Virtual Functions

Class derivation is even more valuable when combined with virtual functions -- functions declared in a base class but implemented differently in each derived class. Consider the `Shape` class: It will need a grow function so that every `Shape` can increase its size.

```
class Shape {
public:
   void grow();
   ....
};
```

But the grow function must be different for each class derived from Shape. Thus we make grow a virtual function

```
class Shape {
public:
   virtual void grow ();
   ........
};
```

The Circle, Square and Triangle classes will now provide their own custom versions of grow. The circle version might look like:

```
class Circle: public Shape {
public:
   void grow () { radius++; }
   ......
private:
   int radius;     // radius of circle
   ....
}
```

Much of the time a virtual function behaves like an ordinary member function. If `c` is a `Circle` object, the call `c.grow()` increases the radius of `c`. When `grow()` is invoked through a pointer to a base class, the virtual function comes into play, using the correct grow function depending on the kind of object pointed to.

If `p` is a pointer to a `Shape` object, then since `Circle` and `Square` are derived from the `Shape` class, `p` could be pointing to either one of these classes. If **`p`** points to a **`Circle`** object then **`Circle::grow`** will be used, and conversely `Square::grow` if `p` points to a `Square` which has to be expanded.

```
Shape* p;
Circle c;
Square s;

p = &c;            // p points to a Circle
p->grow();         //  calls  Circle::grow
p = &s;            // p points to a Square
p->grow();         //  calls  Square::grow
```

Notice the use of -> to call `grow`. Calling a member function from a pointer to an object requires **->** instead of the dot operator.

Many function calls--even overloaded ones--can be resolved by the compiler. Others can't and require dynamic binding by the loader (just how this is done is left to the operating systems course), but the need is clear. Consider:

```
if (....)
   p = &c;
else
   p = &s;
p->grow();         // calls either Circle::grow
                   //          or  Square::grow
```

There is no way that the compiler can know how p will be set during execution.

Templates

Finally we come to templates which are "patterns" from which classes can be created. A template looks much like an ordinary class, except that part of the class definition is left unspecified. By this means the class is more general and easier to reuse.

Consider the `Stack` type of from an earlier lecture. If we translate this type into a C++ class, we might end up with the following definition:

```
class Stack {
public:
    void make_empty();
    int is_empty();
    void push (int);
    int pop ();
    ....
};
```

But this `Stack` can only store integers. If we later need a stack that stores some other type of data (double values, perhaps, or a structure of some kind, or a pointer variable), then we would have to duplicate the code of `Stack` to form `Istack` and `Double_Stack` and so on. A tedious and error-prone process at best. C++ provides a better idea.

```
template <class T>
class Stack {
public:
    void make_empty();
    int is_empty();
    void push(T);
    T pop();
    .....
};
```

The new, more general, template class looks much the same, but with the addition of

```
template <class T>
```
which indicates that Stack is a template that won't be complete until the missing class `T` is filled in. `T` is a dummy name (a "template argument"), since any symbol will do. Notice that both the `push` and the `pop` operations need to know about `T`.

```
template <class T>
void Stack<T> :: push(T x)
{
    .....
}
```

Notwithstanding the `<class T>` notation, Stack does not have to be a class, any C++ type will do. For example,

```
Stack<int> Istack;              // stack of integers
Stack<double> Dstack;           // stack of doubles
Stack<char> Cstack;         // stack of char values
Stack<Map*> Mstack;         // stack of map pointers
```

And for the push operation we would need something like:

```
Istack.push(10);                // put a 10 onto a int stack
Dstack.push(3.4);       // put 3.4 onto a double stack
Cstack.push('x');       // put an 'x' on a char stack
```

Thus from this you can see that C++ is quite a big and powerful language, which should lead to smaller, easier to debug and maintain programs! You now have more than enough to handle Assignment 3, though I am sure that you will find Allen Supynuk's online notes most helpful, since there you will find some worked examples. Missing from these lectures is a discussion of exception handling. This is not done well under g++ so don't try to use it (and I will avoid making it part of this course!).

Debugging and Testing

There are naturally many other tricky details for us to cover, but the framework for the discussion has been laid. It is easy sailing from now on. Even so you may wish to hammer out labs 7, 8 and 9 this week and next, and perhaps strengthen your experience with **gdb**. Here your lab TA should be able to provide good advice on debugging practice.

Personally I still remember an earlier school of thought which builds very small working programs of the essential elements that illustrate the difficult parts, with the support of print statements.

Only after the ideas are straight are they embedded into the real program.

