CS 1B Review Part 1

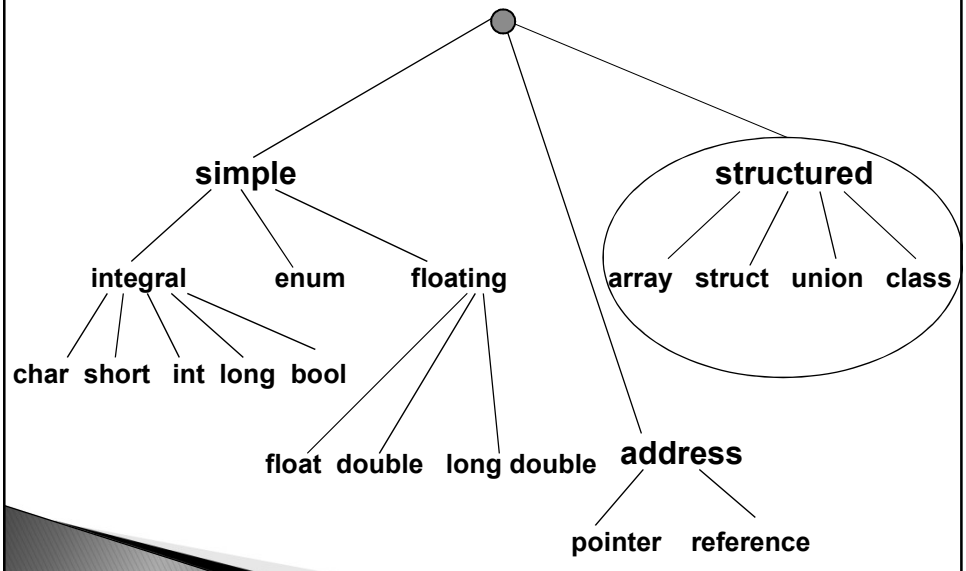
Topics - 1

- Enums
- Typedefs
- String datatype
- Arrays
- Cstrings
- Multi-dimensional arrays
- Binary search
- Structures
- Casting

Topics - 2

- Classes
- Class Constructors
- Destructors
- Helper functions
- Information hiding
- Array of objects
- Static data members
- Const parameter

C++ Data Types



Enumerated Types

- Used to increase readability and maintainability
- Enumerated types are used to declare a set of integer constants
- Syntax:

  enum [tag]
  {comma separated list of identifiers} [variable-list];

Enumerated Type Example

```
enum trees
{
  oak,
  maple,
  cherry
};//  no variables declared yet
enum trees myTree; // declares a variable of type enum trees
```

- See example enum1.cpp

ENUM
Data Type

Typedefs

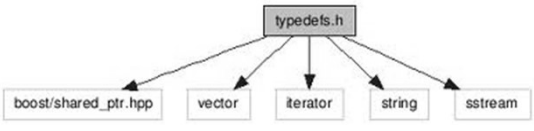
- Used to associate an identifier with a type (no storage allocated)
- Enhances readability and maintainability
- Allows programmers to use types that are appropriate to the application
- Syntax:
  - typedef oldType newType

Typedef Example

- Example:

```
typedef int color;
// color is now a type
color red, white, blue
```

- See example enum1.cpp

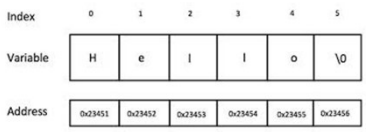


String Class (1)

- String class
  - A class supplied by many compiler vendors
  - Not part of the language
  - Need to #include <string>
  - Cannot always be used (open of files requires a C string)
  - Using the String class can eliminate many of the problems associated with Cstrings
  - Has over 100 members

