Template functions and classes

In our array implementation of stacks we had two classes: int_stack and char_stack. This seems like potential error prone work and duplication to create a new class of stack for every class of object we might want to put on a stack. Templates were invented for just such situations.

```
#include <assert.h>
#include <new.h>

template<class T>
  class Array_stack {
   int sz;
   int top;
   T* stack;
  public:
   Array_stack(int Nitems=100)
       { sz = Nitems; stack = new T[sz]; top = sz; }
   ~Array_stack() { delete[] stack; }
   void push(T x) { assert( !isfull() );
          stack[--top] = x; }
   T pop() { assert( !isempty() );
          return( stack[top++] ); }
   bool isempty() { return( top == sz ); }
   bool isfull() { return( top == 0 ); }
  };
```

```
#include <iostream.h>
#include "mydefs1d.h"

void main()
{
   Array_stack<int> s1(500);
   Array_stack<char> s3;
   int i;

   for( i = 0; i < 20; i++ ) {
      s1.push(i);
   }
   for( i = 0; i < 20; i++ ) {
      cout << s1.pop() << ' ';
   }
   cout << endl;

   for( i = 0; i < 20; i++ ) {
      s3.push('a' + i);
   }
   for( i = 0; i < 20; i++ ) {
      cout << s3.pop() << ' ';
   }
   cout << endl;
}
```

Object Oriented Programming support:

inheritance and virtual functions.

One of the promises made when we started C++ was that you could put off implementation choices to the very end. We're now going to declare a pure virtual class called vstack, two derivations (arrays and linked lists), a main program that declares stacks of both types and passes them one at a time to a function that has no idea which type of stack it is getting and indeed handles an Array_stack at the first call and a Linked_stack on the second.

For the virtual base class vstack (Vstack.h)

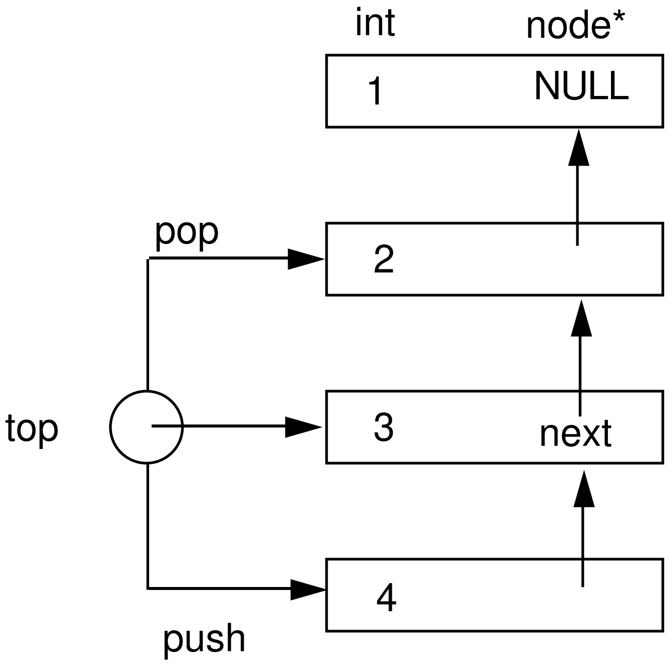
```
template<class T>
  class stack {
  public:
   virtual void push(T) = 0;    // pure virtual function
   virtual T pop() = 0;         // pure virtual function
   virtual bool isempty() = 0; // pure virtual funct.
   virtual bool isfull() = 0;  // pure virtual funct.
  };
```

For a derivation of stacks using arrays (Arraystack.h)

```
#include <assert.h>
template<class T>
  class Array_stack : public stack<T> {
   int sz;
   int top;
   T* astack;
  public:
   Array_stack(int Nitems = 100)
      { sz = Nitems; astack = new T[sz]; top = sz; }
   ~Array_stack() { delete[] astack; }
   void push(T x) { assert( !isfull() );
          astack[--top] = x; }
   T pop() { assert( !isempty() );
          return( astack[top++] ); }
   bool isempty() { return( top == sz ); }
   bool isfull() { return( top == 0 ); }
  };
```

A derivation of stacks using linked lists (linkedstack.h)

```
#include <assert.h>
#include <new.h>
template<class T>
  class Linked_stack : public stack<T> {
   struct node {
       T data;
       node* next;
       node(T d, node* n) {data = d; next = n; }
   };
   node* top;
  public:
   Linked_stack( int Nitems = 100 ) { top = NULL; }
   void push(T x) { top = new node(x, top); }
   T pop() {
       assert( !isempty() );
       T t = top -> data;
       node* oldtop = top;
       top = top -> next;
       delete oldtop;
       return t;
   }
   bool isempty() { return( top == NULL ); }
   bool isfull() { return( FALSE ); }
};
```



