Rationale for the C++ Class

Bjarne Stroustrup from *The C++ Programming Language*, 3rd Edition, page 223:

The aim of the C++ class construct is to provide the programmer with a tool for creating new types that can be used as conveniently as the built-in types.

A type is a concrete representation of a concept.

For example, the C++ built-in type double with its operations +, -, *, etc., provides a concrete approximation of the mathematical concept of a real number. A class is a user-defined type.

We design a new type to provide a definition of a concept that has no counterpart among the built-in types.

A program that provides types that closely match the concepts of the application tend to be easier to understand and to modify than a program that does not.

A well-chosen set of user-defined types makes a program more concise. In addition, it makes many sorts of code analysis feasible. In particular, it enables the compiler to detect illegal uses of objects that would otherwise remain undetected until the program is thoroughly tested.

©2000-2002 McQuain WD

Rationale for the C++ Class

The C++ class type provides a means to encapsulate heterogeneous data elements and the operations that can be performed on them in a single entity.

Like the `struct` type, a `class` type may contain data elements, called data members, of any simple or structured type (including class types).

Also like the `struct` type†, a `class` type may also contain functions, called function members or methods, that may be invoked to perform operations on the data members.

The class type also provides mechanisms for controlling access to members, both data and function, via the use of the keywords `public`, `private` and `protected`.

A variable of a class type is referred to as an object, or as an instance of the class.

† What's the difference? Members of a struct type are, by default, public; members of a class type are, by default, private.

©2000-2002 McQuain WD

Encapsulation and Information Hiding

The C++ class provides support for both…

Encapsulation

A C++ class provides a mechanism for bundling data and the operations that may be performed on that data into a single entity.

Information Hiding

A C++ class provides a mechanism for specifying access restrictions on both the data and the operations which it encapsulates.

©2000-2002 McQuain WD

Interface for a Simple Date Class

Here's a simple class type declaration:

```
class Date {
public:
   Date();
   Date(int newMonth, int newDay,
        int newYear);              // constructors
   int YearIs( ) const;            // returns Year
   int MonthIs( ) const;           // returns Month
   int DayIs( ) const;             // returns Day
private:
   int Year;
   int Month;
   int Day;
};
```

This statement defines a new data type; it does not declare a variable of that type; no storage is allocated.

Keyword "const" applied in a member function prototype specifies that the function is not permitted to modify any data members.

The Date class incorporates three data members, Year, Month, and Day, and four function members.

Typically, the class type declaration is incorporated into a header file, providing a user with a description of the interface to the class, while the implementations of the class member functions are contained in a cpp file.

©2000-2002 McQuain WD

Simple Date Class

Given the class type declaration, a user may declare variables of that type in the usual way:

```
Date Today, Tomorrow, AnotherDay;
```

Unlike simple types (such as `int`), default initializations are performed, as specified in the default (parameterless) constructor. If the class does not provide such a constructor, one is automatically provided, but it will not initialize data members in any useful way.

The data members of the Date class are declared as being private. The effect is that the data members cannot be accessed in the way fields of a struct variable are accessed:

```
cout << Today.Month;
```

will generate a compile-time error. A user of a Date variable may access only those members which were declared public. So, the user could print the `Month` field of `Today` by using the public member function `MonthIs( )`:

```
cout << Today.MonthIs();
```

Note the use of the field selection operator '.'.

©2000-2002 McQuain WD

Member Function Definitions

Of course, the member functions of a class type must be defined. Moreover, it is possible for two different class types to have member functions with the same names. In fact, you've already seen that with I/O streams.

To disambiguate the connection between the function being defined and the class type to which it belongs, the function definition must indicate the relevant class type name by using the scope resolution operator (`::`):

Function definition (implementation) goes in cpp file.

```
// Date::MonthIs()
// Pre:   self has been initialized
// Post:  Month field of self is returned
int DateType::MonthIs() const {

   return Month;
}
```

Note that, as a member function of class Date, MonthIs( ) may access the data members directly:

Members (data or function) declared at the outermost level of a class type declaration have <u>class scope</u>; that is, they are accessible by any function member of an instance of that class type.