String Class

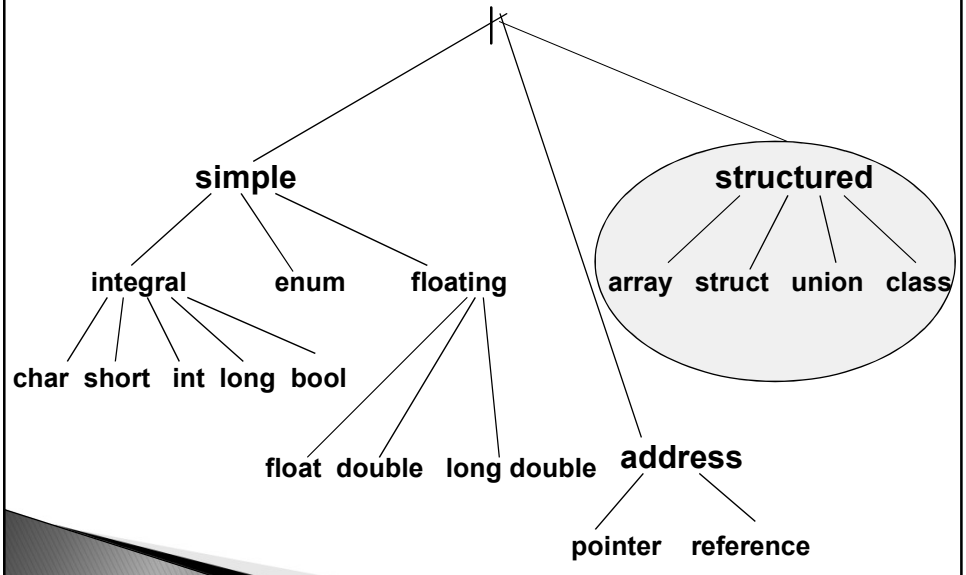
- String class
  - Memory is dynamically allocated when needed
  - Many operators are overloaded
    - + << >> [ ]
  - Has a default constructor that initializes a string object to A NULL string
  - Has another constructor that takes a parameter, creates a string object, and sets it to the parameter
  - Boundary checking member function available



String Class

- Can create an array of string variables
- String myStrings[30];
  - myStrings[2] = “Hello World” ; // accesses third string
- (See examples: string1.cpp through string3.cpp)

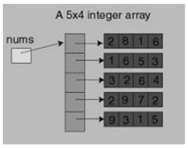
C++ Data Types



Arrays -1

- Used to store multiple values of the same datatype in one variable name
- Stored contiguously in memory
- Individual elements can be accessed using a subscript called an index
- Syntax:

  datatype arrayName [[numberOfCells]];

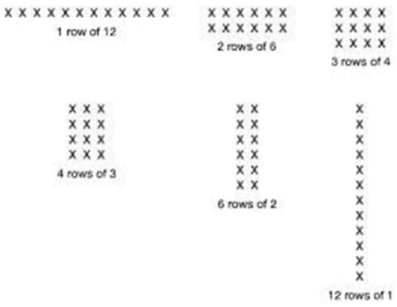


Arrays - 2

- Valid indices are 0 through numberOfCells-1
  - Integers (int, char, bool, short, long, enum, unsigned short, unsigned long, unsigned int) expressions must be used for the index
  - C++ does not do boundary checking (not a compilation error)
    - Given the allocation: int testScores[1000]; - cout << testScores[1000] ; returns an unpredictable value
    - This is a very common problem using loops (for, while, do)

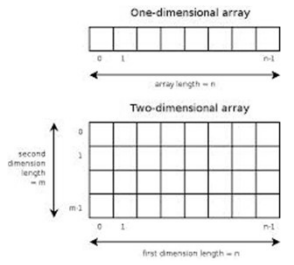
Array Size

- Given the following allocation (assuming 32 bit integers)
  - Int testScores[100];
    - sizeof(testScores) would be 400
    - sizeof(testScores[0]) would be 4



Array Initialization

- int testScores[10] = {100,89,99}
  - elements 3-9 will be initialized to zero
- float dollars[ ] = {12.64,3.99,97.82}
  - # of cells equals the number of initial values
- The contents of a cell is unpredictable if not initialized



Array Addresses

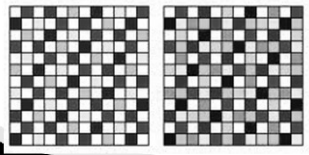
- Given int arrayName[10]
  - If arrayName [0] is at address 1000 then arrayName[1] will be at address 1004 (assuming 32 bit integers)
- The name of the array without brackets evaluates to the address of the first cell in the array
- The name of the array without brackets is a pointer (because its value is an address)
  - The array name is equivalent to &arrayName[0]
- (See example: array1.cpp)

