Object-Oriented Programming

Part III

The Language C++

Summary of Part III

- Basic language extensions (Chapter 9 of Drake's book)
  - Lexical elements
  - Operators and precedence
  - Scope rules
  - Type system
  - Function overloading, reference parameters
  - Input/Output streams

Summary of Part III (cont' d)

- Introduction to classes in C++ (Drake, Chapters 10 and 11)
  - Syntax and semantics of class definitions
  - Data members and member functions
  - Constructors and destructors
  - Static members
  - Class scope and nested classes
  - Operator overloading

Summary of Part III (cont' d)

- Inheritance in C++ (Drake, Chapter 12)
  - Derived classes
  - Dynamic binding of methods and messages
  - Derived class scope
  - Constructors and destructors in derived classes
  - Method refinement and overriding
  - Implementation of dynamic binding
  - Multiple inheritance
- Templates (if time permits)
- Exceptions (if time permits)

Important Disclaimer

- Knowledge of ANSI C is assumed!!!!

C++ vs. Smalltalk

- In Smalltalk we had:
  - Interpreted environment, automatic GUI builder
  - Pure OO model
  - Dynamic typing (full identifier polymorphism)
  - Limited paradigm language (blocks and expressions)
  - Single inheritance
  - Block closures
  - Automatic garbage collection
  - Reference semantics of identifiers

C++ vs. Smalltalk

- In C++ we' ll get:
  - Hybrid OO model
  - The tension between static typing and OO paradigm
  - Multiple inheritance
  - Programmer-controlled memory deallocation
  - Copying **and** reference identifier semantics

Lexical Elements

- Syntax of numeric, character and string literals
- Additional // syntax for comments:

```
// a recursive implementation of factorial
long factorial(int x) {

  if (x <= 1) return 1;        // base case

 /* old style comment still available:  recursive case */
 else return x * factorial(x-1);
}
```


Operators

Same as in C plus some more:

- new // object allocation
- delete // object deallocation
- :: // a new scope operator
- ::* // pointer to data or function member
- .* // selector of pointer member
- ->* // selector of pointer member with dereferencing

Operator Precedence Rules (High to Low)

::, ::*
(), (), [], ->, .
!, – (unary), + (unary), ++, – –, ~, *, &, (type), sizeof, new, delete
->*, .* (pointer to member selectors)
*, /, %,
+, –
<<, >>
<, <=, >, >=
==, !=
&
^
|
&&
||
?:
=, +=, –=, *=, /=, %=, &=, |=, ^=, <<=, >>=
, (expression sequencing)

Type system extensions

Three main differences wrt to ANSI C

1. Identifier definitions can appear anywhere in a block

   Example

```
long fact(int n) {
 if (n <= 1) return 0L;
 else {
   long total = 1;
   for (int i = 2; i <= n; i++)
     total *= i;
   return total;}
```


Type system extensions

Differences wrt to ANSI C

2. const variables cannot appear in lhs of assignments
   - Beware of const pointers, two objects involved (pointer and referent)
   - Mnemonic trick: Read from right to left

   Examples

```
const int SIZE = 100;          // SIZE cannot be changed
const Person *ptr ;            // referent cannot be changed
Person* const ptr ;            // pointer cannot be changed
const Person* const ptr        // neither can be changed
```


Type system extensions

Differences with ANSI C (cont' d)

3. No need for tag names for enum, union, and struct types
   As a result, this is an error in C++:

```
struct Person {
... }

enum Person {
... }
```


The Bool Type

- Defined as follows
  enum bool {false, true};
- Conversion from enum symbol to int is automatic, reverse is not (explicit cast required)

```
enum Weekday {M, Tu, W, Th, F};
Weekday aDay; // no keyword
aDay = 3;     // ERROR
aDay = (Weekday) 3; // OK
```


Additional syntax for type conversion

- Old: (type) expression
- New: type(expression)
- Examples: (double) x;
  double(x);

Dynamic allocations and deallocations

Operators new and delete (plus malloc and free)

- Operator new allocates heap space
- Main differences wrt malloc
  - new takes a type specification (vs. int), returns type correct pointer (vs. void) to allocated memory
  - no need for type conversions and sizeof operator
  - new allows for initialization of allocated memory (done by constructor for class instances)

Dynamic allocations and deallocations

Examples of Operators new and delete

```
double* pDbl;
char* pChar;
pDbl = new double(3.14);
pChar = new char( 'a' );
```


Dynamic Array Allocations

Operator new[]

- Argument in brackets specifies number of items contained in array (any R-T expression allowed)
- Examples:

```
char** StringArray;
char* aString;
const int size = 100;
StringArray = new char*[size];     // define array of strings
aString = new char[10];            // define just 1 string of 10 chars
```


Deallocation Operators

- Operators delete and delete[] deallocate memory allocated with new and new[]
- Both trigger clean-up actions (e.g., class destructors), unlike free
- Examples:

```
delete[] StringArray;
delete[] aString;
```

- Android warning: Failure to use delete[] will crash app!

Functions in C++

Start from C functions; add many notable differences

- Header: empty argument list means no args
  Ellipses used (...) for no arg checking

```
// a function without arguments
int foobar();
```

- In C++ functions must be declared before use
  (no more default int declarations)

Functions in C++

- Reference parameters/arguments

  C: parameters passing by *value* only

  C++: reference parameters supported
- When function is called, only argument address passed to function

  Reference identifier = synonym for an l-value

  At first approximation,
  reference ids = pointers without pointer syntax

Syntax of Reference Parameters

- Follow type specification by an ampersand &
- Do not confuse with "address-of" & operator
- A programming example:

```
bool foobar(double& x, double& y)
{ ...
  x = x + 2*y;
  ...
  return true;}
```


Reference Arguments

- Argument should be an l-value (arg is not copied and must be modifiable)

  (Recall distinction as in statement x=y;)
- Expressions are legal arguments (unlike Pascal), but assignment to the corresponding parameters will only modify a temporary object created on the stack

```
double a, b;
...
flag = foobar(a,b);          // OK call
flag = foobar(a+3, b);       // mmm...
```

- New compilers flag this as an error

Semantics of Reference Parameters

- A cross between passing pointer and passing value (pointer-like behavior, but value-like syntax)
- Unlike pointers, reference ids cannot be dereferenced or changed
- However, no need to deference a parameter identifier the way a pointer would be (i.e. with * and -> operators)
- Reference identifier can be used either as l-value or r-value

Semantics of Reference Parameters

- Very useful to avoid using explicit pointers (as in ANSI C) or value identifiers (which can be quite expensive because of copying semantics)

This swaps the arguments in caller

```
void swap (int& x, int& y) {
  int temp;
  temp = y;
  y = x;
  x = temp;  }

// this call swaps values stored by arg1 and arg2
int arg1, arg2;
…
swap(arg1,arg2);
```


Semantics of Reference Parameters

- Return values can also be passed by reference (return value can be used as l-value or r-value)

```
char& string_last(char* s1) {
  char* scanner = s1;
  while(*scanner != '\0') {
    scanner++;}
  scanner = scanner - 1;
  return *scanner;
  }

// Example of use of reference return value as an l-value
if (str_length(aString) > 0)
  string_last(aString) = 'z';
```


Default Arguments

- Sometimes useful to omit arguments in calls
- Function supplies default values used when caller does not provide arguments
- Default value must be compile-time constant

```
double distance(int x1, int y1, int x2=0, int y2=0)

distance(15,15,10)  // default y2 = 0
distance(15,15,0)   // defaults x2 = 0 and y2 = 0
distance(15)        //  errror
distance(15,15,,10) //  error
```


Const parameters

- Guarantee that a function does not modify argument

```
double avg(const double& x, const double& y) {
    return (x + y) / 2; }
```

- Especially useful for reference and pointer parameters
- Cannot pass const argument for non-const parameter (although opposite is possible)

```
double foo(double& x, double& y);
...
const double a = 10.0;
double b, c;
...
c = foo(a, b); // ERROR
```


Inline Functions

- Expanded in-line at location of calls (faster code)
- An alternative to C's macros for short functions

Example of macro definition and macro use:

```
#define min(x, y) x<y ? x : y
...
c = min(a,b);  //expands to
c = a<b ? a : b;
```