©2000-2002 McQuain WD

Implementation Organization

For separate compilation, a typical organization of the class implementation would involve two files:

`Date.h`	class declaration
`Date.cpp`	function member definitions

Suppose that a user of the Date class writes a program consisting of a single source file, `DateClient.cpp`. Then the user would incorporate the Date class files as follows:

```
//DateClient.cpp
//
#include "Date.h"
. . .
```

```
//Date.h
//
class Date {
   . . .
};
```

```
//Date.cpp
//
#include "Date.h"
int Date::MonthIs( ) const {
   return Month;
}
. . .
```


©2000-2002 McQuain WD

Using the Member Functions

In addition to using the class member functions directly, the user of a class may also implement higher-level functions that make use of the member functions. For example:

```
enum RelationType {Precedes, Same, Follows};

RelationType ComparedTo(Date dateA, Date dateB) {

   if (dateA.YearIs() < dateB.YearIs())
      return Precedes;
   if (dateA.YearIs() > dateB.YearIs())
      return Follows;
   if (dateA.MonthIs() < dateB.MonthIs())
      return Precedes;
   if (dateA.MonthIs() > dateB.MonthIs())
      return Follows;
   if (dateA.DayIs() < dateB.DayIs())
      return Precedes;
   if (dateA.DayIs() > dateB.DayIs())
      return Follows;
   return Same;
}
```


©2000-2002 McQuain WD

Using the Member Functions

Then:

```
Date Tomorrow, AnotherDay;

Tomorrow.Initialize(10, 6, 1881);
AnotherDay.Initialize(10, 12, 1885);

if ( ComparedTo(Tomorrow, AnotherDay) == Same ) {

   cout << "Think about it, Scarlett!" << endl;
}
```

Of course, the Date class designer could also have implemented a member function for comparing two dates.

In fact, that would be more natural and more useful, since there is one natural way for the comparison to be (logically) defined.

©2000-2002 McQuain WD

Additional DateType Methods


```
// add to Date.h:

enum RelationType {Precedes, Same, Follows}; // file scoped
RelationType ComparedTo(Date dateA);         // to public section
```

```
// add implementation to Date.cpp:
RelationType Date::ComparedTo(Date otherDate) {

   if (Year < otherDate.Year)
      return Precedes;
   if (Year > otherDate.Year)
      return Follows;
   if (Month < otherDate.Month)
      return Precedes;
   if (Month > otherDate.Month)
      return Follows;
   if (Day < otherDate.Day)
      return Precedes;
   if (Day > otherDate.Day)
      return Follows;
   return Same;
}
```

```
if ( Tomorrow.ComparedTo(AnotherDay) == Same )
   cout << "Think about it, Scarlett!" << endl;
```


©2000-2002 McQuain WD

Using the Methods

Another example:

```
void PrintDate(Date aDate, ostream& Out) {

   PrintMonth( aDate.MonthIs( ), Out );
   Out << ' ' << aDate.DayIs( )
       << ", " << setw(4) << aDate.YearIs( ) << endl;
}

void PrintMonth(int Month, ostream& Out) {
   switch (Month) {
   case  1: Out << "January"; return;
   case  2: Out << "February"; return;
     . . .
   case 12: Out << "December"; return;
   default: Out << "No Month";
   }
}
```

These are not natural candidates to be member functions…

Then:

```
Date LeapDay;
LeapDay.Initialize(2, 29, 2000);

PrintDate(LeapDay, cout);
```

will print:

```
February 29, 2000
```


©2000-2002 McQuain WD

Taxonomy of Member Functions

Member functions implement operations on objects. The types of operations available may be classified in a number of ways. Here is one common taxonomy:

Constructor	an operation that creates a new instance of a class (i.e., an object)
Mutator	an operation that changes the state of one, or more, of the data members of an object
Observer (reporter)	an operation that reports the state of one or more of the data members of an object, without changing them
Iterator	an operation that allows processing of all the components of a data structure sequentially

In the Date class, Date( ) is a constructor while YearIs( ), MonthIs( ) and DayIs( ) are observers. The given version of Date does not provide any iterators or mutators.

