Features of a Solid C++ Class

In all cases:

   Explicit default constructor
       Guarantees that every declared instance of the class will be initialized in some controlled manner.

If objects of the class contain pointers to dynamically-allocated storage:

   Explicit destructor
       Prevents memory waste.

   Copy constructor
       Implicitly used when copying an object during parameter passing or initialization. Prevents destructor aliasing problem.

   Assignment operator
       Implicitly used when an object is assigned to another. Prevents destructor aliasing problem.
Example – Integer Stack Class Interface

```cpp
// Stack.h - conditional compilation directives omitted to save space
class Stack {
private:
    int Capacity;  // current stack array size
    int Top;       // first available index in stack array
    int* Stk;     // stack array (allocated dynamically)
public:
    Stack(int InitSize);  // construct new stack with Capacity InitSize
    bool Push(int toInsert);  // push toInsert on top of stack, increasing
                              //    stack array dimension if necessary
    int Pop();            // remove and return element at top of stack
    int Peek() const;    // return copy of element at top of stack
    bool isEmpty() const; // indicate whether stack is currently empty
    bool isFull() const;  // indicate whether stack is currently full
    ~Stack();          // deallocate stack array
};
```

- specifies the data members and function members that all Stack objects will have
- imposes access restrictions via public and private sections
- separates user interface from implementation
An explicit destructor is needed since a Stack object contains a pointer to memory that is allocated dynamically on the system heap. If the implementation does not provide a destructor, the default destructor will NOT deallocate that memory.
Stack::Push()

```cpp
// . . . Stack.cpp continued

bool Stack::Push(int toInsert) {
    if (Top == Capacity) {
        int* tmpStk = new(nothrow) int[2*Capacity];
        if (tmpStk == NULL)
            return false;
        for (int Idx = 0; Idx < Capacity; Idx++) {
            tmpStk[Idx] = Stk[Idx];
        }
        delete [] Stk;
        Stk = tmpStk;
        Capacity = 2*Capacity;
    }
    Stk[Top] = toInsert;
    Top++;
    return true;
}
```

If the Stack array is full, we attempt to allocate a larger array. If that succeeds, we copy the contents of the old Stack array to the new one, delete the old one, and then retarget the old Stack array pointer.

As far as user can tell, underlying structure could be a linked list instead of an array.
Stack::Pop() and Stack::Peek()

```cpp
int Stack::Pop() {
    if ( (Top > 0) && (Top < Capacity) ) {
        Top--;
        return Stk[Top];
    }
    return 0;
}

int Stack::Peek() const {
    if ( (Top > 0) && (Top < Capacity) )
        return Stk[Top-1];
    return 0;
}
```

If the test Top > 0 were omitted, a call to Pop() when the Stack was empty would result in an invalid array access.

As designed, Pop() and Peek() have no way to indicate failure due to an empty stack.

Use of keyword “const” in member function declaration specifies that function is not allowed to modify the value of any data member.
bool Stack::isEmpty() const {
    return (Top == 0);
}

bool Stack::isFull() const {
    if (Top < Capacity) return false;
    int* tmpInt = new(nothrow) int[2*Capacity];
    if (tmpInt == NULL) return true;
    delete[] tmpInt;
    return false;
}

isFull() tests to see if the Stack array is, in fact, at its capacity. If it is, isFull() then tests to see if it would be possible to increase the size of the Stack array in the usual manner.

There are a number of shortcomings in this Stack implementation. One is that a Stack object can hold only integer values. We could, of course, easily “clone” the code and modify it to hold doubles, or characters, or any other type (including user-defined).

A better approach would be to design the Stack class so that the implementation allowed the data values to be of any type at all. We will revisit that idea later (more than once).
Try this now...

Implement a **safe** Stack::Pop() function:

*If there’s nothing on the stack, Pop “somehow usefully” signals the caller that there’s an error.*
First revise the Stack class interface to incorporate an error recording mechanism:

```cpp
// Stack.h – conditional compilation directives omitted to save space
enum ErrorType { NO_ERROR, STACK_UNDERFLOW, STACK_OVERFLOW };  
class Stack {
private:
    int Capacity; // current stack array size
    int Top; // first available index in stack array
    int* Stk; // stack array (allocated dynamically)
    ErrorType errorType; // indicates last error detected
public:
    Stack(int InitSize); // construct new stack with Capacity InitSize
    bool Push(int toInsert); // push toInsert on top of stack, increasing
    // stack array dimension if necessary
    int Pop(); // remove and return element at top of stack
    int Peek() const; // return copy of element at top of stack
    bool isEmpty() const; // indicate whether stack is currently empty
    bool isFull() const; // indicate whether stack is currently full
    ErrorType getError() const; // return error state
    ~Stack(); // deallocate stack array
};
```
Revised Stack::Pop()