Aggregate Operations on Arrays

- There aren't any EXCEPT aggregate I/O is permitted for Cstrings (special kinds of char arrays)
- See examples array2.cpp and array3.cpp

Arrays as Parameters to Functions -1

- Arrays can be passed as parameters to functions but in contrast to other variable types, it is not possible to pass a copy of the array
  - Instead the address of the array name serves as the parameter and the function can access the array elements through the address

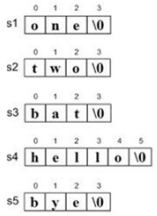


Arrays as Parameters to Functions -2

- Functions can change the array elements
  - The array name is the address of the array (address of the first cell)
  - Function has no knowledge of the number of elements or type
  - Generally, functions that work with an entire array require two items of information as arguments:
    - The beginning memory address of the array (base address)
    - The number of elements to process in the array
- To prevent a function from changing an array use the word “const” in the function heading and prototype
- See example: array4.cpp

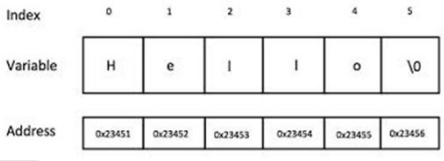
CStrings – 1

- Cstrings or character strings
  - NOT the string class
  - Is a char array terminated by the null character '\0' (with ASCII value 0)
  - char cstring[10]; - can store 9 characters
  - A Cstring constant is constant enclosed within double quotes "ABC"



CStrings –2

- A Cstring variable can be initialized in its declaration in two equivalent ways
  - char helloMessage [6] = { 'H', 'e', 'l', 'l', 'o', '\0' };
  - char helloMessage [6] = "Hello" ;
- char helloMessage [6];
- helloMessage = "Hello" // will give a compilation error since one is trying to change the address of the array helloMessage
- 'A' is data type char and is stored in 1 byte
- "A" is a C string of 2 characters and is stored in 2 bytes



More on CStrings

- Passing CStrings
  - C++ associates a string constant with the address of the 1st character
    - myFunction("I am here"); address of I is passed to myFunction
- Concept of strings related to character arrays (stored in contiguous memory locations)
  - BIG difference - strings end with a NULL while a character array may or may not

Even More on CStrings

- Aggregate C String I/O in C++
  - I/O of an entire C string is possible using the array identifier with no subscripts and no looping (like the %s in C)
  - This cannot be done with any other data type (a loop is required to input an integer array)
  - cout expects a NULL in the array
    - It will send characters to the output stream until it finds one

More on Cstring I/O -1

- When using the extraction operator (>>) to read input characters into a Cstring variable, the >> operator skips any leading whitespace characters such as blanks and newlines
  - It then reads successive characters into the array, and stops at the first trailing whitespace character (which is not consumed, but remains waiting in the input stream)
  - The >> operator adds the null character to the end of the string when it is stored in memory

More on Cstring I/O - 2

- If the string's declared size is not large enough to hold the input characters and the added '\0', the extraction operator stores characters into memory beyond the end of the array
  - Is this a problem?
- Another function is required to read whitespaces
- (See examples cstring1.cpp and cstring2.cpp)

Functions Requiring Cstrings

- Some functions require C strings instead of C++ string objects
  - I/O related functions (open, close, etc.)
- (See example: cstring3.cpp)

Cstring Library Routines – 1

- Functions are needed for the = and == operators
  - strcpy and strcmp
- strlen(string) - returns size_t (unsigned int) the length of a string (not including the \0)
  - length=strlen("I am here") returns 9
- strcpy(copy,original) - copies a string - copy must be big enough to hold original - returns the address of copy

Cstring Library Routines

- strcat(buffer,string)- concatenates strings - buffer must be big enough to hold buffer and string - the NULL of buffer is overwritten
- strcmp(string1, string2) - compares two strings –
  - returns 0 if string1 is identical to string2
  - returns a negative value if string1 < string2
  - returns a positive value if string1 > string2
- Runtime errors (memory overwritten) can occur when not enough storage is allocated
- (See examples cstring4.cpp and cstring5.cpp)