©2000-2002 McQuain WD

Default Constructor

The Date class has a two explicit constructor member functions.

It is generally desirable to provide a default constructor, since that guarantees that any declaration of an object of that type must be initialized. :

```
Date::Date( ) {

   Month = Day = 1;     // default date
   Year  = 1980;
}
```

The "default" constructor is simply a constructor that takes no parameters.

Constructor Rules

- the name of the constructor member must be that of the class
- the constructor has no return value; `void` would be an error
- the default constructor is called automatically if an instance of the class is defined; if the constructor requires any parameters they must be listed after the variable name at the point of declaration

©2000-2002 McQuain WD

Other Constructors

The Date class also has a nondefault constructor. This provides a user with the option of picking the date to be represented (essential since there are no mutator member functions).

```
Date::Date(int aMonth, int aDay, int aYear) {

   if (  (aMonth >= 1 && aMonth <= 12)
      && (aDay >= 1) && (aYear >= 1) ) {
      Month = aMonth;
      Day   = aDay;
      Year  = aYear;
   }
   else {
      Month = Day = 1;     // handling user error
      Year  = 1980;
   }
}
```

The compiler determines which constructor is invoked by applying the rules for overloaded function names (slide 16).

When the constructor requires parameters they must be listed after the variable name at the point of declaration:

```
Date aDate(10, 15, 2000);

Date bDate(4, 0, 2005);    // set to 1/1/1980
```


©2000-2002 McQuain WD

Default Constructor Use

If you do not provide a constructor method, the compiler will automatically create a simple default constructor. This automatic default constructor:

- takes no parameters
- calls the default constructor for each data member that is an object of another class
- provides no initialization for data members that are not objects

Given the limitations of the automatic default constructor:

Always implement your own default constructor when you design a class!

©2000-2002 McQuain WD

Overloading, Briefly

In C++ it is legal, although not always wise, to declare two or more functions with the same name. This is called overloading.

However, it must be possible for the compiler to determine which definition is referred to by each function call. When the compiler encounters a function call and the function name is overloaded, the following criteria are used (in the order listed) to resolve which function definition is to be used:
Considering types of the actual and formal parameters:

1. Exact match (no conversions or only trivial ones like array name to pointer)
2. Match using promotions (bool to int; char to int; float to double, etc.)
3. Match using standard conversions (int to double; double to int; etc.)
4. Match using user-defined conversions (not covered yet)
5. Match using the ellipsis . . . in a function declaration (ditto)

Clear as mud, right? Keep this simple for now. Only overload a function name if you want two or more logically similar functions, like the constructors on the previous slides, and then only if the parameter lists involve different numbers of parameters or at least significantly different types of parameters.

©2000-2002 McQuain WD

Operator Overloading

Standard Operator Overloading

C++ language operators, (e.g., “==”, “++”, etc.) can be overloaded to operate upon user-defined types.

```
// add to Date.h:
bool operator==(Date otherDate) const ;
```

```
// add to Date.cpp:
bool Date::operator==(Date otherDate) const {
   return( (Day   == otherdate.Day   ) &&
           (Month == otherDate.Month ) &&
           (Year  == otherDate.Year  ));
}
```

```
Date aDate(10, 15, 2000);
Date bDate(10, 15, 2001);

if (aDate == bDate) { . . .
```

When logically appropriate, overloading relational operators intelligently allows users to treat objects of a user-defined type as naturally as the simple built-in types.