Inline Functions

- Support for type checks and proper parameter passing semantics
- Problems of #define directive: Operator precedence and side effects

```
c = d * min(a++,  b);          // expands to

c = d * a++ < b ? a++ : b;   // probably not wanted
```


Syntax of Inline Functions

- Inline keyword

  Example of definition:

```
inline int min(int x, int y) { return x<y ? x : y; }
```

- Caveats:
  1. Compiler does not have to perform inline substitution
  2. Restricted to one argument type (opposite of macros)

```
inline double min(double x, double y) { return x<y ? x : y; }
```


Function Overloading

- Ability to define multiple functions with the same identifier and different argument lists in the same scope

  Example

```
void swap(int& x, int& y) {
 int temp = x;
 x = y;
 y = temp; }

void swap(char*& p1, char*& p2) {        // note pointer passed
 char* temp = p1;                        // by reference
 p1 = p2;
 p2 = temp; }
```


Function Overloading

- Compiler uses arg types/number to resolve calls

```
swap(3, 5);              // 1st swap called
swap("hi", "there");     // 2nd swap called
```


Rules for Argument Matching

Basic rules for argument matching

- Return type not used (i.e., because of nesting)
- type and type& are considered the same
- type* and type[] almost considered the same
- const argument different from non-const argument

```
void foo(int& x);
void foo(const int& x);

...
int i;
const int j=10;
foo(i);   // calls 1st definition
foo(j)    // calls 2nd definition
```


Automatic Type Conversions

- Argument matching complicated by automatic type conversions
- Hierarchy for argument matching, in order attempted:
  1. Exact match or trivial conversion (e.g., from type to type& or vice versa; also type[] to type* and type to const type),

     or const coercion in the right direction (from type/type* argument to const type/type* parameter)

Automatic Type Conversions

- Hierarchy for argument matching (cont' d):

  2. Numeric promotion
     - char, short, enum, bool to int
     - int to long
     - float to double

  3. Other standard conversions: all numeric types match all other, the number zero is any pointer

Automatic Type Conversions

- Hierarchy for argument matching (cont' d):
  4. Programmer-defined type conversion between classes (e.g., using constructors, more on this later)
  5. Ellipses (will match anything)

Examples of Overloading Resolution

- Assume following definitions:

```
void foo(int x);
void foo(char* s);
```

- Examples of calls:

```
foo('z');          // OK, 1st definition
foo(''Hello'');    // OK, 2nd definition
foo(5.5);          // loss of information
foo(50L);          // OK, 1st definition
```


Examples of Overloading Resolution

- Additional definition:

```
void foo(double y);
void foo(int x);
void foo(char* s);
```

- Resulting calls:

```
foo('a');          // still OK, numeric promotion
foo(50L);          // ambiguous, two conversions possible
foo(double(50L))   // explicit conversion
```


More on Function Overloading

- Argument matching more complicated when function has multiple parameters
- Rules for multiple parameter case
  1. For each argument, find all functions with best match for that argument
  2. Compute intersection of all sets; if more than one set in intersection, call is ambiguous
  3. If one function in intersection, make sure that function matches at least one argument better than any other definition

Multiple-Parameter Calls

- Assume following overloaded definitions:

```
void foo(int x, int y);
void foo(double x, double y);
```

- Possible calls:

```
int i;
foo(3.5, i);            // ambiguous, empty intersection
foo(3.5, double(i));    // OK, 2nd definition applied (explicit conversion)
```


Function Linking

Main differences wrt ANSI C:

1. Arguments in function calls are checked even if function defined in a different file
2. Name mangling: C++ compiler translates function identifiers into names encoding function arguments (i.e., to resolve overloaded calls)

Identifiers and plausible encodings

```
int min(int x, int y);              ->  _min_int_int
double min(double x; double y);     ->  _min_dbl_dbl
```


Function Linking

- Mangling problem: linking to ANSI C code
  - C compilers do not mangle function identifiers
  - Thus, do not mangle identifiers in calls to C libraries
- Solution: extern ''C'' declaration

  When declaring C function, use this declaration to turn off name mangling

Function Linking

- Disable name mangling for whole C library:

```
extern ''C'' {
 #include <C_library.h>
}
```

- Caveats:
  1. C libraries are often modified to include suitable extern "C" declarations
  2. extern "C" declaration allows C++ functions not to be name mangled (to allow invocation from C code)

Function Pointers

- Similar to C, except now pointers to member functions possible too
- Recall syntax is similar to function declaration, except function identifier is preceded by * and parenthesized:

```
Boolean (*cmp_fn)(double, double);
int (*ptr_fn)(char*, float);
```

- Pointer to function identifier can be assigned an actual function, e.g,

```
int fun1 (char*, float);
…
ptr_fn = &fun1;
```


Function Pointers

- Function invocation can use either usual syntax for dereferencing or not:

```
(*cmp_fn)(0.0, 3.14);        // OK, correct invocation
cmp_fn(0.0, 3.14);           // legal but misleading invocation?
ptr_fn("a", 3.14);           // same problem?
```

- Often used for function parameters

```
double slope( double (*pFun) (double&), double& x1, double& x2) {
  …
 return ((*pFun)(x2) - (*pFun)(x1)) / (x2 – x1 ) ;  }      // nice syntax!
```

- In sum, a useful mechanism with a cumbersome syntax, although not as powerful as block closures

Scope Rules: File Scope Definitions

- Definitions at file scope are visible from the point where they appear throughout file (true of both variable and function identifiers)
- If program consists of multiple linked files, definitions from file A can be used in file B
  - Exception: static definitions
  - Examples of definitions in a file:

```
int i;          // global definition of i

void foo(int i, double y) {     // definition of i overridden
…
}
static int j;     // definition visible only in this file
```


Using Non-Local Definitions

- Recall in C++, all identifiers must be declared before they are used
- To use a variable identifier defined in a different file use extern declaration
  - Function identifiers externed by default
  - Examples of declarations in a different file (cont' d)

```
extern int i;        // using i defined in previous file
…
void foo(int, double); // using foo defined in previous file
…
extern int j;      // error: identifier j was static in previous file
```


Include Files

- Remember to put the following in .h files:
  - constant definitions (using const)
  - type declarations
  - macro definitions (using #define)
  - extern declarations
  - function prototypes
  - definition of inline functions
- Do not put in .h files to avoid multiple definition errors:
  - non-inline function definitions
  - variable definitions (except once)

Include Files

- Facilitate use of identifiers defined in other files
- Use include file for global definitions; use code for local definitions
- Use #include directive to import relevant definitions from other files
- Caveat: Beware of multiple inclusions (avoid by use of #ifndef … #endif macro

```
#ifndef these_defs
#define these_defs
...
#endif
```


Input/Output in C++

- I/O functions of ANSI C still available, e.g., scanf and printf
- C++ also has streams
- Streams: Set of system classes for interactive I/O and file I/O)
- Obviate disadvantages of C-style I/O functions
  - Little or no-type checking on argument lists
  - Lack of extensibility (problem for classes)

Operators << and >>

- Output operator << — Receiver is ostream instance; argument is identifier or expression denoting value to be output
- Overload definitions of << to support different output types
- When used in this way, << is called insertion operator
  - Confusing syntax: When left argument is an integer, this is the left shift operator
  - BTW, same operator precedence as shift

Insertion Operator <<

- Language libraries have overloaded definitions to support built-in types (e.g., int, double, char, etc.)
- Programmer defines additional overloadings to output class instances
- Return type of << expression operator is ostream&
  - Allow cascading of << expressions
  - Parse cascaded expressions left-to-right
  - Predefined stream std::cout denotes standard output (pronounced C-out)

Example of Insertion Operator <<

- A cascaded expression:

```
int x = 10;
double y = 3.14;
std::cout << "Value of x: " << x << " - Value of y: " << y << endl;
```

- Above expression would be parsed as:

```
((((std::cout << "Value of x: ") << x) << "Value of y: ") << y) << endl;
```

- Above expression would output:

```
Value of x: 10 - Value of y: 3.14
```

- Note modifiers (to format, etc.) defined in C++ libraries iostream.h and iomanip.h as is name space std

