

Appendix A. Object Oriented Design

A.1. Introduction

A thorough grounding in the principles of Object Oriented Design is invaluable in Component-Level design. A complete discussion of this topic is beyond the scope of this book, but we offer a few tips that will simplify the design process for those who have not yet taken a course in Object Oriented Design. The information provided here will allow you to create objects in a consistent way, and will permit you to complete the exercises in this book without too much difficulty. Please remember that this is a collection of programming tips, and not substitute for a course in Object Oriented Design.

A.2. Private, Protected or Public: When to Use Them

One problem that many programmers have is trying to decide when to make class variables private, when to make them public, and when to make them protected. As a general rule of thumb, one should make all variables private unless there is an overriding reason not to do so. Initially, the temptation will be to make all data items public, and then selectively make some of them private or protected. This is the precise opposite of the approach you *should* take. The reason people take the wrong approach is because they are used to thinking in terms of functional languages and structures. Classes are not structures, nor should they be treated as structures. That being said, here are a few guidelines to help you decide between public, private, and protected.

A.2.1. Public Variables

A variable that is an object should *never* be made public. A pointer to an object can be made public if it is not the head of a linked list or some other complex structure. The only reason for making such a variable public is to allow the outside world to “give” a subobject to an object of your class. It is never permissible to allow the outside world to manipulate the internal elements of subobjects. When in doubt, *do not* make the pointer public.

Regardless of the type of the variable, if the value of the variable has any required relationship with any other class variable, including implicit variables like the drawing window, it *must not* be made public. If there are any restrictions on the value of the variable other than those imposed by its type, it *must not* be made public. If the variable is a member of a collection of variables that are normally changed as a group (e.g. *Height* and *Width*), then the variable *must not* be made public. If the variable is read-only after it has been initialized, it *must not* be made public. If the variable should not be modified even by member functions after it has been initialized, it should be declared to be a constant.

Public variables are variables of simple types, whose values can be changed without restriction and whose values have no relationship to the value of any other variable in the class. Needless to say, such variables are uncommon.

A.2.2. Protected Variables

If a variable has access restrictions, but can be manipulated in an arbitrary way by member functions, then it should be made protected. If the value of the variable has a required relationship to other variables, but this relationship can change in derived

classes, then the variable should be made protected. However, if the value of the variable has unchanging relationships with other variables, then it should be made private, and derived classes should use accessor and mutator functions to access the variable.

Typically variables that are subobjects should be made protected as long as there are no additional restrictions in the way the subobject is accessed.

A.3. Objects Versus Classes

Many authors are careful to distinguish between objects and classes. A class is a formal description containing definitions of variables and functions. The distinction is generally expressed as follows: an object is an instance of a class that has storage allocated for it, and values assigned to its variables. This distinction is useful, but it ignores the fact that the structure of some objects cannot be described using a single class. A singly linked list, for example, has a head, a tail, and a list of items chained together with pointers. The description of the linked list requires at least two classes, one to define the head and tail pointers, and one to describe the items in the list. Nevertheless, the linked list is a single object.

There are various ways to handle multi-class objects in C++, none of which is completely satisfactory. Using the linked list as an example, let us assume that we wish to create a queue of (x,y) values using a singly linked list. The class declarations of Figure A.1 could be used for this purpose. There are several design flaws in Figure A.1. First, users of these classes can create **CElement** objects if they wish. This violates the spirit of these declarations, since together they represent the design of a single object. Furthermore, the rules for variable access permissions given in the preceding section have been violated. In class **CElement** x and y are normally set as a group, therefore they

must not be public. In class **CQueue** *Head* and *Tail* have a required relationship and cannot be made public. Furthermore, *Head* does not point to a simple object, but to a list of objects. Let us first make these corrections, and then proceed with tying the two classes together. The corrections appear in Figure A.2.

```
class CElement
{
public:
    CElement * Next;
    double x;
    double y;
    CElement( );
    ~CElement( );
};

class CQueue
{
public:
    CElement *Head;
    CElement *Tail;
    CQueue();
    ~CQueue();
};
```

