Sixteen Ways to Stack a Cat

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper presents a series of examples of how to represent stacks in a program. In doing so it demonstrates some of the fundamental techniques and tradeoffs of data hiding as seen in languages such as C, Modula2, and Ada. Since all the examples are written in C++ it also demonstrates the flexibility of C++'s mechanisms for expressing data hiding and access.

1 Introduction

Consider representing a stack of objects of some type in a program. Several issues will affect our design of the stack class: ease of use, run time efficiency, cost of recompilation after a change. We will assume that messing around with the representation is unacceptable so that data hiding for the representation is a must. We will assume that many stacks are necessary. The type of the elements on the stacks is of no interest here so we will simply call it `cat`. The implementations of the various versions of stacks are left as an exercise for the reader.

Please note that the purpose is to illustrate the diversity of data hiding techniques, not to show how best to represent stacks in C++. The techniques shown here apply to a variety of types – most of which are more interesting than stacks – and will be used in conjunction with other techniques. For example, if you actually wanted to build a better stack for C++ you would probably start by making `cat` a parameter, that is, use templates†.

This paper is a fairly lighthearted play with C++ features and techniques. I think it has something to delight and possibly horrify novices and seasoned C++ programmers alike. Most of the techniques shown have serious uses, though.

2 Files as Modules

Consider the traditional C notion of a file as a module. First we define the interface as a header file:

```
// stack interface, stack.h:

typedef int stack_id;

stack_id create_stack(int);
void destroy_stack(stack_id);

void push_stack(stack_id,cat);
cat pop_stack(stack_id);
```

The integer argument to `create_stack` is the maximum size of the desired stack. We can now use stacks like this:

† See Chapter 14 of Ellis&Stroustrup: The Annotated C++ Reference Manual. Addison Wesley. 1990.

```
#include "stack.h"

void f(int sz, cat kitty)
{
        stack_id s = create_stack(sz);
        push_stack(s,kitty);
        cat c2 = pop_stack(s);
        destroy_stack(s);
}
```

This is rather primitive, though. Stacks are numbered, rather than named; the concept of the address of a stack is ill defined; copying of stacks is undefined; the lifetimes of stacks are exclusively under control of the users; the technique relies on the convention of `.h` files and on comments to express the concept of a stack; the names of the stack functions are clumsy.

From a C++ perspective, the problem is that there are no stack objects defined. These stacks do not obey the general language rules for naming, creation, destruction, access, etc. Instead, little ‘‘cookies’’ (the `stack_id`s) are passed to functions that manipulate stack representations.

Note that in many contexts it would be reasonable not to impose a maximum size on a stack. Not imposing a maximum would allow a noticeable simplification of the stack interface. However, this would decrease the value of the stack example as a vehicle for discussion of general data hiding issues because most types do require arguments to the operations that create objects.

3 Stack Identifiers

We can do a little better. First let us make `stack_id` a genuine type:

```
// stack interface, stack.h:

struct stack_id { int i; };

stack_id create_stack(int);
void destroy_stack(stack_id);

void push_stack(stack_id,cat);
cat pop_stack(stack_id);
```

This at least will prevent us from accidentally mixing up stack identifiers and integers:

```
#include "stack.h"

void f(int sz, cat kitty)
{
        stack_id s = create_stack(sz);
        push_stack(s,kitty);
        cat c2 = pop_stack(sz); // error: stack_id argument expected
        destroy_stack(s);
}
```

This error would not have been caught by the compiler given the first definition of `stack_id`. These first two versions both have the nice property that the implementation is completely hidden so that it can be changed without requiring recompilation of user code as long as the interface remains unchanged.

4 Modules

Now let us use a class to specify this stack module. Doing that will allow us to avoid polluting the global name space and relying on convention and comments to specify what is and isn’t part of a stack’s interface:

```
// stack interface, stack.h:

class stack {
public:
        struct id { int i; };

        static id create(int);
        static void destroy(id);

        static void push(id,cat);
        static cat pop(id);
};
```

We can use this module like this:

```
#include "stack.h"

void f(int sz, cat kitty)
{
        stack::id s = stack::create(sz);
        stack::push(s,kitty);
        cat c2 = stack::pop(s);
        stack::destroy(s);
}
```

This looks very much like a Modula-2 module. We don't have a 'with' or 'use' construct to avoid the repetition of ``stack::''. For example:

```
void f(int sz, cat kitty)
{
        use (stack) {                   // pseudo code
                id s = create(sz);
                push(s,kitty);
                cat c2 = pop(s);
                destroy(s);
        }
}
```

However, such a construct for merging name spaces is a syntactic detail and does not always lead to more readable code. Further, an even more radical simplification of the notation is achieved in section 10.