©2000-2002 McQuain WD

Default Function Arguments

Technique provided to allow formal parameters to be assigned default values that are used when the corresponding actual parameter is omitted.

```
// add to Date.h
Date::Date(int aMonth = 1, int aDay = 1, int aYear = 1980);
```

The default parameter values are provided as initializations in the parameter list in the function prototype, but not in its implementation.

```
// add to Date.cpp
Date::Date(int aMonth, int aDay, int aYear){

   if (  (aMonth >= 1 && aMonth <= 12)
      && (aDay >= 1) && (aYear >= 1) ) {
      Month = aMonth;
      Day   = aDay;
      Year  = aYear;
   }
   else {
      Month = Day = 1;     // default date
      Year  = 1980;
   }
}
```

This allows the omission of the default constructor, since default values are provided for all the parameters. In fact, including the default constructor now would result in a compile-time error.

©2000-2002 McQuain WD

Default Function Arguments

If a default argument is omitted in the call, the compiler automatically inserts the default value in the call.

```
Date dDate(2,29);          // Feb 29, 1980
Date eDate(3);             // March 1, 1980
Date fDate();              // Jan 1, 1980
```

Restriction: omitted parameters in the call must be the rightmost parameters.

```
Date dDate(,29);           // error
```

Be very careful if you mix the use of overloading with the use of default function parameters.

Default parameter values may be used with any type of function, not just constructors. In fact, not just with member functions of a class.

©2000-2002 McQuain WD

Default Arguments Usage

Default Arguments in prototypes

- Omitted arguments in the function prototype must be the rightmost arguments.

Default Arguments - Guidelines

- Default arguments are specified in the first declaration/definition of the function, (i.e. the prototype).
- Default argument values should be specified with constants.
- In the parameter list in function declarations, all default arguments must be the rightmost arguments.
- In calls to functions with > 1 default argument, all arguments following the first (omitted) default argument must also be omitted.

Default Arguments and Constructors

- Default argument constructors can replace the need for multiple constructors.
- Default argument constructors ensure that no object will be created in an non-initialized state.
- Constructors with completely defaulted parameter lists, (can be invoked with no arguments), becomes the class default constructor, (of which there can be only one).

©2000-2002 McQuain WD

Inline Member Functions

Class declaration inline functions

- inline functions should be placed in header files to allow compiler to generate the function copies.

```
class Date {
public:
   Date(int newMonth = 1, int newDay = 1,
        int newYear = 1980);
   int YearIs  () const  {return Year};
   int MonthIs () const  {return Month};
   int DayIs   () const  {return Day};
private:
   int Year, Month, Day;
};
```

- Member functions defined in a class declaration are implicitly inlined.
- Efficiency is traded off at the expense of violating the information hiding by allowing the class clients to see the implementation.
- Reference to the class data members by the inline functions before their actual definition is perfectly acceptable due to the class scoping.
- To avoid confusion, the private members can be defined first in the class or the inline functions can be explicitly defined after the class declaration, (but in the header file).

©2000-2002 McQuain WD

Properties of a Good Class

A well-designed class will exhibit the following characteristics:

abstraction

correctness

safety

efficiency

generality

©2000-2002 McQuain WD

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.

©2000-2002 McQuain WD

Example – Integer Stack Class Interface


```
// 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

©2000-2002 McQuain WD

Stack Class Constructor and Destructor

```
// Stack.cpp
#include "Stack.h"
#include <new>
using namespace std;

Stack::Stack(int InitSize) {
   Capacity = InitSize;
   Top = 0;
   Stk = new(nothrow) int[Capacity];
   if (Stk == NULL) Capacity = 0;
}

Stack::~Stack() {
   delete [] Stk;
}
```

Note dynamic allocation of stack array. The use of "nothrow" guarantees that if the memory allocation fails then Stk will be set to NULL.

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.

©2000-2002 McQuain WD

Stack::Push()


```
// . . . 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.