Figure A.1. Two Coordinated Classes.

```

class CElement
{
public:
    CElement * Next;
private:
    double x;
    double y;
public:
    CElement( );
    CElement(double NewX, double NewY);
    ~CElement( );
    void GetValues(double &RetX, double &RetY);
};

class CQueue
{
private:
    CElement *Head;
    CElement *Tail;
public:
    CElement();
    ~CElement();
    void Push(double NewX, double NewY);
    void Pop(double &RetX, double &RetY);
};

```

Figure A.2. Corrected Classes.

Once the two class descriptions have been corrected, it is necessary to add accessor functions to access the values of the protected elements, and constructors to assign them values. For the **CQueue** class, the two functions *Push* and *Pop* will be used to add and delete elements from the queue. The implementation of the *Push* function is given in Figure A.3.

```

void CQueue::Push(double NewX, double NewY)
{
    // constructor will set Next to NULL
    CElement *NewElem = new CElement(NewX, NewY);
    if (Head == NULL)
    {
        Head = NewElem;
    }
    else
    {
        Tail->Next = NewElem;
    }
    Tail = NewElem;
}

```

Figure A.3. The CQueue Class.

The *Next* data item of **CElement** is declared as **public** to permit the statement

```
Tail->Next = NewElem;
```

to compile properly.

This will allow the queue to function properly, but will not reserve the class **CElement** for the exclusive use of the **CQueue** class. (This can be good or bad, depending on your requirements.) There are two ways to tie the two classes together so that **CElement** belongs only to **CQueue**. The first is to move the definition of **CElement** inside of the definition of **CQueue**, and declare the **CElement** class to be a private member of class **CQueue**. This works just fine, but tends to be a bit unreadable. If this is done, the function body of the default constructor would be declared using the syntax `CQueue::CElement::CElement()`. Other function bodies would require the same syntax. Note that **CQueue** does not gain any special permissions within the **CElement** class by virtue of the containment.

Another, more readable way to do the same thing is to declare all functions of **CElement** to be private, including all constructors. This will make *any* use of the **CElement** class illegal, because no external entity, including the global environment, has permission to call the default constructor or any other constructor for **CElement** objects. Then we explicitly give **CQueue** permission to access the **CElement** class by declaring **CQueue** to be a friend of **CElement**. Figure A.4 shows how this is done. This solution is not completely satisfactory either, because **CQueue** now has permission to access *all* data items of **CElement**, not just the *Next* pointer.

```
class CQueue; // forward reference required.
class CElement
{
    friend CQueue;
private:
    CElement * Next;
    double x;
    double y;
    CElement( );
    CElement(double NewX, double NewY);
    ~CElement( );
    void GetValues(double &RetX, double &RetY);
};
```

Figure A.4. CElement With Friend Declaration

A.4. Parameterized Constructors

Proper use of private, public, and protected attributes can complicate the initialization process for derived objects when a parameterized constructor is used. Consider the two classes pictured in Figure A.5.

```
class CThing
{
private:
    long Size;
public:
    CThing();
    CThing(long NewSize);
    ~CThing();
};

class COther : public CThing
{
private:
    long Quality;
public:
    COther();
    COther(long NewSize, long NewQuality);
    ~COther;
};
```

Figure A.5. An Initialization Problem.

When an object of type **COther** is created using the constructor *COther(long, long)*, there is no way to assign the value of *NewSize* to the *Size* variable. Although the

constructor for **CThing** is called before the constructor for **COther**, it is the *default* constructor, *CThing()* that is called, not *CThing(long)*. It is possible to override the selection of the default constructor in the body of *COther(long,long)*. This is done as shown in Figure A.6.

```
CThing::CThing(long NewSize)
{
    Size = NewSize;
}

COther::COther(long NewSize, long NewQuality) : CThing(NewSize)
{
    Quality = NewQuality;
}
```

Figure A.6. Initializing Base Classes.