Operator >>

- Input operator >>: Receiver is an istream instance; argument is identifier (l-value)
- If receiver is istream instance, >> is called extraction operator
  - Do not confuse with right shift!
- Argument is an identifier (l-value) denoting reference to location where input value is stored (no need for operator &, similar to scanf)

Operator >>

- Return value of type istream& allows cascading of input expressions
- Standard input stream: std:cin (pronounced C-in)
- Coding example:

```
int x, y;
std::cin >> x >> y;      // read new x and y values from
                         // standard input
```

- Test for error state after reading values:

```
if (std::cin) {…}
else {…};
```


Classes in C++

Follow principles established by OO paradigm

- A user-defined type
- A set of objects (instances) sharing the same storage structure and behavior
  - Similar to C++ struct construct (main difference: default access level specification)
  - C struct construct with operations
- Support for instance and class variables, instance and class methods

Class Definitions

- Class definition gives:
  1. name — unique identifier for the class
  2. instance variables — components of each instance (similar to struct fields)
  3. class variables — shared by all class instances
  4. instance methods — used by clients to work with instances
  5. class methods — used by clients to work with class

Syntax of Class Definitions

- Divided between two files:
  1. Header file (className.h) — declarations that are imported by clients of className
  2. Code file (className.cpp) — additional definitions, must include className.h
- Syntax of class definition (in header file): Keyword class, followed by class name, followed by body containing variable and function declarations

```
class <class_name> {
...
} ;
```


Body of Class Header

- Define so-called class members:
  - data members, i.e., the variables
  - member functions, i.e., the operations
- Data members defined by:
  - member type (can be value, reference or pointer)
  - member identifier

```
class DLList {
  DLNode head;
  int size;
  ... } ;
```


Body of Class Header

- Members functions:
  - declared with function prototype (at least) in class body
  - actual function definition (with function body) will appear within the code file
  - for inline member functions, give also function definition in class header (.h file)

Body of Class Definition

- Examples of member function declarations in class definition:

```
class DLList {
   ...
  bool isEmpty()  {          // function defined in class header file
     return (size==0) ;
     }
  bool find_element(int x);
  bool insert_element(int x);
  DLList& sort();
   ...
} ;
```


Class Scope

- Class defines its own scope, distinct from file-scope, function and block scope, and scope of other classes
- Identifiers of data members and member functions declared in a class are scoped within that class
- The scope of a class can be explicitly specified with the scope operator ::
- Example: Suppose class C1 defines data member x, member function fun1()
  - C1::x
  - C1::fun1()

Class Scope

- Visibility rules allow direct access to identifiers of data members and member functions from header + code files
- General rules:
  - Member identifiers are visible throughout class definitions, including header and code files
  - Member identifiers follow scope rules based on nesting of units within units
- Consequence: Class-level definitions will:
  - Hide outer-scope definitions of same identifier
  - Be hidden by inner-scope definitions of identifier

Class Scope

- Contrast visibility of members through class code with non-member identifiers, visible only from point where they are declared or defined

  Example: Loop index variable defined in the loop header
- Class identifier declared by literal following class keyword, defined after end of class body
  - Possible to define pointer and reference identifiers of type C1 while defining C1, but not value identifiers
- Non-inline member functions defined at file scope in code file; however, assume they are nested within class

Definition of Member Functions

- Use scope operator in code file
- Scope operator has two formats:
  1. <class_name>::<member_id>
  2. ::<member_id>
- Use (1) to denote class members outside header file that defines class
- Use (2) to denote global identifier hidden by a local definition

Scope Operator ::

- Examples of operator use when defining non-inline member functions

```
int y;                          // file scope definition in class
                                // header file
class C1 {
  int x, y;                     // data member declarations
  void foobar(char*);           // member function declaration
    ...
} ;
                                // in the code file
void C1::foobar(char* y) {
  char* z;
  z = y ;                       // parameter y, not file scope y
  C1::y = 150 ;                 // data member y, not file scope y
  z = z + 10 * ::y;             // use file scope definition of y
   ... }
```


Access to Class Members

- Each member has an access level that determines who has the right to access that member
- Three access levels are available:
  - **private** (default)

    This member identifier is accessible only within the defining class (both in class definition and .cpp file)
  - **public**

    This member is accessible anywhere it is visible (possibly using a qualified name like a struct field)

Access to Class Members

- Third access level:
  - **protected**

    This member identifier is accessible only within the defining class and its subclasses
- Private access is most restrictive, public is least
- Smalltalk: All variable identifiers are protected; all method identifiers are public
- Java: A variation of C++'s approach with 4 access levels
  - protected is different, though

Access to Class Members

- General guidelines to conform with OO paradigm and information hiding
  - Data members should generally be protected, sometimes even private
  - Member functions in class's protocol should generally be public
  - Auxiliary functions should be protected or private

Access to Class Members

- Impact on subclass code class libraries

  When writing subclasses of class C1, members defined protected and public by C1 are accessible in subclass's code but private members are not
- Syntax: appropriate keyword before each portion of class definition

```
class C1  {
public:                         // beginning of public portion
  double foo (double, double);
  void bar(int, int);
protected:                      // protected portion
   ...
} ;
```


Example of Header File

```
class DLList {
 public:
    DLList() ;                          // constructor declared
    bool is_empty() {return (size==0);} // inline function
    bool find_element(int x);           // function declarations
    DLList& insert_element(int x);
    bool delete_element(int x);
    ~DLList() ;                         // destructor declared
 protected:
    int size;                           // data member declarations
    DLNode* first_node;
 private:
   bool DL_scan(int x, DLNode*);        // auxiliary function
} ;
```


Example of Code File

- Some plausible definitions in code file of DLList class

```
// assume .h file declared this id
static const int DLList initial_size = 128;

DLList::DLList() { … }                    // will see constructors later;

bool DLList::find_element(int x) {    // assumed to be in class scope
  if (this->is_empty()) return false;
  else {
    if (first_node->value() == x) return true;    // assume msg value() is
                                                  // defined in DLNode
    else return  this->DL_scan(x, first_node);
  }
}
```


Example of Code File

- Other plausible definitions for doubly linked-list class

```
bool DLList::DL_scan(int x, DLNode* ptr_node) {
  ptr_node = ptr_node->next() ;      // defined in DLNode
  if (!ptr_node) return false;
  else {
    if (ptr_node->value() == x) return true;
    else return DL_scan(x, ptr_node);
  }
}
```


Writing Client Code

- Instances of a class created by static, automatic, or dynamic allocation
  - true of both built-in types and classes
  - similar to ANSI C
- Members of an instance specified with qualified names (e.g., like fields in a struct)
  - member name resolution in scope of class definition (similar to struct)
  - Beware of access restrictions (e.g., for nonpublic members)

Example of Client Code

- Instances of a class created by static, automatic, or dynamic allocation

```
void client(DLList& list1) {
  DLList temp_List;              //  A value identifier (R-T stack)
  DLList* ptr_list;              //  Pointer (referent to be allocated on heap)
  ptr_list = new DLList;         //  Referent allocated on the heap
  if (list1.find_element(10))    //  No dereferencing of list1 (reference iden.)
       {  list1.insert_element(20);  }
  …
  delete ptr_list;               //  avoid a memory leak!
  ptr_list = &list1;
  ptr_list->insert_element(30); // list1 argument permanently modified
}
```


Examples of Illegal Client Code

- Illegal invocations of member functions (assume client code below:)

```
…
DLList* ptr_list = … ;
…
ptr_list->DL_scan(…);          // DL_scan is private in DLList
insert_element(10);            // insert_element undefined at file scope
ptr_list->insert_element(10);  // finally, a good call
```


Class Definition: Additional Details

- Member functions declared, but not defined, in class definition may omit parameter names
  - Hardly a good idea for readability, though
- Members functions defined in class definition are automatically inlined

More Details on Class Definitions

- Member functions declared but not defined in class definition are defined at file scope in code file (e.g., DLList member functions)
  - Function still assumed to be in class scope (recall DLList::insert_element() could access private instance variables, such as first_node directly)
  - Function identifier still defined in class scope
  - Use qualified names to invoke such functions outside class of definition