Note that `static` member functions were used to indicate that the member functions do not operate on specific objects of class `stack`; rather, class `stack` is only used to provide a name space for `stack` operations. This will be made explicit below.

5 Modules with Sealed Identifiers

To make the correspondence to Modula-2 modules more exact, we need to stop people from messing around with the stack identifiers and stop them from trying to create (C++ style) stack objects:

```
// stack interface, stack.h:

class stack {
public:
        class id {
                friend stack;
        private:
                int i;
        };

        static id create(int);
        static void destroy(id);

        static void push(id,cat);
        static cat pop(id);
private:
        virtual dummy() = 0;
};
```

The representation of an *id* is now accessible only to class `stack`, ‘‘the implementation module for `stack`s,’’ and class `stack` is an abstract class so that no objects of class `stack` can be created:

```
stack x;  // error: declaration of object of abstract class stack
```

The use of a pure virtual function to prevent the creation of objects is a bit obscure, but effective. An alternative technique would have been to give `stack` a private constructor.

Naturally, the example of stack usage from section 4 works exactly as before.

6 Packages

In the style of Modula-2, we are now passing around ‘‘little cookies’’ (opaque types) used by the implementation module to identify the objects. If we want we can pass around the objects themselves (or pointers to them) in the style of Ada:

```
// stack interface, stack.h:

class stack {
public:
        class rep {
                friend stack;
        private:
                // actual representation of a stack object
        };

        typedef rep* id;

        static id create(int);
        static void destroy(id);

        static void push(id,cat);
        static cat pop(id);
private:
        virtual dummy() = 0;
};
```

The typedef is redundant, but it allows our user code to remain unchanged:

```
#include "stack.h"

void f(int sz, cat kitty)
{
        stack::id s = stack::create(sz);
        stack::push(s,kitty);
        cat c2 = stack::pop(s);
        stack::destroy(s);
}
```

That `rep` is very much like an Ada private type. Users can pass it around but not look into it.

One disadvantage is that by placing the representation of stacks in the declaration of class `stack` we force recompilation of user code when that representation changes. This can lead to a major increase in compilation time. Actually, this recompilation isn't necessary, because the user code *and* that code's use of information about the implementation was left unchanged. A reasonably smart dependency analyser could deduce that no recompilation of user code is necessary after even a radical change to the representation. However, a dumb (that is, time stamp on source-file based) dependency analyser will not deduce that and will recompile user code just because the representation was changed. A smart dependency analyser will rely on smaller units of change, on understanding of the semantics of C++, or on both to minimize recompilation. Smart dependency analysers are rumored to exist, but they are not widely available.

On the positive side, the representation information is now available to the compiler when compiling user code so that genuine local variables can be declared and used:

```
#include "stack.h"

void g(int sz, cat kitty)
{
        stack::rep s;
        stack::push(&s,kitty);
        cat c2 = stack::pop(&s);
}
```

This is unlikely to work without additional code in the implementation, though, unless some mechanism for initializing a stack representation exists. This could be done by adding suitable constructors and destructors to class `rep`, but it would be more in the spirit of packages to provide an explicit initialization function. It would also be natural to support use reference arguments to eliminate most explicit pointers.

```
// stack interface, stack.h:

class stack {
public:
        class rep {
                friend stack;
                // actual representation of a stack object
        };

        static rep* create(int);
        static void destroy(rep&);

        static void initialize(rep&,int);
        static void cleanup(rep&);

        static void push(rep&,cat);
        static cat pop(rep&);
private:
        virtual dummy() = 0;
};
```

The `create()` function is now redundant (a user can write one without special help from the class), but I have left it in to cater for code and coding styles that relies on it. If needed, it could be made to return a reference instead of a pointer.

```
#include "stack.h"

void h(int sz, cat kitty)
{
        stack::rep s;
        stack::initialize(s,sz);
        stack::push(s,kitty);
        cat c2 = stack::pop(s);
        stack::cleanup(s);
}
```

The `cleanup()` operation is needed because the `initialize()` operation and/or the `push()` operation are likely to grab some free store to hold the elements. Unless we want to rely on a garbage collector we must clean up or accept a memory leak.