©2000-2002 McQuain WD

Stack::Pop() and Stack::Peek()


```
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.

©2000-2002 McQuain WD

Stack::isEmpty() and Stack::isFull()


```
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).

©2000-2002 McQuain WD

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.

©2000-2002 McQuain WD

Revised Stack Interface

First revise the Stack class interface to incorporate an error recording mechanism:

```
// Stack.h - conditional compilation directives omitted to save space
enum ErrorType { NO_ERROR, STACK_UNDERFLOW, STACK_OVERFLOW };   New
class Stack {
private:
   int  Capacity;            // current stack array size
   int  Top;                 // first available index in stack array
   int* Stk;                 // stack array (allocated dynamically)
   ErrorType LastError;      // 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   New
   ~Stack();                 // deallocate stack array
};
```


©2000-2002 McQuain WD

Revised Stack::Pop()

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

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

New

New

Other changes:

- constructor should initialize `LastError` to `NO_ERROR`
- `Push()` should set `LastError` to `STACK_OVERFLOW` when Stack size cannot be increased (and could be `void` now)
- `Peek()` should also set `LastError` to `STACK_UNDERFLOW` if Stack is empty (and would no longer be `const`)
- implement `getError()`

©2000-2002 McQuain WD

Using Revised Stack::Pop()


```
#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";
   }
}
```