Naming of Class Members

- Member functions and data members cannot have same name
  - Typically, append or prepend underscore to data member name to distinguish from member function
- But, member function identifiers do not conflict with file scope function ids or with ids defined in different classes even if these functions have same argument signatures
  - Remember each class has its own scope
- Also, member functions can be overloaded within a class

Naming of Class Members

- Example of proper naming of data and function members (with overloading)

```
class Point {
 public:
  int x()   {return x_;}
  int y()   {return y_;}
  int x(int new_x) {int old_x = x_; x_ = new_x; return old_x;}
  int y(int new_y) {int old_y = y_; y_ = new_y; return old_y;}
    …
 protected:
   int x_, y_;
 …
}
```


Referencing the Receiver

- The this identifier used in member functions to denote receiver
  - Implicit declaration in every class C1:

    C1* const this;
- Useful for concatenating calls to a member function (assuming cascaded calls all return a reference to receiver)
  - Do not return the receiver by value or else you will end up working with copies of the receiver, not the receiver

Example of Concatenated Calls

- Assume DLList::insert_element() coded as follows:

```
DLList& DLList::insert_element(int x) {
…
return *this; }
```

- Then, these are legal concatenated calls, parsed from left to right:

```
DLList a_list;
int x, y, z;
…
((a_list.insert_element(x)).insert_element(y)).insert_element(z);
```


Counterexample of Concatenated Calls

- Assume DLList::insert_element() now returns by value:

```
DLList DLList::insert_element(int x) {
…
return *this; }
```

- Then, these are still legal, but original receiver (a_list) gets x only; y and z added to receiver's copies:

```
DLList a_list;
int x, y, z;
…
((a_list.insert_element(x)).insert_element(y)).insert_element(z);
```


i>clicker question

- Read following code segment–What value is printed out:

```
// Suppose a is an integer array with embedded content {5, 5, 5}
int total = 0 ;
a.add_all(10).add_all(10).add_all(10) ;
for (int i = 0; i <= 2; i++)  {
   total += a[i] ; }
std::cout << total ;

// Suppose add_all is member function for Array
Array add_all(int x) {
  for (int j = 0; j <= 2; j++) {
    this->at(j) += x ;}
 return *this;
```

- A=15, B=105, C=75, D=45, E="I hate C++"

Deviations from OO Paradigm

Three violations of object-oriented paradigm

1. this pointer not needed when member function calls another member function with same receiver

   Example — Recall find_element() code:

```
bool DLList::find_element(int x) {

  if (is_empty()) return false;          // lack of receiver makes is_empty()
                                         // look like file-scope function
  else {
    if (first_node->value() == x) return true;
    else return DLScan(x, first_node);  }     // again, no receiver
}
```


Deviations from OO Paradigm

Better (more readable) code for find_element()

```
bool DLList::find_element(int x) {
  if (this->is_empty()) return false;                  // yes!
  else {
    if (first_node->value() == x) return true;
    else return this->DLScan(x, first_node);           // yes!
  }
}
```


Second Deviation from OO Paradigm

2. Member function of class C can access data members of instances of class C other than receiver
   - Access restrictions applied by class, rather than by instance
   - A violation of information hiding principle?

Second Deviation: A Coding Example

- Suppose class Point defines distance method to compute Euclidean distance between receiver and an argument
- Possible code (recall x_ and y_ are protected in Point)

```
double Point::distance(Point& p1)  {
double x_diff = x_ - p1.x_ ;
double y_diff = y_ - p1.y_ ;
return sqrt(x_diff*x_diff + y_diff*y_diff) ;
}
```


Friend Functions and Classes

3. A class can declare other classes or functions as friends
   - A friend function f class C1 has access to private and protected members of C1 even if f is not a member of C1
   - If class C2 is friend of C1, all functions in C2 have access to private and protected members of C1

Motivation for Friends

Enhance flexibility of language by allowing access to representation of objects to nonclass members

— The classical example: Extraction and insertion operators for streams

— Redefining operators as member functions would cause wrong argument order in calls:

```
class C {
  public:
    C& operator<<(ostream& s); ... }
    C c1; }
....                                  // plausible client code
c1 << std::cout;                      // inconsistent with built-in << !!!!
```


Motivation for Friends

Solution: define stream operators as overloaded, file-scope friend functions

— Allow access to private and protected members of class instances

— Define appropriate syntax for calls

```
// this definition would define a nonmember (file-scope) operator
...
ostream& operator>>(istream& s, Point& inst_point);
ostream& operator<<(ostream& s, Person& inst_person);
```


Syntax of Friend Declarations

- Keyword friend followed by function prototype (or class declaration) for friend function or class

  Example — A file-scope DLList set intersection

```
class DLList {
friend DLList& DLList_intersect(DLList& list1, DLList& list2);
// DLList_intersect can access private and protected members
… }

// code for file-scope function DLList_intersect()
DLList& DLList_intersect(DLList& list1, DLList& list2) {

if (list1.first_node == NULL)                  // protected member access
 ... }
```


More on Friends

- Friend functions can be file-scope functions or members of other classes
- Friend class: give friendship privileges to all member functions of a class
- Friend declaration must be in body of class granting access privileges to friends
  - — Can be in private, protected or public portion
  - — Best placed at beginning of body
- Friendship is not inherited, nor symmetric, nor transitive

Discussion of Friends

- Useful when a function must access implementation of multiple classes directly
- The bad news about friends:
  - — violate information hiding when not used carefully
  - — friend functions cannot be dynamically bound (they are not part of class declaring it as friend)
  - — friendship is not inherited (good for information hiding, but friends must be redeclared for subclasses)
- Conclusion: Use friends only when really necessary, or lose some advantages of OO paradigm

Constructors and Destructors

- Big difference between class and nonclass instances:

  When class object allocated (deallocated), an appropriate constructor (destructor) is automatically invoked to initialize (finalize) new old instance_

- Constructors and destructors invoked regardless of whether instance allocated statically, dynamically or automatically
- Note: constructor (destructor) does not allocate (deallocate) memory for data members (new and delete do that, or the run-time system support)

Constructors

- Member functions having same name as their class
- Do not include (1) return type in header or (2) return statement in body; new instance is always returned
- Use parameters to specify initial values for newly created instance
- Often overloaded (for different kinds of initialization) and public (to allow use by clients)
- Typically, allocate space for data members that are pointers, initialize data members

Examples of Constructors

- Plausible constructors (inlined) for Point class:

```
class Point {
public:
  Point() { x_= 0; y_ = 0; }              //  default constructor, no args
  Point(int init_x, int init_y) { x_ = init_x; y_ = init_y; }
  Point(const Point& p1) { x_ = p1.x() ; y_ = p1.y(); }
  …
} ;
```

- The default constructor: A constructor without arguments (not necessarily generated automatically)

Examples of Constructors

- Plausible constructor definitions (.cpp file) for DLList:

```
DLList::DLList()  { first_node = NULL; size = 0; }

DLList::DLList(int x)  {
  first_node = new DLNode;
  first_node->value(x);              // Assume DLNode::value(int),
                                     // DLNode::previous(DLNode*),
  first_node->previous(NULL);  // DLNode::next(DLNode*)
  first_node->next(NULL);         // defined in DLNode
  size = 1;
}
```


Examples of Constructors

- Additional (copy) constructor for DLList (a deep copy)

```
DLList::DLList(const DLList& list)  {
  if (list.is_empty())    {                  // arg list is empty?
    first_node = NULL; size = 0;  }      // yes, do empty initialization
  else {   DLNode* node1; DLNode* node2;
    first_node = new DLNode;
    node1 = first_node;
    node2 = list.head();                  // DLNode accessor
    node1->previous(NULL);             // DLNode modifier
    node1->value(node2->value()) ;   // using overloaded
                                          // accessor + modifier of DLNode
    size = list.element_count();       // DLList accessor
    while (…)                              // while loop shown on next slide
    node1->next(NULL);  }
}
```


Examples of Constructors

While loop in constructor:

```
while (node2->next()) {
  DLNode* next_node = new DLNode;
  next_node->previous(node1);
  node1->next(next_node);
  node1 = next_node;
  node2 = node2->next();
  node1->value(node2->value()) ;
   }
};
```