Now enough information is available to the compiler to make inlining of operations such as `initialize()`, `push()`, and `pop()` feasible even in an implementation with genuine separate compilation. The definitions of the functions we want inlined can simply be placed with the type definition:

```
// stack interface, stack.h:

class stack {
public:
        class rep {
                friend stack;
                // actual representation of a stack object
        };


        // ...

        static cat pop(rep& x)
        {       // extract a cat from the representation of stack x
                // return that cat
        }

        // ...
};
```

Inlining and genuine local variables can be essential to make data hiding techniques affordable in applications where run time efficiency is at a premium.

7 Packages with Controlled Representations

Alternatively, if we did not want users to allocate `rep`s directly we could control the creation and copying of `rep`s by making these operations private:

```
// stack interface, stack.h:

class stack {
public:
        class rep {
                friend stack;

                // actual representation of a stack object

                rep(int);                   // constructor
                rep(const rep&);            // copy constructor
                void operator=(const rep&); // assignment operator
        };

        static rep* create(int);
        static void destroy(rep&);

        static void initialize(rep&,int);
        static void cleanup(rep&);

        static void push(rep&,cat);
        static cat pop(rep&);
private:
        virtual dummy() = 0;
};
```

This ensures that only the `stack` functions can create `rep`s:

```
stack::rep* stack::create(int i)
{
        rep* p = new rep(i);    // fine: create is a member of stack
        // ...
        return p;
}

f()
{
        stack::rep s(10); // error: f() cannot access rep::rep(): private member
}
```

Naturally, the example of stack usage from section 6 works exactly as before.

8 Packages with Implicit Indirection

If we are not interested in inlining but prefer to minimize recompilation costs even when we don't have a smart dependency analyser, we can place the representation ‘‘elsewhere:’’

```
// stack interface, stack.h:

class stack_rep;

class stack {
public:
        typedef stack_rep* id;

        static id create(int);
        static void destroy(id);

        static void push(id,cat);
        static cat pop(id);
private:
        virtual dummy() = 0;
};
```

This scheme keeps implementation details not only inaccessible to users but also out of sight. With this definition (alone) a user cannot allocate `stack_rep`s. Unfortunately C++ does not allow you to define a class ‘‘elsewhere’’ *and* have its name local to another class. Consequently, the name `stack_rep` must be global.

The indirection (implied by the use of `id`) is implicit to the users of stacks and explicit in the implementation of stacks.

9 Simple Minded Types

One simple improvement would be to put the operations that create and destroy `stack_rep`s into class `stack_rep`. Actually, for a C++ programmer it would be natural to put all the operations into the `rep` and rename it ‘‘`stack`:’’

```
// stack interface, stack.h:

class stack {
        // actual representation of a stack object
public:
        typedef stack* id;

        static id create(int);
        static void destroy(id);

        static void initialize(id,int);
        static void cleanup(id);

        static void push(id,cat);
        static cat pop(id);
};
```

so that we can write:

```
#include "stack.h"

void f(int sz, cat kitty)
{
        stack s;
        stack::initialize(&s,sz);
        stack::push(&s,kitty);
        cat c2 = stack::pop(&s);
        stack::cleanup(&s);
}
```

The redundant use of the typedef ensures that our original program still compiles:

```
#include "stack.h"

void g(int sz, cat kitty)
{
        stack* p = stack::create(sz);
        stack::push(p,kitty);
        cat c2 = stack::pop(p);
        stack::destroy(p);
}
```

The likely difference between `f()` and `g()` is two memory management operations (a `new` in `create` and a `delete` in `destroy`).

10 Types

Now all we have to do is to make the functions non-static and give the constructor and destructor their proper names:

```
// stack interface, stack.h:

class stack {
        // actual representation of a stack object
public:
        stack(int size);
        ˜stack();

        void push(cat);
        cat pop();
};
```

We can now use the C++ member access notation:

```
#include "stack.h"

void f(int sz, cat kitty)
{
        stack s(sz);
        s.push(kitty);
        cat c2 = s.pop();
}
```

Here we rely on the implicit calls of the constructor and destructor to shorten the code.