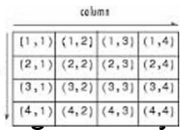
Multi-Dimensional Arrays

- Two dimensional arrays are related to matrices (board games, computer screen)
- A two dimensional array is a collection of components, all of the same type, structured in two dimensions, (referred to as rows and columns)
  - Individual components are accessed by a pair of indexes representing the component's position in each dimension
- For example:

  1 2 3

  4 5 6

  2 rows and 3 columns



- Two-dimensional arrays are simulated by ha of arrays
  - This means having an array where each cell in the array is an array

Multi-Dimensional Arrays

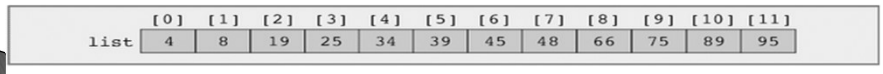
- Syntax:
  - DataType ArrayName [ConstIntExpr] [ConstIntExpr] . . . ;
- Example:
  - float twoDim[3][4] ;
- C++ stores arrays in row major order
  - The first row is followed by the second row, etc.
  - (See example: marray1.cpp)

Passing Two Dimensional arrays

- Just as with a one-dimensional array, when a two - (or higher) dimensional array is passed as an argument, the address of the caller's array is sent to the function
- The size of all dimensions except the first must be included in the function heading and prototype
- (See examples: marray2.cpp and marray3.cpp)

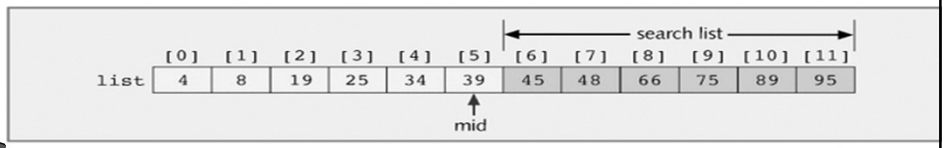
Binary Search

- A binary search is similar to a dictionary search
- A binary search uses the “divide and conquer” technique to search the list
- First, the search item is compared with the middle element of the list.
- If the search item is less than the middle element of the list, we restrict the search to the upper half of the list; otherwise, we search the lower half of the list.
- Consider the following sorted list of length = 12

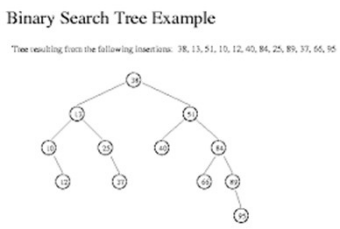


Determine if 75 is in the List

- **Compare 75 with the middle element in the list, list[5] (which is 39).**
- **Because 75 ≠ list[5] and 75 > list[5], we then restrict our search to the list list[6]...list[11]**



Binary Search Code




```
int binarySearch(const int list[], int listLength,
                                 int searchItem)
{
    int first = 0;
    int last = listLength - 1;
    int mid;
    bool found = false;
    while(first <= last && !found)
    {
            mid = (first + last) / 2;
            if(list[mid] == searchItem)
            found = true;
            else
              if(list[mid] > searchItem)
                              last = mid - 1;
              else
                              first = mid + 1;
    }
if(found)
      return mid;
    else
      return –1;
}//end binarySearch  (see example bsearch.cpp)
```