For example in Assignment 2 my Map started with M=10 and N=6. Then I entered about a dozen words and printed out the 3 most frequent. Map traverse was used to call print statements to show exactly what was in each entry.

Other prints showed what hash number was generated, how conflicts were resolved and when extensions or transitions to the next map were made.

These print statements may be preserved in the final version, but put under the control of compile-time debug flags (or better yet, command line parameters).

A good debugger will reduce the need for print statements, but you are still better off with a very small initial set of test data for your program, so that you can run through everything by hand and be sure you have caught all the careless errors.

I made plenty: using **sizeof** when I meant **strlen** (it was a while before I noted that my long words were truncated to 8 bytes!). Also, only with a small example can you be confident that expansions are occurring at exactly the right time, for example.

Despite this apparent care I allocated space for strlen(s) instead of 1+strlen(s). Strangely this error caused me no grief at all!! and remains today as an undocumented bug in the program!

There is good reason why it caused no problem. Do you have any thoughts about why things might work?

Returning to Allen Supynuk's C++ and OOP Notes

Why Abstract Data Types (ADTs)?

Abstract Data Types (ADTs)are a higher level way of looking at programming.
Prototypical ADTs include lists, stacks, and sets.

Formal definition of ADT

An abstract data type (ADT) is a mathematical model with a collection of operations defined on it. [Aho, 1983]

Stacks

Suppose your program needs to keep track of a stack of things. You have at least two choices for ways of implementing stacks:

* As an array
* As a linked list

Each method has it's advantages:

* Arrays are very efficient on storage, but are difficult to extend, easy to implement
* Linked lists have the overhead of a pointer for each item, are easy to extend, but are a bit harder to implement

Depending on the needs of your program, you will pick one of these methods.
Unfortunately, you may not be able to determine which is the best method to use until the middle or end of a project.

The Stack ADT

Ignoring how you implement lists, what sorts of things do you want to do?

```
*   void init (stack S, unsigned int Nitems)

*   void push (stack S, item x)

*   item pop (stack S)

*   bool isempty (stack S)

*   bool isfull (stack S)
```

In the best of all possible worlds, you would like to be able to declare things to be of type STACK and ITEM, then choose at the last possible moment (perhaps when you are linking your final program together) which implementation to use.

This is precisely what ADTs are about.

For now, let's assume all items in the stack are integers. (We'll make our ADT more general
when we learn about templates.)

Declaring the public interface

See ~c201/Notes/Supynuk/C++examples/mydefs1.h

```
class Istack {
    int sz;        // stack size
    int top;       // location of top element
    int* stack;    // pointer to origin of stack
public:
    // Note that the stack is implicit
    void init (int Nitems = 100);
    void push (int x);
    int pop ();
    bool isempty();
    bool isfull();       // prototypes
};
```

Note

* classes look a lot like structs. In fact, C++ treats structs as a class whose members are by default public

* by default, members of classes are private. We could say private before int sz, if we wanted to make this explicit

An array implementation of Istack

See ~c201/Notes/Supynuk/C++examples/stack1.C

```
#include "mydefs1.h"
#include <assert.h>

// Array implementation of stacks.
// Grow from high index towards 0.

void Istack :: init (int Nitems = 100) {
    sz = Nitems;
    stack = new int[sz];
    top = sz;         // Stack initially empty
}
void Istack :: push (int x) {
    assert (!isfull());
    stack[--top] = x;
}
int Istack :: pop() {
    assert(!isempty());
    return( stack[top++] );
}
bool Istack :: isempty() {
    return( top == sz );
}
bool Istack :: isfull() {
    return( top == 0 );
}
```

Notes

* assert(exp) does nothing if exp is false, otherwise it prints a run-time error, See King Chapter 24.

* We have not defined a destructor.

The main thing about Istack

See ~c201/Notes/Supynuk/C++examples/stackmain1.C

```
#include <iostream.h>
#include "mydefs2.h"

main()
{
    Istack s1, s2;
    int i;

    s1.init(500);
    s2.init();          // Default, 100 cells

    for ( i = 0; i < 20; i++ ) {
        s1.push(i);
    }               // try putting 501 elements!

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

    s2.push(10);        // Push a 10 onto s2
    i = s2.pop();       // Take it off into i
        // try s2.pop();   here
}
```

Note:

* It is short!
* Tested by pushing 501 things onto s1 and by adding another s2.pop() before exiting
* The (correct) output is:

19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

* If we forget to call init, we are in big trouble
* We haven't provided any way to get rid of a stack
  we don't have a destructor