11 Types with Implicit Indirection

If we want the ability to change the representation of a stack without forcing the recompilation of users of a stack we must reintroduce a representation class `rep` and let `stack` objects hold only pointers to `rep`s:

```
// stack interface, stack.h:

class stack {

        struct rep {
                // actual representation of a stack object
        };

        rep* p;
public:
        stack(int size);
        ~stack();

        void push(cat);
        cat pop();
};
```

The indirection is invisible to the user.

Naturally, to take advantage of this indirection to avoid re-compilation of user code after changes to the implementation we must avoid inline functions in `stack`. If our dependency analyser is dumb we might have to pace representation class `rep` ‘‘elsewhere’’ as was done in section 8 above.

12 Multiple Representations

In all of the examples above, the binding between the name used to specify the operation to be performed (e.g. `push`) and the function invoked were fixed at compile time. This is not necessary. The following examples show different ways to organize this binding. For example, we could have several kinds of stacks with a common user interface:

```
// stack interface, stack.h:

class stack {
public:
        virtual void push(cat) = 0;
        virtual cat pop() = 0;
};
```

Only pure virtual functions are supplied as part of the interface. This allows stacks to be used, but not created:

```
#include "stack.h"

void f(stack& s, cat kitty)
{
        s.push(kitty);
        cat c2 = s.pop();
}
```

Since no representation is specified in the stack interface, its users are totally insulated from implementation details.

We can now provide several distinct implementations of stacks. For example, we can provide a stack implemented using an array

```
// array stack interface, astack.h:

#include "stack.h"

class astack : public stack {
        // actual representation of a stack object
        // in this case an array
        // ...
public:
        astack(int size);
        ~astack();

        void push(cat);
        cat pop();
};
```

and elsewhere a stack implemented using a linked list:

```
// linked list stack interface, lstack.h:

#include "stack.h"

class lstack : public stack {
        // actual representation of a stack object
        // in this case a linked list
        // ...
public:
        lstack(int size);
        ~lstack();

        void push(cat);
        cat pop();
};
```

We can now create and use stacks:

```
#include "astack.h"
#include "lstack.h"

void g()
{
        lstack s1(100);
        astack s2(100);

        cat ginger;
        cat blackie;

        f(s1,ginger);
        f(s2,snowball);
}
```

13 Changing Operations

Occasionally, it is necessary or simply convenient to replace a function binding while a program is running. For example, one might want to replace `lstack::push()` and `lstack::pop()` with new and improved versions without terminating and restarting the program. This is fairly easily achieved by taking advantage of the fact that calls of virtual functions are indirect through some sort of table of virtual functions (often called the `vtbl`).

The only portable way of doing this requires cooperation from the `lstack` class; it must have a constructor that does no work except for setting up the `vtbl` that all constructors do implicitly:

```
class noop {};

lstack::lstack(noop) {} // make an uninitialized lstack
```

Fortunately such a constructor can be added to the program source text without requiring recompilation that might affect the running program that we are trying to update (enhance and repair).

Using this extra constructor we define a new class which is identical to `lstack` except for the redefined operations:

```
// modified linked list stack interface, llstack.h:

#include "stack.h"

class llstack : public lstack {
public:
        llstack(int size) : lstack(size) {}
        llstack(noop x) : lstack(x) {}

        void push(cat);
        cat pop();
};
```

Given an `lstack` object we can now update its pointer to its table of virtual functions (its `vtbl`) thus ensuring that future operations on the object will use the `llstack` variants of `push()` and `pop()`:

```
#include "llstack.h"

void g(lstack& s)
{
        noop xx;
        new(&s) llstack(xx);    // turns s into an llstack!
}
```

Naturally, we must rely on some form of dynamic linking to make it possible to add `g()` to a running program. Most systems provide some such facility.

This use of `operator new` assumes that

```
// place an object at address 'p':
void* operator new(size_t,void* p) { return p; }
```

has been defined and relies on the `llstack::llstack(noop)` constructor for suppressing (re)initialization of the data of `s` and for updating `s`'s `vtbl`.

This trick allows us to change the `vtbl` for particular objects without relying on specific properties of an implementation (that is, portably). There is no language protection against misuse of this trick.