A similar method can be used to initialize aggregate classes within an object.

Consider the class pictured in Figure A.7.

```
class CMore
{
private:
    CThing First;
    CThing Last;
public:
    CMore();
    CMore(long FirstSize, long LastSize);
    ~CMore();
};
```

Figure A.7. An Aggregate Initialization Problem.

The constructor *CMore(long,long)* should be defined as shown in Figure A.8.

```
CMore::CMore(long FirstSize, long LastSize) :
    First(FirstSize), Last(LastSize)
{
}
```

Figure A.8. Initializing Aggregate Classes.

If a class has constant items, they must be initialized using the same syntax as Figure A.8, namely **Variable-Name(Value)**. Any number of initializers can be used after the colon separated by commas. The base class initializers can be mixed with the member initialization items.

A.5. Pure Encapsulation

In most functional programs it is possible to classify the various subroutines into categories. There are routines that perform file management, others that handle internal storage, and so forth. In Object Oriented Design, each category of subroutines should be implemented as a class. If there are global variables associated with the functions, they must be made members of the class. In many cases the functions grouped in this manner will not change significantly. When this happens, it is known as Pure Encapsulation. The class serves only to group similar functions together. Grouping functions in this manner will generally make the code much easier to read and to modify. An example of pure encapsulation is the Mouse Handler used in the Simple Graphical Editor. (See Chapter 7.)

A.6. Virtual Base Classes and Multiple Inheritance

Suppose you are creating a set of classes that will be used by other programmers to create *Useful Objects*. There are three features that may be incorporated into a *Useful Object*, which we will call *Feature A*, *Feature B*, and *Feature C*. These features are completely independent of one another. A particular *Useful Object* may have any combination of these features. Regardless of the features that a *Useful Object* has, there are certain data items that it must incorporate. To implement *Useful Objects*, you create four classes, **CUseful**, which contains the basic data items, **CUsefulA** which inherits

from **CUseful**, and implements *Feature A*, **CUsefulB**, which also inherits from **CUseful**, and implements *Feature B*, and finally **CUsefulC** which inherits from **CUseful** and implements *Feature C*. To create a new *Useful Object*, your client programmers will create a new class and derive the class from **CUsefulA**, **CUsefulB**, or **CUsefulC**, depending on which features they need. If they need more than one feature, they will use multiple inheritance and derive their new class from two or more of **CUsefulA**, **CUsefulB**, and **CUsefulC**. Under normal circumstances, this would create multiple copies of the basic data items. However, you had the foresight to declare class **CUseful** to be a virtual class, as shown in Figure A.9. When a virtual class is inherited from more than one class in multiple inheritance, only a single copy of that class is created. Thus in a multiple inheritance that derives from **CUsefulA** and **CUsefulB**, there is a single copy of the base class **CUseful**. Note that *virtual* is a property of the inheritance, not a property of the base class.

```
class CUseful
{
    ...
}

class CUsefulA : virtual public CUseful
{
    ...
}

class CUsefulB : virtual public CUseful
{
    ...
}

class CUsefulC : virtual public CUseful
{
    ...
}
```

Figure A.9. Virtual Base Class Declarations.

A.7. The Deficiencies of C++

C++ is an excellent language for Object Oriented Programming. Nevertheless there are some additional features that would make life easier for Object Oriented programmers. In this section lists the features that I would like most to see added to C++.

A.7.1. Read Only and Write Only Data Items

The choice between public and private for a variable is too restrictive. Consider the class of Figure A.10 that defines a variable-sized array.

```
class CVarIntArr
{
private:
    long Size;
    int * Data;
};
```

Figure A.10. A Variable Array Class.