Casting

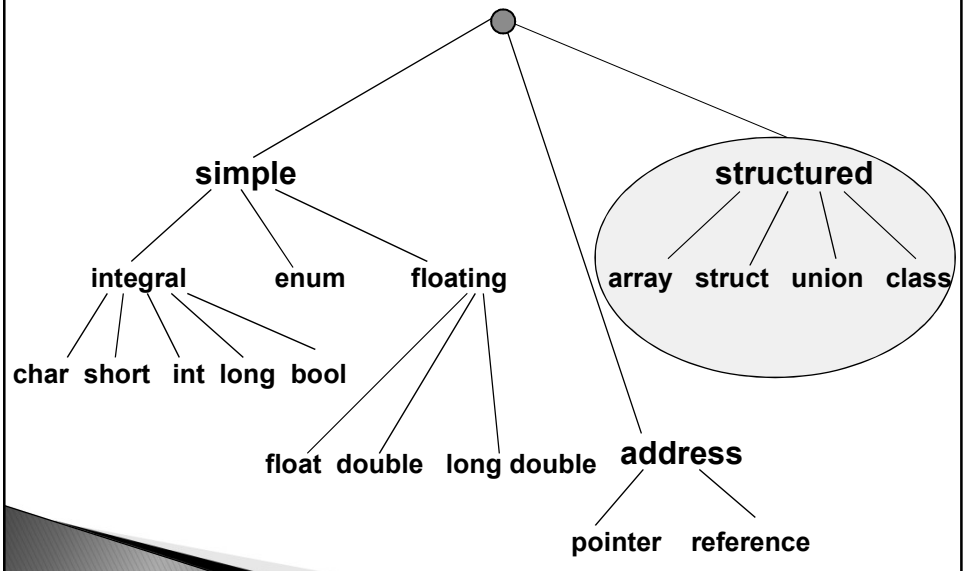
- Without casting C++ performs implicit type coercion
- Cast operator
  - An expression is evaluated and then converted
  - static_cast<dataTypeName> (expression)

See cast1 example

Casting with enums Example

enum sports {BASKETBALL, FOOTBALL, HOCKEY};
sports popularSport;
popularSport = FOOTBALL; // legal
popularSport++; // is illegal (no arithmetic operation is legal)
popularSport = popularSport +1 ; // is illegal
popularSport=static_cast<sports>(popularSport+1) // changes PopularSport to HOCKEY

C++ Data Types



Structured Data Types

- A structured data type is a type in which each value is a collection of component items
- The entire collection has a single name
- Each component can be accessed individually
- Often related information of various types is store together for convenient access under the same identifier
- Arrays, structs, and classes are C++ structured data types

`struct` type Declaration

SYNTAX

```
struct  TypeName // does not allocate memory
{
    MemberList
} ;
```

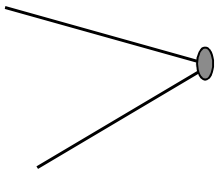
MemberList SYNTAX

```
DataType   MemberName ;
DataType   MemberName ;
```

Sample struct

- Example

```
struct  AnimalType              // declares a  struct data type
{                               //  does not allocate memory
    long          id ;
    string        name ;
    string        species ;
   string         country ;
    int           age ;
    float         weight ;
    } ;
AnimalType    thisAnimal ; // declares a variable of AnimalType
AnimalType    thatAnimal = {123,"Leo","lion","India",10,500.0} //
    initialized values
```



Struct Members

- Cannot have type void
- Cannot nest a structure of it own type
  - It can have members that are structures of other types
- Member names must be unique within a structure
  - They do not have to be unique between structure
- Members can be arrays, pointers, other structures, etc.

```
struct mystruct
{
   char  a;
   int   b;
   float c;
};

struct mystruct myvar;

myvar.b = 99;
```

Accessing struct Members

Dot (period) is the member selection operator

After the struct type declaration, the various members can be used in your program only when they are preceded by a struct variable name and a dot

EXAMPLES

thisAnimal.weight
anotherAnimal.country

Aggregate Operations on structs – 1

- An operation on a data structure as a whole, as opposed to an operation on an individual component of the data structure
- Valid operations:
  - Assignment: thisAnimal=thatAnimal;
  - Pass an argument: (by value or by reference)
  - Return as value of a function
  - Dot operator (thisAnimal.age)

Aggregate Operations on structs

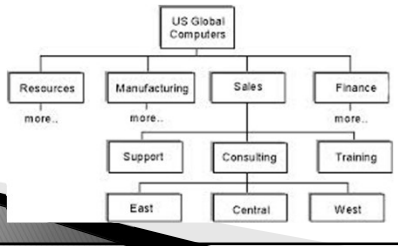
- Invalid operations (must be done one member at a time)
  - I/O
  - Arithmetic (thisAnimal + thatAnimal;)
  - Comparisons of entire struct variables (thisAnimal>thatAnimal;)
- (See example: struct1.cpp)