Changing the `vtbl` for every object of class `lstack` in one operation, that is, changing the contents of the `vtbl` for class `lstack` rather than simply changing pointers to `vtbl`s in the individual objects, cannot be done portably. However, since that operation will be messy and non-portable I will not give an example of it.

14 Changing Representations

A more interesting – and probably more realistic – challenge is to replace both the representation and the operations for an object at run time. For example, convert a stack from an array representation to a linked list representation at run time without affecting its users. To ensure that the cutover from one representation to another can be done by a single assignment we reintroduce the `rep` type and make `push()` and `pop()` simple forwarding functions to operations on the `rep`:

```
// stack interface, stack.h:

class stack {
        rep* p;
public:
        stack(int size);
        ˜stack();

        rep* get_rep() { return p; }
        void put_rep(rep* q) { p = q; }

        void push(cat c) { p->push(c); }
        cat pop() { return p->pop(); }

        int size() { return p->size(); }
};
```

The idea is to have operations that convert between the different representations and then let them use `get_rep()` and `put_rep()` to update the pointer to the representation. In a real system `get_rep()` and `put_rep()` would most likely not be publicly accessible functions.

First we define `rep` exactly as we did `stack` before :

```
// stack interface, rep.h:

class rep {
public:
        virtual void push(cat) = 0;
        virtual cat pop() = 0;
        virtual int size() = 0;
};
```

and use it as a base for the different implementations:

```
// array stack interface, astack.h:

#include "rep.h"

class astack : public rep {
        // actual representation of a stack object
        // in this case an array
        // ...
public:
        astack(int size);
        ˜astack();

        void push(cat);
        cat pop();

        int size();
};
```

Elsewhere we can define a stack implemented using a linked list:

```
// linked list stack interface, lstack.h:

#include "rep.h"

class lstack : public rep {
        // actual representation of a stack object
        // in this case a linked list
        // ...
public:
        lstack(int size);
        ~lstack();

        void push(cat);
        cat pop();

        int size();
};
```

Now we can convert the representation of a `stack` from a `astack*` to a `lstack*` by changing `stack::p` using `stack::get_rep()` and `stack::put_rep()` and (in general) also copying the elements:

```
rep* convert_from_a_to_l(stack& s)
{
        rep* rp = s.get_rep();
        astack* ap = new astack(s.size());
        // copy s's elements to *ap
        s.put_rep(ap);
        return rp;
}
```

In other words, we solve the problem by introducing yet another indirection†. This assumes that the size argument from the original constructor has been stored away somewhere so that it can be used as the argument to the new `astack`. In a real system we would also need to check that `s` really had an appropriate representation and would most likely also have to provide some further consistency checking and interlocking. However, the fundamental idea is illustrated.

15 Changing the Set of Operations

Finally, I will show a version of the stack example that is somewhat un-C++-like in that it dispenses with static type checking of the operations. The idea is to completely disconnect the users and the implementers and simulate a dynamically type-checked language. Overreliance on such techniques can make systems slow and messy. However, the flexibility offered can be important in localized contexts where the inevitable problems caused by lack of formal structure and of run-time checking can be contained.

In this and the following examples a bit of scaffolding is needed to make the programming techniques convenient. This makes the toy examples somewhat longer to define but does not, in fact, noticeably increase the size of a realistic system relying on them. The most primitive building block for this example is lists of (operation_identifier,function) pairs:

† The first law of computer science: Every problem is solved by yet another indirection.

```
typedef cat (*PcatF)(void*,cat);

struct oper_link {
        oper_link* next;
        int oper;
        PcatF fct;

        oper_link(int oo, PcatF ff, oper_link* nn)
                : oper(oo), fct(ff), next(nn) {}
};
```

Using `oper_link`s we can specify a class `cat_object` that allows us to invoke an unspecified set of functions on an unspecified representation:

```
class cat_object {
        void* p;        // pointer to representation
        oper_link* oper_table; // list of operations
public:
        cat_object(oper_link* tbl = 0, void* rep = 0)
                : oper_table(tbl), p(rep) { }

        cat operator()(int oper, cat arg = 0);

        void add_oper(int, PcatF);
        void remove_oper(int);
};
```

Default arguments are user to spare the programmer the bother of specifying arguments where they are not in fact necessary for a particular operation. This technique is elaborated in section 16 below.