In Figure A.10, it is necessary to make both variables, *Size* and *Data* private, because they have a required relationship, namely that *Size* must contain the size of the array pointed to by *Data*. However, not every access to these variables would destroy this relationship. In fact, only a write access would be a problem. Unfortunately, there is no way to declare these variables to be read-only. The problem is particularly annoying when attempting to access the elements of the array. An access of the form `X.Data[i]` should be legal regardless of whether it is a read access or a write access. It would be more helpful to separate the read and write portions of the access and declare each one separately, as in Figure A.11.

```
class CVarIntArr
{
private-write:
    long Size;
    int * Data;
public-read:
    long Size;
    int * Data;
};
```

Figure A.11. Separated Read/Write Access.

Unfortunately, the example of Figure A.10 is not legal C++.

Naturally, one can provide (or *not* provide) accessor and mutator functions for a private variable to restrict access rights to it. And one can (and should) declare these functions to be inline so that there is no function-call overhead when using them, but having separate read/write access rights would be a more convenient and natural way to program the same thing.

A.7.2. Friend access rights

When you declare a class B to be a friend of class A, class B immediately gains read/write access to everything in class A. This is a little bit like taking a new friend home and immediately giving him permission to go into your bedroom and dig through your underwear drawer. I am very much in favor of limited friendships where the access rights of friend functions and classes are clearly defined. In the specification of friend access rights, it would be nice if the read and write access to variables could be specified separately. Figure A.12 shows what I have in mind. Lets assume that the keyword “grant” is used to grant specific rights, and the keyword “restrict” is used to remove access rights.

```
class B
{
    ...
}
class A
{
    friend class B grant (read X, write Y, call F);
    ...
}
class C
{
    friend class B restrict(write Size, write Data);
    ...
}
```

Figure A.12. Restricted Friend Access Rights.

In Figure A.12, class B would be able to read variable X, write variable Y, and call function F, but would be unable to access the members of A in any other way. Class C has full access to the members of B, except for Size and Data, which are read-only. Unfortunately, the syntax of Figure A.12 is illegal.

This sort of access can be designed into a class using the existing features of C++, but it is quite cumbersome to do so. One technique is to separate the data items and functions into different sets, depending on the access rights we wish them to have. Each separate set could then be placed in its own sub class. Each subclass could then have its own friend declarations. The subclasses would then be aggregated into a containing class, which would have universal friend access to all of them. This becomes even more cumbersome if the member functions of one subclass must access the data items of another subclass. Each subclass could contain a pointer to the aggregated object, but the result is would be complicated mess that is best avoided.

On the whole, I feel that it is best to avoid friend functions and classes unless it is absolutely necessary to use them.

A.7.3. Metamorphic Objects

The primary use for polymorphic objects is to avoid the cumbersome decoding of type codes. We have made use of this fact in designing the Simple Graphical Editor of Chapter 7. When calling the *Draw* function of a **CGraphicalObject**, it is actually the *Draw* function of the subtype that is called. In essence, the virtual functions of the object take the place of the type code. We could probably use virtual function pointers *exactly* the way a type code is used, if we had it in mind to brutalize ourselves. However, the virtual function gives us immediate access to the routine that the type-decoder would select, and is thus significantly more useful than a type code for identifying an object.

Unfortunately, type codes are not the only state variables in an object. Consider the Simple Graphical Editor. The support class variable *Mouse.Code* is a state variable that determines the action that will be performed the next time the *OnLButtonUp* function is called. The *OnLButtonUp* has a decoding routine that determines what action will be taken based on the value of *Mouse.Code*. However, when the value is assigned to *Mouse.Code*, the future action of the *OnLButtonUp* function is completely known. We find ourselves in the position of losing that information, only to regain it later by decoding the *Mouse.Code* variable. It would be far simpler to be able to replace the *OnLButtonUp* function with one that did exactly what we wanted. If we could replace the *OnLButtonUp* function, and the *OnMouseMove* function, we could eliminate the *Mouse.Code* variable entirely, along with the decoding routines. Doing this is called *Metamorphosis*, because the identity of the object is being changed by replacing its virtual functions.