Array vs. Structures

	Aggregate Operation	Array	Struct
1	Arithmetic	No	No
2	Assignment	No	Yes
3	Input/output	No (except strings)	No
4	Comparison	No	No
5	Parameter passing	By reference only	By value or by reference
6	Function returning a value	No	Yes

Hierarchical Structures

- A member of a struct member can be another struct type
  - This is called nested or hierarchical structures
- Hierarchical structures are very useful when there is much detailed information in each record



Sample Hierarchical structs

- Example:

```
struct time
    {
    int hour;
    int min;
    int sec;
    } myTime;
struct date
    {
    int month;
    int day;
    int year;
    time myTime;
    } myDate;
```

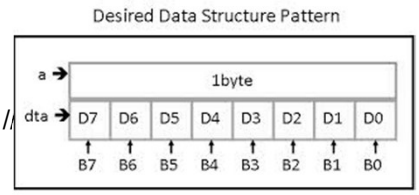
- (See example: struct2.cpp)

Unions in C++

- A union is a struct that holds only one of its members at a time during program execution

EXAMPLE

```
union   WeightType
{
   long    wtInOunces ;
   int     wtInPounds;
   float   wtInTons;
} ;
```



Passing structs

- Can be done three ways:
  - Pass the entire structure (copy)
  - Pass by reference
  - Pass by using a pointer
- (See example: struct3.cpp)

Arrays of Structures

- **Syntax is similar for built in data types (int, floats, etc.)**

```
struct part          // specify a structure
  {
   int modelnumber; // model number of widget
   int partnumber; // part number of widget part
   int quantityPerBox [2]; // quantity in a box (2 types)
   float cost;              // cost of part
  };
  part myPart[10]; // define array of structures
```

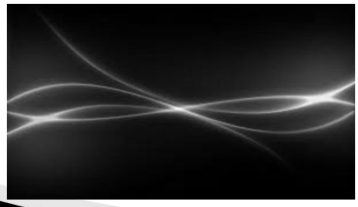
- **(See example: struct4.cpp)**

Functions within Structures

- (See example: struct5.cpp)

Abstraction

- Is the separation of the essential qualities of an object from the details of how it works or is composed
- Focuses on what, not how
- Recommended for managing large, complex software projects
- Abstract Data Type (ADT)
  - A data type that specifies the logical properties without the implementation details



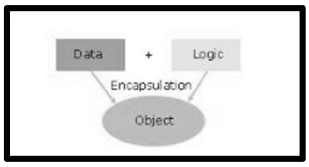
Data Abstraction

- Separates the logical properties of a data type from its implementation

LOGICAL PROPERTIES	IMPLEMENTATION
What are the possible values?	How can this be done in C++?
What operations will be needed?	What data types be used?

C++ Classes – 1

- Classes combine both data and operations into a single cohesive unit (encapsulation)
  - It is similar to a struct with operations that manipulate the data
- Class type variables are called class objects, objects, class instances or instantiations of the class
- The components of a class are called members of the class



C++ Classes – 2

- Classes are abstract data types (ADTs)
  - Data abstraction is the separation of a data type's logical properties from its implementation details
- A client of the class is any software that declares and manipulates class objects
  - Declares class variables
  - Uses public member functions to manipulate/handle class objects

C++ Class Syntax

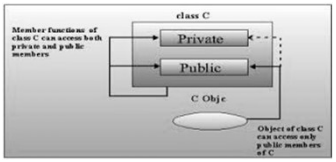
- Syntax:

```
class className
 {
      public:
         // data elements or member functions
      private:
         // data elements or member functions
      protected:
         // data elements or member functions
 };
```

- The class declaration creates a data type and names the members of the class
- It does not allocate memory for any variables of that type
- Client code still needs to declare class variables

Access Specifiers – 1

- Classes are composed of data and functions (methods) members that are public, private, or protected
- public members: public interface
  - Provides the interface between the client code and the class objects
  - They are the only way a client can access/read/modify private data members
  - Function members (methods) are generally declared public