The application operator simply looks for an operation in its list and executes it (if found):

```
cat cat_object::operator()(int oper, cat arg)
{
        for (oper_link* pp = oper_table; pp; pp = pp->next)
                if (oper == pp->oper) return pp->fct(p,arg);
        return bad_cat;
}
```

If the operation fails the distinguished object `bad_cat` is returned.

Given this feeble framework we can now build a stack:

```
// stack interface, stack.h:
enum stack_oper { stack_destroy = 99, stack_push, stack_pop };

cat_object* make_stack(cat_object* = 0);
```

As usual, the implementation of the stack is left as an exercise to the reader. However, here is a hint:

```
#include "stack.h"

struct rep {
        // ...
        void push(cat);
        cat pop();
};
```

```
static cat stack_push_fct(void* p, cat c)
{
        ((rep*) p)->push(c);
        return bad_cat;
}

static cat stack_pop_fct(void* p, cat)
{
        return ((rep*) p)->pop();
}

static cat stack_destroy_fct(void* p, cat) { /* ... */ }

cat_object* make_stack(cat_object* p)
{
        if (p == 0) p = new cat_object(0,new rep);  // get a clean object
        p->add_oper(stack_push,&stack_push_fct);       // and make it
        p->add_oper(stack_pop,&stack_pop_fct);         // behave
        p->add_oper(stack_destroy,&stack_destroy_fct); // like a stack
        return p;
}
```

We can now create and use stacks:

```
#include "stack.h"

void g(cat kitty)
{
        cat_object& s = *make_stack();
        s(stack_push,kitty);
        cat c2 = s(stack_pop);
        s(stack_destroy);
}
```

We can add operations to a stack at run time. For example, we might want an operation for peeking at the top `cat` without popping:

```
enum { stack_peek = stack_pop+1 };
cat stack_peek_fct(void*,cat);

void h(cat_object& s)
{
        s.add_oper(stack_peek,&stack_peek_fct);

        cat top = s(stack_peek);
}
```

Note that the operations on these stacks do not involve addresses; they can be transmitted between address spaces without special effort. This technique for invoking operations is often called message passing.

16 Tailoring

The dynamically typed stack above could be improved in many ways. For example, the operation lookup could be made faster, an inheritance mechanism could be added, the naming of operations could be made more general and safer, the method for passing arguments could be made more general and safer, etc. Here I will just demonstrate two techniques of more general interest.

Firstly, the message passing mechanism can be hidden behind an interface that provides notational convenience and type safety for the key stack operations. The point is that combinations of the techniques described in this paper can be used to handle more delicate cases. In particular, any data abstraction technique can be used to hide ugliness in an implementation:

```
// improved stack interface, Stack.h:

#include "stack.h"
class stack : public cat_object {
public:
        stack() { make_stack(this); }
        ˜stack() { (*this)(stack_destroy); }
        void push(cat c) { (*this)(stack_push,c); }
        cat pop() { return (*this)(stack_pop); }
};
```

This allows us to write

```
#include "Stack.h"

void g(int sz, cat kitty)
{
        stack s;

        // compile time checked uses:

        s.push(kitty);
        cat c2 = s.pop();

        // run time checked uses:

        s.add_oper(stack_peek,&stack_peek_fct);
        cat top = s(stack_peek);
}
```

This uses the unchecked ‘‘message passing’’ notation only where needed.

Secondly, in the dynamically typed stack example I dodged the issue of argument types and argument type checking by simply providing a fixed number of arguments of fixed type. Similarly, I simply had all operations return a `cat`. That is surprisingly often a viable choice for the sort of interfaces for which you actually need ‘‘message passing.’’ A larger class of problems can be handled by allowing arguments of a fixed number of types. For example:

```
class argument {
        enum type_indicator { non_arg, int_arg, ptr_arg, string_arg, cat_arg };
        type_indicator t;
        union {
                int i;
                void* p;
                char* s;
                cat c;
        };
public:
        argument(noop) :t(non_arg) { }
        argument(int ii) : i(ii), t(int_arg) { }
        argument(void* pp) : p(pp), t(ptr_arg) { }
        argument(char* ss) : s(ss), t(string_arg) { }
        argument(cat cc) : c(cc), t(cat_arg) { }

        operator int() { return t==int_arg ? i : -1; }
        operator void*() { return t==ptr_arg ? p : 0; }
        operator char*() { return t==string_arg ? s : 0; }
        operator cat() { return t==cat_arg ? c : bad_cat; }
};
```

This assumes that `cat` is declared so that `==` and `?:` can be applied.