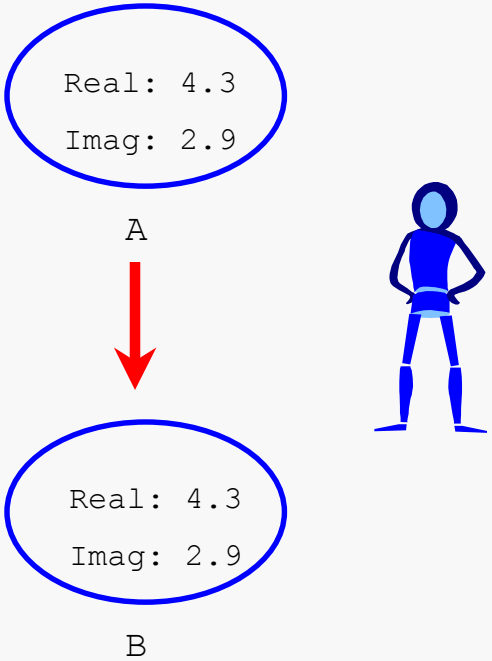

©2000-2002 McQuain WD

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…

©2000-2002 McQuain WD

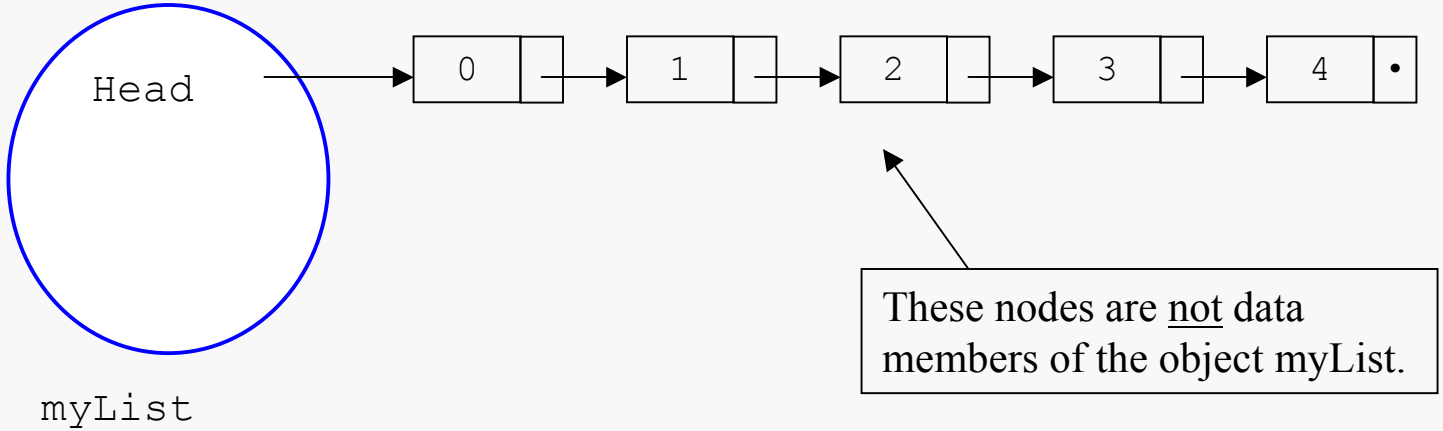
Dynamic Content

Consider a `LinkList` class:

```
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);
}
```



©2000-2002 McQuain WD

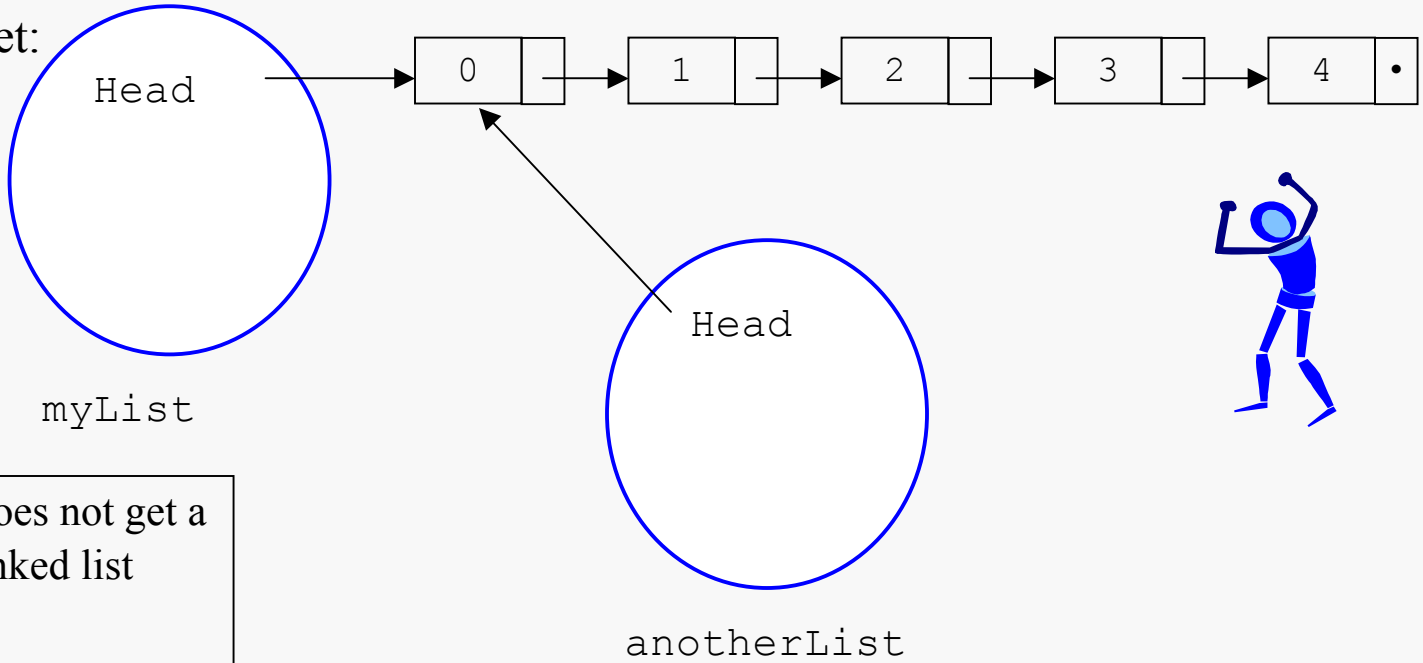
Shallow Copying

Now, suppose we declare another LinkList object and assign the original one to it:

```
LinkList anotherList;

anotherList = myList;
```

Here's what we get:



`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.