Access Specifiers – 2

- private members: members are inaccessible to clients
  - If a client attempts to access a private member, a compilation error will occur
  - Can be accessed only by the class's member functions and friend functions
  - Default
  - Data members are generally declare private
- protected:
  - Not accessible by client code but are accessible by derived classes
- All members in C++ struct are public

Class member access specifier	Access from own class	Accessible from derived class	Accessible from object
Private member	Yes	No	No
Protected member	Yes	Yes	No
Public member	Yes	Yes	Yes

C++ Member Functions

- Member function syntax:

```
Return-type className::functionName([parameter-list])
  {
  body
  }
```

- If a function only inspects (not modify) data members, the word "const" should be specified in both the function prototype and the heading of the function definition
  - Example: void print ()const;
  - Within the body of a const function, a compilation error occurs if an attempt is made to modify a private data member

Scope Resolution Operator (::)

- C++ programs typically use several class types
- Different classes can have member functions with the same identifier, like Write( )
  - Member selection operator is used to determine the class whose member function Write( ) is invoked

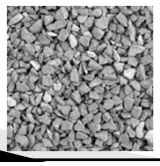
```
currentTime.Write( ) ;
numberZ.Write( ) ;
```

- In the implementation file, the scope resolution operator is used in the heading before the function member's name to specify its class

```
void  TimeType :: Write ( )   const
{          . . .
}
```

Aggregate Operations on Class Objects

- Valid aggregate operations on class objects are the same as the valid operations on structs
- Valid operations:
  - Assignment: object1=object2
  - Pass an argument: (by value or by reference)
  - Return as value of a function
  - Dot operator (object1.print())
- All other operators (+ - * / > < etc.) are invalid unless they are defined by the programmer



Specification and Implementation Files

- An Abstract Data Type (ADT) consists of specification and implementation files
  - Specification file
    - Contains class data members and function prototypes
    - Resides in a header file (.h)
    - There are no implementation details
  - Implementation file
    - Contains the member function definitions
    - Resides in a .cpp file
    - Need to #include the specification file
- Client code should reside in a separate .cpp file
  - Need to #include the specification file
- See Style guide for formats
- (See examples: class1.cpp through class3.cpp)

Functions and Classes

- Class objects can be passed as parameters to functions and returned as function values
- As parameters to functions, classes can be passed either by value or by reference
- If a class object is passed by value, the contents of the data members of the actual parameter are copied into the corresponding data members of the formal parameter
- If a variable is passed by reference, then when the formal parameter changes, the actual parameter also changes

Class Constructors – 1

- An operation that creates a new instance of a class
- A member function whose purpose is to initialize the private data members of a class object
- The name of a constructor is the same as the name of the class
- Return types are invalid for constructors
- Constructors are not invoked with the . operator

Class Constructors – 2

- A class may have several constructors with different parameter lists (signatures)
  - A constructor with no parameters is the default constructor
- A constructor is implicitly invoked when a class object is declared
  - If there are parameters, their values are listed in parentheses in the declaration
- Good idea to have a default constructor

Constructor Syntax

- Syntax
  - className([parameter list]);
- Particular constructor invoked depending on signature
- Sample class specification

```
class  MyTime
{
   public :
   MyTime ( int  initHrs ,  int  initMins ,  int  initSecs ) ;
   MyTime ( ) ;       // default constructor
...
}
```

Invoking the Default Constructor

The syntax to invoke the default constructor is:

className classVariableName;

The statement

clockType yourClock;

declares yourClock to be a variable of the type clockType

The default constructor is executed

Invoking a Constructor with Parameters – 1

- The syntax to invoke a constructor with parameter is:

```
className  classVariableName(argument1,argument2,. . .
   );
```

where argument1, argument2, etc. is either a variable of an expression

Invoking a Constructor with Parameters – 2

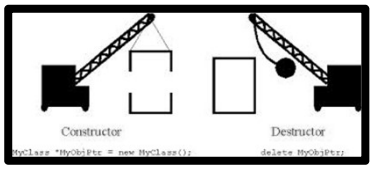
- The number of arguments and their type should match with the formal parameters (in the order given) of one of the constructors
  - If the type of the arguments do not match with the formal parameters of any constructor (in the order given), C++ will use type conversion and look for the best match
    - For example, an integer value might be converted to a floating-point value with zero decimal part
    - An ambiguity will result in a compile time error