Of course, one could use function pointers to achieve the same thing, but this technique permits the unconditional replacement of a virtual function with another function of the same type. We wish to restrict the replacement to a set of predefined functions.

True metamorphosis can be achieved using polymorphic subobjects. (I believe that C++ should support metamorphosis directly, but until that happens, polymorphic subobjects are a workable substitute.) To achieve our goal, we first define a base class of type **CMouseHandler**. To this class we add two pure virtual functions named *OnMouseMove* and *OnLButtonUp*. We place no data items in the class. We then proceed by deriving new classes from **CMouseHandler**, one for each value of *Mouse.Code*. In the definitions of the *OnLButtonUp* and *OnMouseMove* functions, we place the code to handle that particular value of *Mouse.Code*. Figure A.13 shows the definition of **CMouseHandler**, and the derived class used to handle the *Mouse.Code* **MDMove**. The definition of the **MDMove** *OnLButtonUp* function is also given in Figure A.13. The functions *OnLButtonUp* and *OnMouseMove* must have the same parameters as the original *OnLButtonUp* and *OnMouseMove* functions, and one additional pointer that will point to the support object. Unlike true metamorphic functions, our new *OnMouseMove* and *OnLButtonUp* functions do not have automatic access to the variables of the support class.

```

class CMouseHandler
{
public:
    virtual void OnMouseMove(UINT nFlags,CPoint point,
                             CSGEditorCtrl *Base)=0;
    virtual void OnLButtonUp(UINT nFlags,CPoint point,
                              CSGEditorCtrl *Base)=0;
}

class CMouseHandlerMDMove : public CMouseHandler
{
public:
    void OnMouseMove(UINT nFlags,CPoint point,CSGEditorCtrl *Base);
    void OnLButtonUp(UINT nFlags,CPoint point,CSGEditorCtrl *Base);
}

void CMouseHandlerMDMove::OnLButtonUp(UINT nFlags,CPoint point,
                                       CSGEditorCtrl *Base)
{
    // undraw the selection rectangle
    Base->DoSRect();
    // move the selection.
    long XDelta = Base->Mouse.EndPoint.x - Base->Mouse.StartPoint.x;
    long YDelta = Base->Mouse.EndPoint.y - Base->Mouse.StartPoint.y;
    Base->Model->MoveSelection(XDelta,YDelta);
    Base->CWnd::Invalidate();
}

```

Figure A.13. Derived Classes for Emulating Metamorphosis.

Several changes must now be made to the support class. Each of the classes derived from **CMouseHandler** must be declared as friends of the support class, **CSGEditorCtrl**. We must also add a public variable, *MH*, of type *CMouseHandler ** to the **CMouse** class. In the constructor for the **CMouse** class, we must allocate a new object of type **CMouseHandlerMDNone** and assign its pointer to *MH*. We must alter the *Down* and *Up* functions of the **CMouse** class as shown in Figure A.14.

```
void CMouse::Down(CMouseHandler * NewHandler)
{
    delete MH;
    MH = NewHandler;
}

void CMouse::Up()
{
    delete MH;
    MH = new CMouseHandlerMDNone;
}
```

Figure A.14. Changes to the CMouse Functions.

Now we must alter the calls to the **CMouse** *Down* function to reflect the change in the operand. A former call of the form *Mouse.Down(MDMove)* must be replaced with *Mouse.Down(new CMouseHandlerMDMove)*. Finally, we replace the switch statement of *OnMouseMove* with *Mouse.MH->OnMouseMove(nFlags,point,this)*, and similarly for the switch statement of the *OnLButtonUp* function.

Once these changes have been made, we can eliminate the *Code* member of the **CMouse** class. We leave it to the reader to decide which implementation style is preferable.

A.8. Conclusion

The material presented in this appendix represents Object Oriented Programming and Design tips. There are a number of important issues that we have not mentioned. The reader is strongly encouraged to take a course in the subject.