Comments on DLList Copy Constructor

- Constructor does deep copy of other instance
- Deep copying a structure: Create another structure mirroring original (for structure and content)
  - No sharing of memory between two structures
- Shallow copying: Only elements at first level of original structure are copied
  - Objects referenced by elements at first level and beyond are shared between structures
- Shallow copying is often default behavior in C++

Constructor Invocation

- Syntax depends on whether instance allocated
  1. statically and automatically, or
  2. dynamically
- In general, argument list may be given with or without equal sign and class name
- Examples of constructor invocations (first case)

```
Point p1;                  // Use default const. Point::Point()
Point p2 = Point(p1);      // Use Point::Point(const Point&)
Point p3 = Point(10,10);   // Use Point::Point(int, int)
Point p4(20,20);           //  same as case of p3
Point p5(15);              //  error (argument mismatch)
```


Warnings on Constructor Invocations

- Don't let assignment syntax confuse you in definitions (with assignment)
  - Sign "=" appearing in identifier definition invokes constructor, rather than operator=
- Default constructor can't be invoked with empty argument list (without = syntax), e.g.,

```
Point p1() ;               // this declares a function p1 that returns
                           // a Point instance by value (copied)
Point p2(p1);              // This is OK, initialize p2 from p1
```


Constructors and Dynamic Allocation

- Follow new operator by suitable constructor invocation

```
Point* ptr_pt = new Point;              // use default constructor
DLList* ptr_list1 = new DLList;         // use default constructor
DLList* ptr_list2 = new DLList(10);     // use DLList(int)
DLList* ptr_list3 = new DLList(list1); //  use DLList(const DLList&)
```

- Explicit invocation of constructors in code also legal; (creates anonymous temporary object)

```
...
Point pt(…), temp_pt ;
temp_pt = Point(15,15);          // anonymous Point instance
pt.distance(temp_pt);            // allocated in run-time stack
```


Compiler-Generated Constructor

- Compiler generates automatically a default constructor only if programmer defines **no** constructors for a class
  - Call it the **default default constructor** to distinguish it from programmer-defined default constructors
  - Default default constructor is **public** by definition
  - Default default constructor often has an empty body, it just prevents a missing function definition error (In general, you should define your own)
  - Additional default actions (for embedded instances and base class instances) will be discussed later on

Compiler-Generated Constructor

- Riddle: Assume that class Point defines only following constructors:

```
Point(int x, int y);
Point(const Point& pt);
```

  - Then, what will this client code do?

```
...
Point pt;
pt.x(10);
pt.y(20);
...
```


Destructors

- Definition: Destructor for class C is the member function executed whenever an instance of C is deallocated (either automatically, or by invoking delete or delete[] operator, or upon program termination)
- Destructor executed at the end of object's lifetime
- Syntax of the destructor for class C
  - name is ~C
  - empty arg list
  - no return type or return statement

Destructors

- Typical destructor actions: Deallocate memory allocated by a constructor for class C, release I/O resources (close files, network connections, etc.)
- Destructor usually executed immediately before an instance (i.e., data members) is deallocated
- Not executed when free invoked (just like malloc does not invoke constructor)
- Not required when instance deletion only involves releasing memory used by data members alone (no data members are pointers)

Examples of Destructors

- Destructor for Point class not needed (no cleanup actions required when Point instance deallocated)
- If a class has no destructor, one is generated automatically by the compiler
- Compiler-generated destructor invokes destructors for data members (whenever a data member is a class instance)

Examples of Destructors

- Destructor for DLList class deallocates nodes

```
DLList::~DLList() {
if (first_node != NULL)
  this->recursive_delete(first_node);
}

// private function DLList::recursive_delete(DLNode&)
void DLList::recursive_delete(DLNode* a_DLNode) {
  if (a_DLNode->next() != NULL)
    this->recursive_delete(a_DLNode->next());
  delete a_DLNode;
}
```


Examples of Destructors

- When an array of instances deallocated, destructor is invoked once for each array element

```
DLList* list_array;    // an array of 10 DLLists
…

list_array = new DLList[10];  // DLList() constructor executed 10 times

…                      // work with 10 DLLists

delete[] list_array;   // ~DLList invoked 10 times (10x deep delete)
```


Dennis Ritchie (1941-2011)

- Invented C language
- Co-invented Unix OS (with Ken Thompson)
- Co-authored K&R C book (with Brian Kernighan)
- Turing Award recipient (1983) (with Ken Thompson)
- National Medal of Technology (1999) (with Ken Thompson)

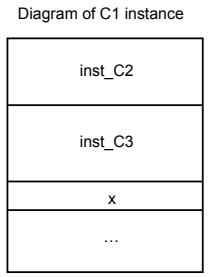
Constructor Initialization Lists

- Useful when a class C1 has instance variables that belong to other classes C2, C3, etc. (has-a relation)
- In this case, call C1 a composite class
- Key idea: Let data members that are instances of other classes be initialized by constructors in their classes (e.g., C2, C3, etc.)
- Invocation of constructors for class members is through constructor initialization list (special syntax to invoke constructors of other classes)

Diagram of class with embedded instances

Code:

```
class C1 {
 public:
  …
 protected:
  C2 inst_C2;
  C3 inst_C3;
  int x;
  …
} ;
```



Note: **inst_C2** and **inst_C3** must be value members

Syntax of Initialization Lists

- Initialization list is comma-separated list between head and body of constructor definition for class C
- For each data member to be initialized, place pair consisting of member name followed by parenthesized argument list in initialization list
- Argument list of a member must match parameter list of a constructor in the member' s class
- First list item introduced by a colon

```
C::C(<arglist>) : mem_1(<arglist_1>), mem_2(<arglist_2>),
                  mem_3(<arglist_3>)
 { /* body of C constructor */}
```


Example of Initialization List

- Suppose Person class has members of classes String and Account:

```
class Person {
 protected:
  String name_;
  long id_;
  long ssn_;
  Account& acct_;
  ... }
```

- Suppose that String has following constructors:

```
String::String();                 // default constructor
String::String(char*);            // initialize String from C-style string
String::String(const String&);    // copy constructor
```


Example of Initialization List

- Suppose that Account has following constructors:

```
Account::Account();                 // default constructor
Account::Account(const Account&);   // copy constructor
```

- Constructor for Person initializes name_ and acct_ members with constructors

```
Person::Person(char* aString, Account* ptrAcc, long SSN, long id)
      : name_(aString), acct_(*prtAcc) {
  ssn_ = SSN;
  id_  = id;
}
```


Details on Initialization Lists

- Non-class data members can also appear in list
  - In this case argument list contains one value only, of the appropriate built-in type
  - Data member is initialized with that value
  - A plausible redefinition of Person constructor

```
Person::Person(char* aString, Account* ptrAcc, long SSN, long id)
      : name_(aString), ssn_(SSN), id_(id), acct_(*prtAcc)
 { /* body is empty now */ }
```


Details on Initialization Lists

- What if class data member is omitted from initialization list?
  - Default constructor for that data member is invoked by default by the compiler
  - If class of data member does not have a default constructor, a compiler error will result!!!
  - Example: omitting name_ in initialization list above will invoke the default constructor for String
- What if class data member appears multiple times in initialization list? A compile-time error, of course!