Second revise the Stack::Pop() implementation to record the error:

```cpp
int Stack::Pop() {
    if ( (Top > 0) && (Top < Capacity) ) {
        errorType = NO_ERROR;
        Top--;
        return Stk[Top];
    }
    errorType = STACK_UNDERFLOW;
    return 0;
}
```

Other changes:

- constructor should initialize `errorType` to `NO_ERROR`
- `Push()` should set `errorType` to `STACK_OVERFLOW` when Stack size cannot be increased (and could be `void` now)
- `Peek()` should also set `errorType` to `STACK_UNDERFLOW` if Stack is empty (and would no longer be `const`)
- implement `getError()`
```cpp
#include "Stack.h"
#include <iostream>
#include <string>
using namespace std;
string toString(ErrorType e);

void main() {
    Stack s1(5);

    s1.Push(99); s1.Push(345); s1.Push(235);

    for (int Idx = 0; Idx < 5; Idx++) { // causes 2 underflow errors
        s1.Pop();
        // Check for error after each Pop()
        if (s1.getError() != NO_ERROR) {
            cout << "Error: " << toString(s1.getError()) << endl;
        }
    }
}
string toString(ErrorType e) {
    switch (e) {
    case NO_ERROR : return "no error";
    case STACK_UNDERFLOW : return "stack underflow";
    case STACK_OVERFLOW : return "stack overflow";
    default : return "unknown error";
    }
}
Assignment of Objects

A default assignment operation is provided for objects (just as for struct variables):

class Complex {
    private:
        double Real, Imag;
    public:
        Complex( );
        Complex(double RealPart, double ImagPart);
        . . .
        double Modulus( );
    };
    . . .
    Complex A(4.3, 2.9);
    Complex B;
    B = A;    // copies the data members of A into B

The default assignment operation simply copies values of the data members from the “source” object into the corresponding data members of the “target” object.

This is satisfactory in many cases. However, if an object contains a pointer to dynamically allocated memory, the result of the default assignment operation is usually not desirable…
Consider the LinkList class discussed earlier:

class Integer {
private:
    int Data;
public:
    Integer(int newData);
};
typedef Integer ItemType;

LinkList myList();

for (int Idx = 0; Idx < 5; Idx++) {
    Integer newInteger(Idx);
    myList.AppendNode(newInteger);
}

These nodes are not data members of the object myList.
Now, suppose we declare another LinkList object and assign the original one to it:

```c++
LinkList anotherList;
anotherList = myList;
```

Here’s what we get:

```
Head -> 0 -> 1 -> 2 -> 3 -> 4
```

`anotherList` does not get a new copy of the linked list nodes.

It just gets a copy of the `Head` pointer from `myList`.

So both `LinkList` objects share the same dynamic data.

This is almost certainly NOT what was desired when the code above was written.

This is known as making a “shallow copy” of the source object.
When an object contains a pointer to dynamically allocated data, we generally will want the assignment operation to create a complete duplicate of the “source” object.

This is known as making a “deep copy” (since the copy operation must follow any pointer in the object to its target).

In order to do this, you must provide your own implementation of the assignment operator for the class in question, and take care of the “deep” copy logic yourself. Here’s a first attempt:

```c++
LinkList& LinkList::operator=(const LinkList& otherList) {
    Head = NULL;       // don’t copy pointers
    Tail = NULL;
    Curr = NULL;

    LinkNode* myCurr = otherList.Head;   // copy list

    while (myCurr != NULL) {
        ItemType xferData = myCurr->getData();
        AppendNode(xferData);
        myCurr = myCurr->getNext();
    }
}
```

What if the target of the assignment already has its own linked list?
Here’s a somewhat improved version:

```cpp
LinkList& LinkList::operator=(const LinkList& otherList) {
    if (this != &otherList) { // self-assignment??
        MakeEmpty(); // delete target’s list
    }
    LinkNode* myCurr = otherList.Head; // copy list
    while (myCurr != NULL) {
        ItemType xferData = myCurr->getData();
        AppendNode(xferData);
        myCurr = myCurr->getNext();
    }
}

bool LinkList::MakeEmpty() {
    gotoHead();
    while ( !isEmpty() ) {
        DeleteCurrentNode();
    }
    return (Head == NULL);
}
```

---

Test for self-assignment.
Delete target’s linked list, if any.

Note use of target’s this pointer. Also note that the default scope is that of the target object, not the source object.
When an object is used as an actual parameter in a function call, the distinction between shallow and deep copying can cause seemingly mysterious problems.