©2000-2002 McQuain WD

Deep Copying

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:

```
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();
   }
   return (*this);
}
```

What if the target of the assignment already has its own linked list?

©2000-2002 McQuain WD

…Improved Version

Here's a somewhat improved version:

```
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();
      }
   }
   return (*this);
}
```

Test for self-assignment.

Delete target's linked list, if any.

```
bool LinkList::MakeEmpty() {

   gotoHead();
   while ( !isEmpty() ) {
      DeleteCurrentNode();
   }

   return (Head == NULL);
}
```

Note use of target's this pointer. Also note that the default scope is that of the target object, not the source object.

©2000-2002 McQuain WD

Passing an Object as a Parameter

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.

```
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?

©2000-2002 McQuain WD

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.

©2000-2002 McQuain WD

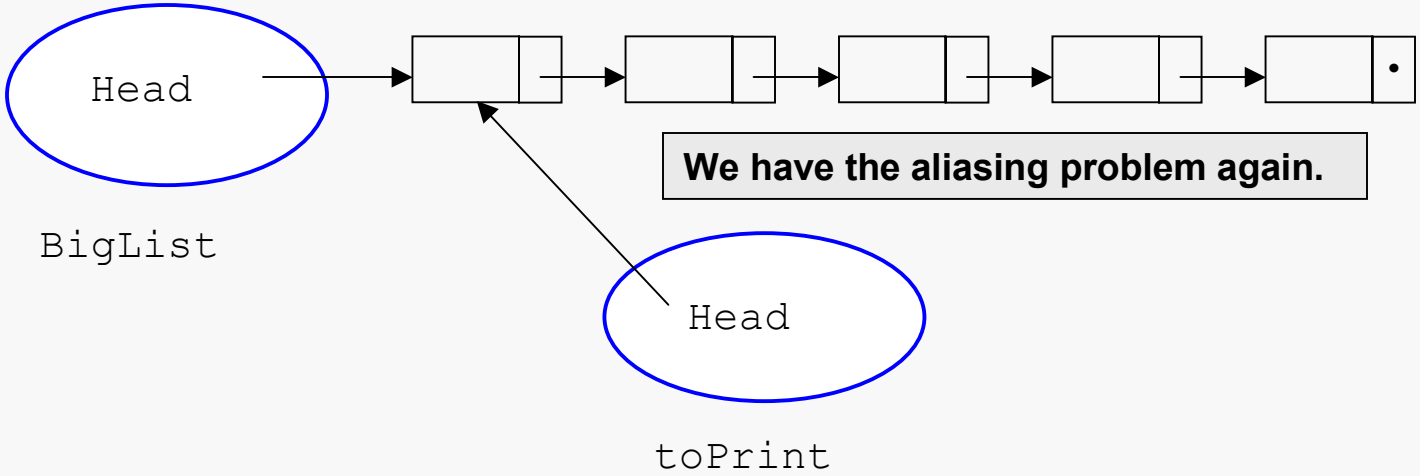
Passing Objects by Value

In this case:

```
// 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…

©2000-2002 McQuain WD

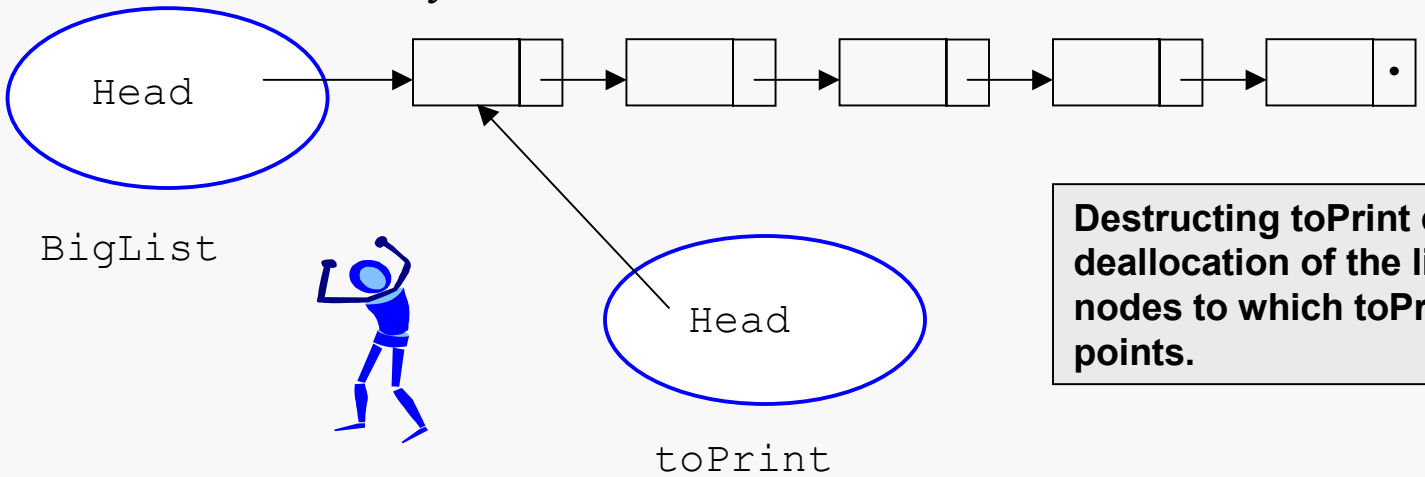
Passing Objects by Value

As `PrintList()` executes, the `Curr` pointer in `toPrint` is modified and nodes are printed:

```
void PrintList(LinkList toPrint, ostream& Out) {

  // operations on toPrint, which is local
}
```

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



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

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.

©2000-2002 McQuain WD

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.

```
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();
   }
}
```

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

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.

©2000-2002 McQuain WD

Initialization

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

```
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.

©2000-2002 McQuain WD

Moral

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.

©2000-2002 McQuain WD

Normal Method Invocation

Method invocations, like all function calls, have processing overhead:

`toPrint.getData( );` requires the following steps:

save registers on the stack
create new activation record
push function arguments on the stack
execute code for `toPrint.getData()`
remove activation record
restore saved registers

If the body of the called function were simply listed inline, with the appropriate substitutions of variables if there were parameters, we'd only have to do:

execute code for `toPrint.getData()`

©2000-2002 McQuain WD

Inline Member Functions

Generally more efficient for *small to medium* functions

Expanded in-place of invocation

- Eliminates method invocation overhead
- Compiler generates necessary code and maps parameters automatically
- Still only one copy of function implementation

Two ways to specify an inline method

- Provide implementation during <u>class</u> definition (default inline)
- Use ‘inline’ keyword in <u>function</u> definition (explicit inline)

©2000-2002 McQuain WD

Inline Member Function Examples


```
// Stack.h
class Stack {
private:
. . .
public:
   . . .
   bool isEmpty() const {return (Top == 0);};  // default inline
   bool isFull() const;
   . . .
};
```

```
inline bool Stack::isEmpty() const {  // explicit inline
   return (Top == 0);
}
```


©2000-2002 McQuain WD

Tradeoffs of inline methods

Default inline violates some basic software engineering goals

- separation of interface and implementation is broken
- information hiding is lost – implementation is now exposed to clients

All code that invokes an inline must be recompiled when:

- method is changed
- switching from inline to regular or vice-versa

Inline is <u>request,</u> not <u>command</u> to compiler

- Could manually inline, but at what cost…?

Size of executable may increase

- although that's not usually that much of a factor

©2000-2002 McQuain WD

‘this’ pointer

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.

©2000-2002 McQuain WD

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

©2000-2002 McQuain WD