Details on Initialization Lists

- If data member initialization omitted from list, one could do initialization in body of constructor
  - But this would probably waste the first initialization by the default constructors, e.g.,

```
Person::Person(char* aString, Account* ptrAcc, long SSN, long id)
            : acct_(*prtAcc)
{ // beware: must overload operator=() in String class
  name_ = String(aString);          // 2nd initialization!!!!
  /* other initialization actions /*
}
```


Order of Execution of Constructors

- Order of execution of constructors:
  1. Execute constructors in initialization list
  2. Execute body of constructor
- Order of execution of constructors in initialization list is determined by order of definition of data members in class C, not order of data members in initialization list
  - Deal with missing members using default constructor

Order of Execution of Constructors

- Example of order of execution of constructors in Person class

```
class Person     {          // possible class definition
  protected:
    String name_;
    long id_;
    long ssn_;
    Account acct_;  ... }

// execute String constructor first, even though acct_ is first in list
Person::Person(char* aString, Account* ptrAcc, long SSN, long id)
: acct_(*prtAcc), name_(aString)
{ ... }
```


More on Execution Order

- If class C1 also has data member of class C2, and C2 has data member of class C3, constructor for C3 is executed first, followed by constructor for C2, followed by (remainder of) constructor for C1
  - Instance of C1 built from the "inside out"
- Order of execution of destructors is the order of execution of constructors reversed
  - Instance of C1 destroyed from the "outside in"

More on Initialization Lists

- Initialization lists are needed for initializing **const** data members, and **data members that are references**
- Example of const and reference member initialization

  Suppose Person definition is changed to:

```
class Person {
  protected:
    String name_;
    const long id_;
    Account& acct_;  ... }
```


Examples of Member Initializations

- Initialize (nonstatic) const member through initialization list:

```
Person::Person(char* aString, Account* ptrAcc, long SSN, long id)
        : acct_(*prtAcc), name_(aString), id_(id)
{ ... }
```

- Recall that const identifiers cannot appear in left-hand side of assignments
  - Impossible to initialize const data member in constructor' s body

The Copy Constructor

- Copy constructor for class C: Constructor executed automatically whenever a C instance must be copied
  - One copy constructor (at most) for each class
- Automatically invoked in following cases:
  1. Argument of a value parameter passed to function
  2. Function returning a value (not a reference or a pointer) back to its invoker
  3. Instance initialized from an existing instance

  Existing instance can be temporary (anonymous) object (e.g. on run-time stack)

Syntax of Copy Constructors

- Usual constructor syntax with one parameter of type const C&
- Example: Copy constructor of class String

```
String::String(const String& s);
```

- If not specified by programmer, compiler-generated copy constructor will perform member-wise copying (shallow-copy)
- Compiler-generated copy constructor usually adequate when class does not have any pointer members

Copy Constructors

- Riddle: Suppose that you defined a copy constructor with a value parameter. What would happen?

Default Copy Constructor

- Compiler generated copy constructor takes two actions (excluding inheritance):
  1. (built-in type data members): Copied one-by-one from argument to receiver
  2. (class data members): Invoke copy constructor for class of data member (hopefully public)
- Often avoids need for programmer-defined copy constructor in composite classes

Example of Default Copy Constructor

- Assume following definition of Person class:

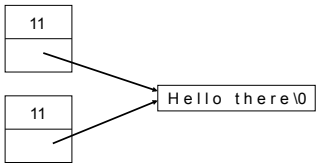
```
class Person {
protected:
  String name_;
  long ssn_ ;
  long id_ ;
  Account* ptrAcct_ ;  … }
```

- Default copy constructor would do following:

```
Person::Person(const Person& p1) : name_(p1.name_)
 { ssn_ = p1.ssn_ ;
   id_ = p1.id_ ;
   ptrAcct_ = p1.ptrAcct_; }
```


Copy Constructor Actions

- When C has pointer members, programmer-defined copy constructor may be necessary
  - Should pointer in two instances share referents?
- Likely effects of compiler-generated copy constructor for String class:



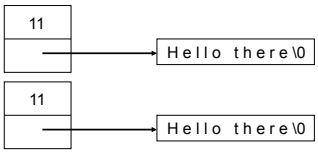
Copy Constructor Actions

- Beware of unintended effects of default copy constructor
  - Changes made to one string instance will affect other instance
  - Also, upon deletion of one instance, space deallocation may result in a dangling pointer
- Sometimes referents should be shared, sometimes not
  - Is referent an independent object, loosely coupled with object being copied? (then, don't copy)
  - Or is referent really part of object being copied?

Examples of Copy Constructors

- Copy constructor for String class: Copy char array

```
String::String(const String& s)  {
  size_ = s.size_ ;
  string_ = new char[s.size_ + 1] ;
  strcpy(string_, s.string_) ;  }
```



Examples of Copy Constructors

- Disclaimer: You don't always need to copy referent
- Example: A Widget class with a member pointing to Window instance
  - If widget copied, don't replicate Window instance

More on default default constructor

- Recall default default constructor: Default constructor (no args) automatically generated by compiler when programmer specifies no constructors
- As we saw, default default constructor has empty body
- However, default default constructor will implicitly invoke default constructors of embedded class instances in instance being initialized
  - Similar behavior to default copy constructor

Static Members

- Static data member and member functions are class variables and methods in C++
- Static data members:
  - one variable shared by all instances of a class
  - scope = defining class; access restrictions apply
  - lifetime is entire program execution (allocated in static space)
  - initialized immediately before program execution
  - finalized upon program termination

Syntax of Static Data Members

- Declared in class definition (header file), defined in class implementation (code file)
- Syntax

```
class C1 {
  …
  static <type_spec> <id> {, <id>};
}
```

- Example (.h file):

```
static int tally;          // be careful: usually not initialized in class definition
```


Syntax of Static Data Members

- Example of definition (with initialization) in code file:

```
int C1::tally = 0;
```

- If static member belongs to a class, initialization causes execution of constructor before program starts execution

```
Point Point::origin = Point::Point(0,0);          // Point(int,int) invoked
```

- Note: Definition and initializations in code file not subject to access restrictions (Above statements are legal even if origin and tally are private in C1 and Point)

More on Static Data Members

- Similar to global variables, except scope is defining class and access is restricted by defining class
  - Directly accessible within class of definition
  - Access by clients and subclasses depends on whether member is public or protected
  - When accessible outside defining class, use qualified reference (either class name or access through instance):

```
C1::tally;          // good client code, if tally public in C1
inst_C1.tally;      // legal, but not desirable client code (confusing)
```


More on Static Data Members

- Modifiable by const member functions (static data members are not considered part of an instance)
- Typically, hold default values in instance initialization, class statistics, or other class-dependent information
- Can be part of defining class:

```
class Point {
…
static Point origin;        // the declaration of origin in Point
} ;
```


not covered

Static, Const Data Members

- Static, const data members of integral type (e.g., int, long, short) can be initialized in class definition
- Ugly syntax makes declaration look like a definition:

```
class C1  {
…
   static const int month_number = 12;     // still a declaration
   static int tally = 0 ;                  // illegal: tally not const
   static const double pi = 3.1416;        // illegal: pi not integral
} ;
```

- Data member must still be defined (without repeating initialization) in code file (ugh!)

Static Member Functions

- Static member functions = class methods of C++
  - Member function operating on class and class variables rather than an instance
  - No direct access to non-static data members
  - Typically, interface to private and protected static data members, no this pointer
  - Directly accessible within class of definition
  - Static function identifier is subject to private, protected, and public access specifications

Static Member Functions

- Syntax: Keyword static not necessary in function definition (only in declaration), e.g.,

```
class Point {
public:
  static int NumberOfPoints();      //  declaration of static function
protected:
  static int tally_ ;               // declaration of static data member
…  }

// definitions in code file
int Point::tally_ = 0;              // initial value of tally_
int Point::NumberOfPoints() {       // function code
 return tally_;  }
```


Access to Static Member Functions

- Outside defining class, static function is accessible with scope operator, or through a class instance (first syntax is preferable, though)

```
Point::NumberOfPoints();        // good syntax
point1.NumberOfPoints();        // misleading?
```


Class Header Files

Don't forget to include the following in header files:

- Class definition (header and body)
- In class body: member function prototypes (at least), data member declarations (not definitions), all under appropriate access level specifications
- Private members included for compiler to know how much space to allocate to value instances of class
- If class definition changed, all clients must be recompiled because of potentially different space requirements

Class Header Files

Also, don't forget the following in header files:

- #include's for needed typedefs, enums, and constants needed for the class definition
- #include's for classes needed by the class definition, e.g.,
  - classes defining inline member functions invoked in this class
  - superclass(es) of this class
  - classes used by this class

Avoid Multiple Class Definitions

- As header files must be included, there is a risk that some classes may be multiply defined
  - Suppose class B includes C, then class A includes B and C...
- Avoid multiple definition errors with ifndef, endif syntax:

```
#ifndef SET_H
#define SET_H
class Set {
/* ... */}
#endif
```


Nested Classes

- Classes need not be defined at file (global) scope
  - Class can be nested within a class (even within a function, but not very useful)
- Nested classes useful for hiding local classes from global scope
  - Normally, classes defined at file scope level: **Class identifiers are typically defined in global scope**
  - Nested classes avoid conflicts on class names by defining class within scope of containing class

Example of Class Nesting

- Code skeleton for class nesting:

```
class Outer {
   …
   class Inner {
      …
   } // end of Inner
   …
} // end of Outer
```

- Main consequences are on visibility of (1) identifier Inner and (2) member identifiers of both classes
  - **Scope rules based on nesting of units within units apply to nested classes**

Example of Useful Nested Classes

- Suppose two classes, BinaryTree and LinkedList, require different Node classes, for tree and linked-list nodes
- Solution 1: Use different names for two node classes (e.g., LLNode and TreeNode)
- Solution 2: Two classes with same name (e.g., Node), each nested within LinkedList or BinaryTree
  - In this case, no name conflict because two Node identifiers defined in different scopes (e.g., class BinaryTree and class LinkedList)

Definition of Nested Classes

- Nested classes can appear in private, protected, or public part of enclosing class
  - Determine access level of nested class' s identifier with respect to clients and subclasses of enclosing class

    Earlier example: Should Node identifier be private, protected or public in BinaryTree and LinkedList?