The error handling for the conversion operators could be improved, but even as it stands this would allow the cat_object class to be written:

```
typedef argument (*ParIumentF)(void*,argument);

struct oper_link {
        oper_link* next;
        int oper;
        ParIumentF fct;

        oper_link(int oo, PcatF ff, oper_link* nn)
                : oper(oo), fct(ff), next(nn) {}
};
```

and can be used like this:

```
#include "stack.h"

void g(cat kitty)
{
        argument_object& s = *make_stack();
        s(stack_push,kitty);    // converts ‘kitty’ to argument
        cat c2 = s(stack_pop);  // converts argument to cat
        s(stack_destroy);
}
```

The object `no_arg` is passed (by default) to indicate that no argument was specified by the user.

You may have noticed that the size argument to the stack create operation disappeared when we moved to the message passing style. The reason was to avoid the complication of dealing with arguments of different types until we had the mechanism to do so. Putting that argument back in is now left as an exercise to the reader.

Adding ‘‘list of `argument`s’’ to the list of acceptable `argument` types is left as yet another exercise to the reader. Allowing such lists again increases the range of applications for which the message passing technique is acceptable.

17 Dynamic Classes

Note that the concept of a class was almost lost in the message passing examples above. Each object had its own list of acceptable operations that could be modified independently of the lists of any other object. This could be seen as too flexible for many applications and also not sufficiently amenable to space and time optimizations. These problems can be alleviated by re-introducing the notion of a class as an object containing information common to a set of objects. Here we will only represent the set of acceptable operations on an object of one of these ‘‘dynamic classes.’’

```
class argument_class_rep {
public:
        oper_link* oper_table; // list of operations

        void add_oper(int, ParIumentF);
        void remove_oper(int);
};
```

The reader can easily extend this notion, though.

The objects looks much as they did before. The only change is that an indirection through an object representing a class has been introduced on the path to the table of operations:

```
class argument_object {
        void* p;        // pointer to representation
        argument_class_rep* crep; // class
public:
        argument_object(argument_class_rep* cc, void* rep = 0)
                : crep(cc), p(rep) { }

        argument operator()(int oper, argument arg = no_arg);
};
```

Given this we can create an object representing stacks and provide an operation for gaining access to it:

```
static argument_class_rep stack_class;  // the object representing stacks

argument_class_rep* get_stack_class()
{
        if (stack_class->oper_table == 0) {
                stack_class.add_oper(stack_push,&stack_push_fct);
                stack_class.add_oper(stack_pop,&stack_pop_fct);
                stack_class.add_oper(stack_destroy,&stack_destroy_fct);
        }
        return &stack_class;
}
```

Finally, we can provide a function for making stacks

```
argument_object* make_stack()
{
        return new argument_object(get_stack_class());
}
```

and use it exactly as the previous versions:

```
void g(cat kitty)
{
        argument_object& s = *make_stack();
        s(stack_push,kitty);    // converts ‘kitty’ to argument
        cat c2 = s(stack_pop);  // converts argument to cat
        s(stack_destroy);
}
```

This version, however, does not allow operations for an individual object to change. It does however, open the possibility of trivially changing the operations on all objects of a dynamic class.

18 Conclusions

C++ covers the spectrum of data hiding techniques from C (files as modules) through Modula-2 (modules) and Ada (packages) to C++ (classes), and beyond. Given a free choice a C++ programmer would naturally choose one of the class-based strongly-typed techniques (sections 10, 11, or 12), but the other techniques can occasionally be useful.

19 Acknowledgements

Andrew Koenig, Brian Kernighan, and Doug McIlroy lent ears and blackboard to some of these little puzzle programs. Jim Coplien suggested the extension of the range of examples to include function binding examples. The nine-plus cat lives in this paper are dedicated to Dave McQueen who once in desperation proposed the death penalty for presenting ‘‘yet another stack example.’’ Also thanks to Andy for giving me a practical demonstration of the difficulty of stacking cats.