Linked Stack showing effect of push() and pop() operations

The main program (stackmain.C)

```
#include "Vstack.h"
#include "Linkedstack.h"
#include "Arraystack.h"
extern void reverse( stack<int> &s, int n );

void main()
{
   array_stack<int> s1(500);
   linked_stack<int> s2;  // or s2(137); but not s2();

   reverse(s1, 20);
   reverse(s2, 20);
}
```

For the function that uses vstack (reverse.C)

```
#include <iostream.h>
#include "Vstack.h"
// Doesn't know about Arraystack or linkedstack
void reverse( stack<int> &s, int n )
{
int i;

   for( i = 0; i < n; i++ ) {
      s.push(i);
   }
   for( i = 0; i < n; i++ ) {
      cout << s.pop() << ' ';
   }
   cout << endl;
}
```

A program that uses reverse

```
#include "Vstack.h"
#include "Linkedstack.h"
#include "Arraystack.h"
extern void reverse( stack<int> &s, int n );

void main()
{
    Array_stack<int>  s1(500);
    Linked_stack<int> s2;   // or s2(137); but not s2();
    Array_stack<char>  s3;
    Linked_stack<char> s4(50);

    reverse (s1, 20);
    reverse (s2, 20);
//  reverse (s3, 20);   //but reverse only knows about integers
//  reverse (s4, 20);   //need more generality, perhaps?
}
```

Public vs. protected vs. private access

members of a class can be:

* **`private`**, the name can only be used by member functions and friends of the containing class

* **`protected`**, the name can be used by the above and by member functions and friends of derived classes

* **`public`**, the name can be used by any function

Use these to control visibility in the interface.

Designing with objects

This section has largely been paraphrased from [Enstlinger, 1990] and [Stroustrup, 1992].

Rules of thumb

* Don't expect to know everything before you start
* Get something working as soon as possible
* Change the design as you go
* Try to make your mistakes as early as possible

Design steps

* Find the concepts (classes) and their fundamental relationships
* Refine the classes by specifying the operations on them
* Classify these operations. Consider construction, copying, and destruction
* Consider minimalism, completeness, and convenience
* Refine the classes by specifying their dependencies on other classes
* Consider inheritance
* Use dependencies
* Specify the interfaces for the classes
* Separate functions into private, public and protected operations
* Specify the exact type of the operations on the classes

Discovering classes

Discovering classes

* Look for external factors; interactions between objects (classes) and the world outside them. These help you to find the objects themselves. Look for boundaries in the real world. These are often reflected in your objects. Look for initialization and cleanup. These belong in constructors and destructors.

* Look for things that are duplicated. Are there are two or more of some object? Might you someday want to add another one of these objects? Common interfaces belong inside an abstract base type.

* Look for the smallest common denominator in the system. For a text editor, you sometimes want to edit characters, words, lines, blocks, files. This gives you a whole set of related objects and relationships.

* Will this class work in other situations? Is it general? Which things stay the same? Which differ?

* Look at the data. Group data that seems to belong together. Is this an object? Should this data be hidden or at least not be carelessly modified? Put it inside an object

* Write a main program using the objects you have so far. Do you have everything you need?

More on const

const is tricky enough that it deserves some study.

Unlike C, in C++ `const` really is constant The following declarations work in C++ but not in C:

```
const int maxn = 1000;     //#define maxn 1000
int a[maxn + 1];
```

const pointer vs pointer to const vs const pointer to const

When pointers and const are involved in the same declaration, what does it mean? Does it mean the pointer doesn't change (const pointer)? Does it mean the thing the pointer points at doesn't change (pointer to const)? Or does it mean neither the pointer nor the thing it points at change? Not so clear, but luckily, it is possible to express all three concepts in C++:

```
int i = 1, j = 3;
int* const p = &i;       // const pointer
const int* q = &i;       // pointer to const
const int* const r = &i;   // const pointer to const

*p = 5;        // okay
p = &j;            // error
*q = 5;            // error
q = &j;        // okay
*r = 5;            // error
r = &j;            // error
```

const reference variables

You can have a reference to an object, and tell the compiler that you don't want to use this reference to change the variable.

```
int i;
const int j = 1;
const int &r = i;
const int &s = j;

i = 2;    // okay
j = 2;       // error
r = 2;       // error
s = 2;       // error
```

const reference variables and arguments to functions

Rather than have a large object pushed onto the stack, you can pass a reference to the object. If you want to indicate that the function won't be making any changes to the object, you can pass it by const reference. This is probably the most common use of references, since it acts like an "efficient call by value:"

```
void print_large_array( const int &ar[], int n )
```

or

```
void print_large_array( const int* &ar, int n )
```

The above two are equivalent.