Typical Access Levels for Nested Classes

- Typically, nested class is defined in protected part of enclosing class, to prevent clients of enclosing class from getting access to nested class
  - Clients of enclosing class cannot access protected identifier of nested class
  - Another advantage: Nested class can declare all member functions and data members to be public, to allow full access to enclosing class, without violating information hiding outside the enclosing class

Example of Nested Class Definition

- Nest protected class Node within BinaryTree class:

```
class BinaryTree {
  protected:
    class Node {

      public:
      // methods that BinaryTree uses

      protected:  // public?
      // members hidden from BinaryTree if needed, otherwise public

    }  // end of Node
...
} // end of BinaryTree
```


More on Nested Class Definitions

- Nested classes defined in public part of enclosing class can be accessed outside enclosing class through nested scope operator:

  outerClass::innerClass::memberId
- Example - This time suppose that two Node classes defined in public portions of BinaryTree and LinkedList
  - Client can use both Node classes:

    BinaryTree::Node node1;
    LinkedList::Node node2;
  - Two Node identifiers do not conflict because they are defined in different scopes

More on Nested Class Definitions

- Code of nested class can see ids defined in enclosing class, but no access or inheritance relationship implied

  (e.g., enclosing class cannot see protected members of nested class and vice versa)
- "Nested" position gives visibility, no access, to identifiers of outer classes
- Recall that **visibility** and **access** are distinct concepts in C++

Access vs. Visibility: A Riddle

- Recall that member identifier x can be used in scope S if:
  1. x is visible in S, and
  2. x is accessible in S
- Then, what happens here?

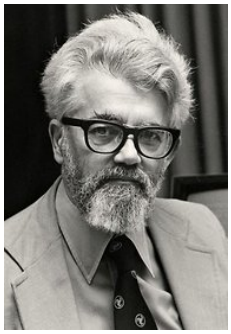
```
int x=0;                          // file scope definition
class List {
 protected:
   char* x;                       // class scope definition (List class)
   class Node {
   public:
     int m1 () { return x++; }    //  question: which x?
... } ... }
```


Tips on Use of Nested Classes

- Can promote module independence
  - Hide some classes from global scope
- Typical use:
  - Nested class defined in protected part of enclosing class
  - Nested class's members defined in public part of class to allow access to enclosing class
  - Deny access to clients of enclosing class

John McCarthy (1927-2011)

- Inventor of LISP (1959)
- Cofounded AI lab at MIT (1958) (with Marvin Minsky)
- Codesigned 1st time-shared OS
- Founded SAIL lab at Stanford (1968)
- Turing Award recipient (1972)
- Kyoto Prize (1988)
- National Medal of Science (1991)



Read Chapters 9 and 10 in Drake's book!

Operator Overloading

- Only predefined operators for user-defined classes:
  1. **=** (assignment)
  2. **.** (component selection)
  3. **->** (component selection with dereferencing)
  4. **&** (address of)
  5. **new** (instance creation)
  6. **delete** (instance deletion)

Operator Overloading

- C++ allows built-in operators to be overloaded by classes
- Examples:
  - Define + (addition) operator in Matrix or Complex class
  - Define == (equality) operator for any class
  - Redefine = (assignment) operator to avoid sharing of structures, as with copy constructors (default operator does member-wise copying)
  - Others possible: Stream (<<, >>), relational (>, >=, <, <=), (indexing) [], etc.

Syntax of Operator Overloading

- Syntax: use keyword operator before operator symbol, eg.,

```
operator=(const String&);
operator<=(const String&);
```

- Overloaded operator can be member or file-scope function

  E.g., two ways to define string concatenation:

```
// member function in String
String& String::operator+(const String& rhs);

// file-scope function
String& operator+(String& lhs, const String& rhs);
```

- Operators = Functions with shorthand invocation syntax

Syntax of Operator Overloading

- File scope and member style operators are not equivalent (e.g., C++ will not convert a numeric receiver)
- Semantics of redefined operator should be consistent with semantics of language definition for operator
- For a numeric class, **+** should create a new instance; however, **+=** should modify directly an existing operand (the left-hand side)

Examples of Operator Definitions

- A Fraction class to handle fractional numbers (Smalltalk inspired, somewhat)

```
class Fraction {
  public:
    Fraction();                          // default constructor
    Fraction(int x, int y);              // another constructor
    Fraction(int x);                     // conversion from integer
    bool operator==(const Fraction& operand2) const;
    Fraction operator+(const Fraction& operand2) const;
    Fraction& operator+=(const Fraction& operand2);
    Fraction operator*(const Fraction& operand2) const;
    …                                    // other definitions
  protected:
    int numerator;
    int denominator;  } ;
```


Fraction Example Continued

- Note: No direct accessors and modifiers, no explicit copy constructor and assignment operator (default versions are OK)
- Code for Fraction constructors

```
Fraction::Fraction() {          // default constructor
  numerator = 0 ;
  denominator = 1 ;  }

Fraction::Fraction(int x, int y) :  numerator(x), denominator(y)
      { }

Fraction::Fraction(int x) : numerator(x), denominator(1) { }
```


Additional Fraction Method

```
bool Fraction::operator==(const Fraction& operand2)  const {
  if (numerator*operand2.denominator ==
          denominator*operand2.numerator)
  return true;
  else return false;
}
```


Additional Fraction Methods

```
// operator+ must return a new instance (can only be done by value)
Fraction Fraction::operator+(const Fraction& operand2) const {
  return Fraction((numerator*operand2.denominator +
                   operand2.numerator*denominator),
                  denominator*operand2.denominator);
}

// operator+= must return by reference, not value: Avoid copying of
// Fraction instance that holds result, by returning it directly
Fraction& Fraction::operator+=(const Fraction& operand2)  {
  numerator = numerator*operand2.denominator +
                   operand2.numerator*denominator ;
  denominator = denominator*operand2.denominator;
  return *this;
}
```


Operator Invocation

- Traditional function syntax possible, along with short-hand notation

```
Fraction f1(1,3);
Fraction f2(1,2);
Fraction f3;              // default constructor invoked
f3 = f1 + f2;             // invokes Fraction::operator+
f3 = f1.operator+(f2);  // different syntax, same effects as above
```

- Cascading can be used too, e.g.,

```
f1 += f2 += f3 ;          // now f1 holds 5/3 and f2 holds 4/3
```

(assignment operators are usually right-associative)

More on Operator Overloading

- One more tip: operator[] should return reference to element of composite object (to be usable as l-value)
- Example of such an operator for String class

```
char& String::operator[](const int i) const {
if (i < 0 || i >= size_)
... // error processing
else return string_[i];  }   // recall string_ defined as char*
```

- Example of use of operator[] for String class

```
String s1( ‘ ‘hello’ ’ );     // assume suitable constructor
s1[1] = ‘u’ ;                 // now s1 == ‘ ‘hullo’ ’
```


Limitations of Operator Overloading

- Cannot create a new operator
  - Only predefined operators can be overloaded
  - (e.g., cannot define operator** for exponentiation)
- Cannot change operator precedence, arity and associativity
  - Syntax would be confusing and parsing would be too expensive
- At least one argument must be of class type
  - E.g,. cannot redefine / between int's to return Fraction instance