Example Invoking a Constructor with Parameters

- For example:
  clockType myClock(5,12,40);
- This statement declares a class variable myClo
- The constructor with parameters of the class clockType will be executed and the three data members of the variable myClock will be set to 5, 12, and 40
- (See examples: class4.cpp and class5.cpp)

Class Destructors

- An operation that destroys an instance of a class
- Whenever a class object goes out of scope (for example: control passes to the end of a block), a class destructor is implicitly invoked
- Destructors are not invoked with the . operator
- A destructor need not always be defined
  - Depends of the type of the class data members
- A class can have only one destructor and it has no parameters



Syntax of Class Destructors

- Syntax of the class destructor
  ~className( );
- Sample destructor

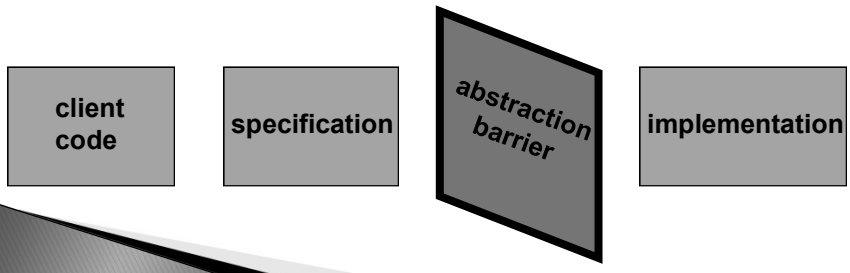
```
class  MyTime
{
    public :
   ~myTime( );
    ...
}
```

Helper Functions

- Helper Functions
  - Public member functions may need to invoke functions
  - Clients do not need to access implementation details of a member functions
  - Private member functions invoked by public member functions

Information Hiding

- Class implementation details are hidden from the client's view
  - This is called information hiding
- Public functions of a class provide the interface between the client code and the class objects



Information Hiding – 1

- Two different type of programmers
  - Author of the abstract data types (ADTs) (C++ classes)
    - Know only what the ADT will do for the client, but nothing about the context in which the ADT will be used by the client

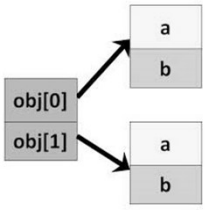
Information Hiding – 2

- Clients of the ADT
  - Know only what the ADT will do, not how the ADT carries out its tasks
  - Implementation details are hidden from the client's view
  - Know how to use the ADT via the application program interface (API)
  - **Number of parameters**
  - **Type of parameters**
  - **If they are input or output**
  - **The purpose of each parameter**
  - **Any restrictions on each parameter**
  - **The type and purpose of the return value**

- Only functions needed by a client should be public
  - Helper functions should be private

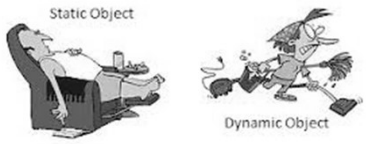
Arrays of Objects

- Syntax is similar for built in data types (int, floats, etc.) and structs
- TimeType startTime[2]; // array of objects
- (See examples: class6.cpp and class7.cpp)



Static Data Members

- A static data member is not duplicated for each object
  - It is shared by all objects of the class
- A static variable could be used to keep track of the number of objects for a class
- See example: class8.cpp



Const Parameter Modifier – 1

- The keyword “const” is a promise to the compiler that you won’t write code that changes something
- Using “const” is a request that the compiler enforce this promise
  - Example: const int myInt = 3; // you won’t change myInt

Const Parameter Modifier – 2

- If one is using call by reference (call by copy never changes parameters) and one does not want a function to change the value of the parameter, one can use the const modifier
  - Automatic error checking done at compilation time
  - Can fool the compiler by indirectly change the variable by using pointers
- The const parameter must appear on the function definition if it appears on the prototype
  - If const is at the end of the prototype and function heading
    - A function cannot change any passed parameters
    - A member function cannot change any of the classes data members
- See example: class9.cpp and class10.cpp