const references to class data

A class acts like a struct that is passed implicitly to all its member functions. You can indicate that a member function does not modify the data area of its class by using const, giving the same "efficient call by value" for the implicit class argument that the previous section gave to explicit arguments:

```
class sample {
    int n; public:
    void set( int i ) { n = i; }     // Allowed to change n
    int get() const { return n }     // Cannot change n
}
```

returning const

You can indicate that the return value is in some sense constant:

```
const int n = 1;
const v() { return n; }
const int w() { return n; }
const int* x() { return &n; }
int* const y() { return &n; }
                // error return of non-`const* '
                // pointer from `const* '

const int* const z() { return &n; }

int main() {
    int i = 1;
    const int* q = &i;      // pointer to const
    int* s;                 // pointer

    i = v();  // okay
    i = w();  // okay
    q = x();  // okay
    s = x();     // error
      // assignment of non-`const* ' pointer from `const* '
    q = y();  // okay
    s = y();  // okay
    q = z();  // okay
    s = z();     // error
      // assignment of non-`const* ' pointer from `const* '
}
```

Linkage of const variables

Unlike C, const variables declared at the outermost level have internal linkage by default:

```
#include <iostream.h>

const int zero = 0;      // Internal in C++, External in C
extern const int one = 1;   // External in both C++ and C
static const int two = 2;   // Internal in both C++ and C
```

References to pointers

References to pointers can be used in place of "pointers to pointers":

```
#include <iostream.h>

void setstr( char* &var, char* str )
{ var = str; }

int main() {
    char* s;
    setstr( s, "hello" );
    cout << "s is '" << s << "'" << endl;
}
```

The output is:

s is 'hello'

Functions that return Lvalues

Lvalues are "things that can occur on the left-hand side of the assignment operator." References allow functions to return lvalues:

```
int a[10];

int &f( int i ) {
   return a[i - 1990];
}

int main() {
   f(1994) = 7;     // Sets a[4] = 7
}
```

This is also useful for overloading operators; operator= for type T would be of type T&.

Copy constructors

C++ will call the copy constructor for class T when:

* An instance of class T is created from an existing instance of class T. For example:
* A function is called and an argument of class T is passed by value:
* An unnamed temporary variable is created to hold the return value of a function returning an instance of class T

Examples:

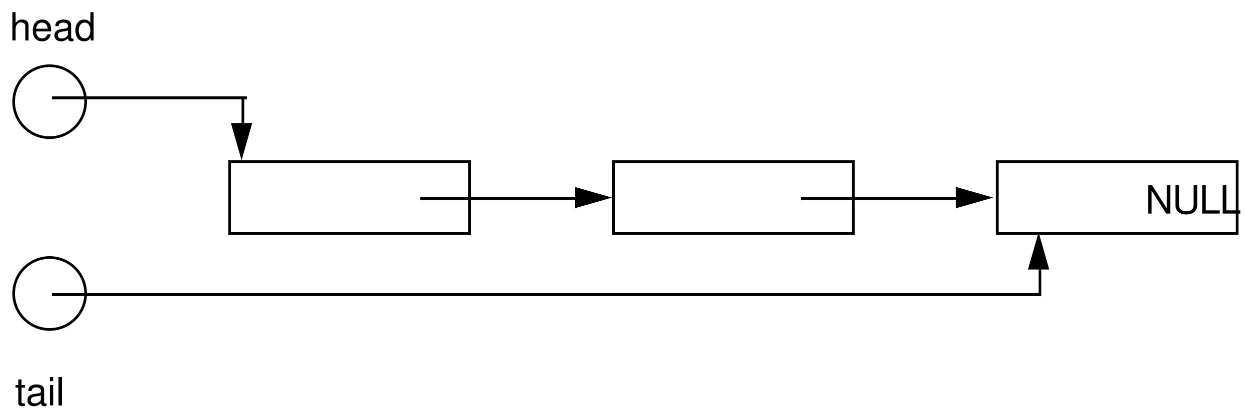
```
class T {
    char* p;
public:
    T() {};
    T( char* s ) { p = strdup( s ); }
    ~T() { delete p; }
};

T f( T c ) { return c; }

int main() {
    T* a = new T( "hello" );

    T b;

    T c = *a;
        // copy constructor will be called for c
    b = f( *a );
        // copy constructor will be called twice,
        // once to pass a to f, once to set b
    delete a;
        // The string that b and c point to is gone!
}
```

C++ will supply a copy constructor for each class that recursively copies member data from the source instance to the destination. However, if any of the class members are pointers, then this will give you two pointers to the same memory location. The copy constructor should be of type:

```
T( const T & )
```

That is, it is a constructor that can read but not modify (that's why the `const` is there) another instance of T. C++ passes the instance by reference for efficiency. In our example:

```
class T {
   char* p;
public:
   T() {};
   T( char* s ) { p = strdup( s ); }
   ~T() { delete p; }
   T( const T& u ) { p = strdup( u.p ); }
};
```



Operations:

AddToHead
AddToTail
DeleteFromHead
DeleteFromTail
QueueEmpty