More on Operator Overloading

- When overloading **++** and **--** use arg convention to distinguish between prefix and suffix operators
- First argument (or receiver, for member function defined operator) must be of class type
- Overloading prefix operator (as member function):

```
Fraction operator++();
```

- Overloading suffix operator (also as member function):

```
Fraction operator++(int i);    //argument i is ignored
```


File-Scope vs. Member Operators

- Operators can also be defined as (overloaded) file-scope functions (not as member functions)
- Two kinds of definitions are **almost** equivalent
- Example: File-scope definition of Fraction::operator+()

```
//  assume operator+(const Fraction&, const& Fraction) is declared as
//  friend in Fraction class
Fraction operator+(const Fraction& lhs, const Fraction& rhs) {
   return Fraction((lhs.numerator*rhs.denominator +
                    rhs.numerator*lhs.denominator),
                    lhs.denominator*rhs.denominator);
}
```


File-Scope vs. Member Operators

- Difference: cannot convert receiver in operator invocation (member-function operator in this case)
- Recall integer conversion operator defined for Fraction:

```
Fraction::Fraction(int x) : numerator(x), denominator(1)
  { }
int x = 3;
Fraction f1(1.0,2.0), f2;
f2 = f1 + x;    // works with both kinds of operators
f2 = x + f1;    // ERROR when operator+() defined as member function:
                //  integer addition invoked in this case.
```


File-Scope vs. Member Operators

- String concatenation would work similarly with conversion from char*
- Conclusion: Defining arithmetic operators as overloaded file-scope functions (as opposed to member functions) is probably a good idea

The Assignment Operator

- Used whenever a value identifier of given class is on left-hand size of an assignment
- Must be defined as member function, not file scope operator
  - Ensure that lhs is a class instance
  - Few other operators have similar constraint, i.e., cannot be defined as file scope operators (e.g., operator(), operator[], operator->)

The Assignment Operator

- Default (compiler-generated) assignment operator: Copy each member of rhs to lhs
  - also invoke assignment operator of embedded instances for composite classes
  - similar behavior to default copy constructor
  - again, may not be OK when pointers are copied, because of structure sharing between lhs and rhs

An Example of Assignment Operator

- Example of assignment operator requiring space allocation and deallocation

```
String& String::operator=(const String& rhs)  {
  size_ = rhs.size();                  // copy the size from rhs
  delete[] string_;                    // deallocate old char array
  string_ = new char[size_ + 1];       // allocate space for new char array
  strcpy(string_, rhs.string());       // copy chars from rhs
  return *this;                        // return reference to receiver
}
```

- Very similar to copy constructor for String (can you spot the difference?)
- But what if rhs and receiver are the same?

An Example of Assignment Operator

- Corrected assignment operator:

```
String& String::operator=(const String& rhs) {
  if (this == &rhs) return *this;       // if same rhs and lhs, do nothing
  else {
    size_ = rhs.size();
    delete[] string_;
    string_ = new char[size_ + 1];
    strcpy(string_, rhs.string());
    return *this;   }
}
```


not covered

Overloading new and delete Operators

- Programmer is allowed to overload **new** and **delete** operators to implement class-specific memory management
- Example: Linked-list that keeps own free list containing deallocated nodes
  - **delete** would put nodes on list
  - **new** would look in list before going to general heap
  - Programmer-defined operators invoked as usual
- An advanced capability to be used very rarely and carefully

Overloading **new** Operator

not covered

- One arg of type size_t, defined in stddef.h
- Argument specifies size of instance in bytes
- Argument not specified by programmer (operator new has no args), filled in automatically by compiler
- Return type is void*
- Code declaring overloaded new operator

```
// built-in type size_t defined in library stddef.h
// return value is pointer to newly allocated node
void* operator new(size_t) ;   // arg is number of bytes
```


Overloading **delete** Operator

not covered

- Two parameters, void return value
  1. void* (starting location)
  2. size_t (size of deallocation)
- Declaration of delete operator

```
// void parameter is location to be deleted,
// size_t is size of deallocation
// no return value
void operator delete(void*, size_t) ;
```


Coding Node Class

not covered

- Linked-list class that keeps free list explicitly
- Node class redefines new and delete with versions working off user-defined free list (as opposed to system heap)

Coding Node Class

not covered

- File Node.h

```
#include stddef.h
Class Node {
  friend class Queue;
  friend ostream& operator<<(ostream&, Queue&);
  protected:
    Node(const ElemType& el) ;
    void* operator new(size_t);
    void operator delete(void*, size_t);
    ElemType elem;
    Node* next;
    static Node* pFreeList;
    static const int regionSize;
} ;
```


File Node.cpp

not covered

```
Node* Node::pFreeList = NULL;     // initialize static data member
const int Node::regionSize = 32;     // initialize static const member

void* Node::operator new(size_t size) {
  Node* pList;
  if (pFreeList == NULL)  {              // free list empty?
      // in this case, allocate new block and link it
      pFreeList = (Node*) new char[size * regionSize] ;
      for (pList = pFreeList; pList != &pFreeList[regionSize -1]; pList++)
          pList->next = pList + 1;
      pList->next = NULL;   }
  // now remove node from list
  pList = pFreeList;
  pFreeList = pFreeList->next;
  return pList;
}
```


Coding **delete** Operator

not covered

- Definition of **delete** operator in Node.cpp

```
void Node::operator delete(void* pNode, size_t size)  {
    // prepend new node to existing free list for later use
    ( (Node*) pNode)->next = pFreeList;
    // correction from Drake's book
    pFreeList = (Node*) pNode;
}
```


not covered

Smart Pointers

- Overload operators for component selection and dereferencing: operator*(), operator->()
- Idea: If class C1 redefines, say, operator->, then every time a client uses message expression using this operator (e.g., instC1->msg1()), programmer-defined member function, rather than built-in operator-> is used

not covered

Potential Uses of Smart Pointers

- Suppose instances may be stored on disk or in memory

  Then smart pointers could be used to retrieve items from disk if not already present in memory
- Provide access to distributed objects
- Transparent access to persistent objects (a la BOSS)
- Keep reference count to instances (to support automatic garbage collection)

not covered

Implementation of Smart Pointers

- Problem: Arg of -> operator is method name, cannot be passed as parameter

  C++ does not define type for method names
- Solution: Define operator-> with no args, interpret expression with programmer defined operator->:

  smtPtrInst->msg(arg1,arg2)

  as follows:

  (smtPtrInst.**operator->()**)**->**msg(arg1,arg2)
- Compiler generates code that (1) applies programmer-defined operator, (2) dereferences result, and (3) sends right-operand message to referent

not covered

Implementation of operator->()

- Unary postfix operator
- Must return pointer to referenced object
- Must be defined as a member function, not at file scope
- Illegal to delete instance directly, only through smart pointer

not covered

Typical use of operator->()

- Use wrapper class that maintains smart pointers to "protected" objects
- Example: Smart Caching of Person instances

  Use PersonPtr class to access Person instances, swap in and out of memory

not covered

Class PersonPtr

```
class Person;                       // forward declaration

class PersonPtr {
public:
  PersonPtr(long);                  // get a Person handle via SSN
  ~PersonPtr():                     // finalize handle (write Person obj to file)
  Person* operator->() const;       // send message to Person object
  Person& operator*() const;        // obtain reference to Person object
protected:
  long ssn;
  Person* pPerson;                  // local copy of Person (NULL if swapped)
  PersonPtr(const PersonPtr&);  // no copying
  PersonPtr& operator=(const PersonPr&)   // no assignment operator
  static PersonFile personIndex("personIndexData");
  static int CACHE_SIZE;
  static PersonCache cache(CACHE_SIZE);  };
```


not covered

Code File for PersonPtr

```
#include "PersonPtr.h"

inline PersonPtr::PersonPtr(long socSecNum)
   :  ssn(socSecNum), pPerson(NULL)
   { }

inline Person* PersonPtr::operator->() {
  if (pPerson == NULL)
     pPerson = cache.loadFromFile(ssn, personIndex, pPerson);
  return pPerson;  }

inline Person& PersonPtr::operator*() {
  if (pPerson == NULL)
     pPerson = cache.loadFromFile(ssn, personIndex, pPerson);
  return *pPerson; }
```