but one of the
containers before deleting them.

On the other hand, if we assume that it is not the
responsibility of the of the container to delete
contained objects, then the responsibility to clean up
falls on the user of the container. In order to ensure
that all contained objects are properly deleted, it is
necessary to extract all of the contained objects from
a container before deleting the container itself.

The solution to this dilemma is to support both
paradigms. The `Ownership` class given in
Program 6 does precisely that.

### Program 6: Ownership Class Definition

```cpp
class Ownership
{
    bool isOwner;
protected:
    Ownership() : isOwner(true) {}
    Ownership(Ownership & arg):isOwner
    { arg.isOwner=false; }
};
```
public:
    void AssertOwnership()
    {isOwner=true};
    void ReadingOwnership()
    {isOwner=false};
    void isOwner()const
    {return isOwner};
};

The Ownership class encapsulates a single Boolean variable, isOwner, which records whether the container is the owner of the contained objects. By default, the isOwner field is set to true in the constructor. Two member functions, AssertOwnership and RescindOwnership, provide a means for the user of a container to change the state of the isOwner datum. The IsOwner accessor reveals the current ownership status.

The behaviour of the copy constructor is subtle: It transfers ownership status from the original container to the copy. This behaviour is useful because it simplifies the task of returning a container as the result of a function.

3 Conclusions

It is said that “computer science is the science of abstraction”. But what exactly is abstraction? Abstraction is “the idea of a quality thought of apart from any particular object or real thing having that quality”. For example, we can think about the size of an object without knowing what that object is. Similarly, we can think about the way a car is driven without knowing its model or make.

Abstraction is used to suppress irrelevant details while at the same time emphasizing relevant ones. The benefit of abstraction is that it makes it easier for the programmer to think about the problem to be solved.

Acknowledgements: The research was supported by the Grant Agency of CNMP (grant No. 73-CEEX-II-03 from 31/07/2006).

References:

[3] Martin Erwig, Programs are Abstract Data Types, 16th IEEE Int. Conf. on Automated Software Engineering, pp. 400-403, 2001