```cpp
void PrintList(LinkList& toPrint, ostream& Out) {
    ItemType nextValue;
    int Count = 0;

    Out << "Printing list contents: " << endl;
    toPrint.gotoHead();
    if (toPrint.isEmpty()) {
        Out << "List is empty" << endl;
        return;
    }

    while ( toPrint.moreList() ) {
        nextValue = toPrint.getCurrentData();
        nextValue.Print(Out);
        toPrint.Advance();
    }
}
```

The LinkList object is passed by reference because it may be large, and making a copy would be inefficient.

What do we risk because the list is not passed by constant reference or by value?

Why is the list not passed by constant reference?
Passing Objects

In the previous example, the object parameter cannot be passed by constant reference because the called function does change the object (although not the content of the list itself).

However, since constant reference is not an option here, it may be preferable to eliminate the chance of an unintended modification of the list and pass the LinkList parameter by value.

However, that will cause a new problem.

When an object is passed by value, the actual parameter must be copied to the formal parameter (which is a local variable in the called function).

This copying is managed by using a special constructor, called a *copy constructor*. By default this involves a shallow copy. That means that if the actual parameter involves dynamically allocated data, then the formal parameter will share that data rather than have its own copy of it.
Passing Objects by Value

In this case:

```cpp
// use pass by value:
void PrintList(LinkList toPrint, ostream& Out) {
    // same implementation
}

void main() {
    LinkList BigList;
    // initialize BigList with some data nodes
    PrintList(BigList, cout); // print BigList
}
```

We have the aliasing problem again.

However, the consequences this time are even worse…
As `PrintList()` executes, the `Curr` pointer in `toPrint` is modified and nodes are printed:

```cpp
void PrintList(LinkList toPrint, ostream& Out) {
    // operations on myList, which is local
}
```

When `PrintList()` terminates, the lifetime of `toPrint` comes to an end and its destructor is automatically invoked:

But of course, that’s the same list that `BigList` has created. So, when execution returns to `main()`, `BigList` will have lost its list, but `BigList.Head` will still point to that deallocated memory.

Havoc will ensue.

Destructing `toPrint` causes the deallocation of the list of nodes to which `toPrint.Head` points.
Copy Constructors

Possible solutions to this problem:
  • always pass objects by reference – undesirable
  • force a deep copy to be made when pass by value is used

The second option can be achieved by providing a user-defined copy constructor for the class, and implementing a deep copy.

When a user-defined copy constructor is available, it is used when an actual parameter is copied to a formal parameter.

```cpp
LinkList::LinkList(const LinkList& Source) {
    Head = Tail = Curr = NULL;
    LinkNode* myCurr = Source.Head; // copy list
    while (myCurr != NULL) {
        ItemType xferData = myCurr->getData();
        AppendNode(xferData);
        myCurr = myCurr->getNext();
    }
}
```

The copy constructor takes an object of the relevant type as a parameter (constant reference should be used). Implement a deep copy in the body of the constructor and the problem described on the previous slides is solved.

No self-test is needed because the copy constructor is used to initialize the target object.
Initialization

When an object is declared, it may be initialized with the value of an existing object (of the same type):

```c++
void main() {
    LinkList aList; // default construction
    // Now throw some nodes into aList
    // ...

    LinkList anotherList = aList; // initialization
}
```

Technically initialization is different from assignment since here we know that the “target” object does not yet store any defined values.

Although it looks like an assignment, the initialization shown here is accomplished by the copy constructor.

If there is no user-defined copy constructor, the default (shallow) copy constructor manages the initialization.

If there is a user-defined copy constructor, it will manage the copying as the author of the LinkList class wishes.
When implementing a class that involves dynamic allocation, if there is any chance that:

- objects of that type will be passed as parameters, or
- objects of that type will be used in initializations

then your implementation should include a copy constructor that provides a proper deep copy.

If there is any chance that:

- objects of that type will be used in assignments

then your implementation should include an overloaded assignment operator that provides a proper deep copy.

This provides relatively cheap insurance against some very nasty behavior.
A predefined variable, provided automatically, which is a pointer to the object itself.

```
LinkList& LinkList::operator=(const LinkList& otherList) {
    if (this != &otherList) { // self-assignment??
        ...
    }
}
```

There are also situations where an object may want to pass itself to another object.

*this* makes that possible.

Very possibly the worst naming decision in the entire C++ language.
Private Member Functions

A member function that can only be called within the class.

Avoids duplication of code.

Useful for:

- Helper Functions (e.g., key search function in linked list class)
- Error Checking
- Repeated Code
- Intermediate values