Multimedia Programming 2004

Lecture 6

Erwin M. Bakker
Joachim Rijsdam

C++ Classes

- Abstract Data Types: Struct + their operations
- C++ class is a concept to define data structures and their operations
- The actual implementation of the abstract data type can be hidden to the user!

C++ Classes: List Example

- Abstract Data Type List:
- Data:
  - Name
  - Address
  - Comments
- Basic operations:
  - Create: Create a list
  - Add: Add data to list
  - Delete: Delete data from the list
  - Find: Find data in the list
  - List: Make a print of the contents of the list
- We would like to hide the implementation to the user of our list

C++ Classes: CList

```
class CList                    \\ Declaration of CList
{
public:
   CList();                    \\ Mandatory constructor
   ~CList();                   \\ Mandatory destructor

   void Add(int element);      \\ The public operations
   void Delete(int element);   \\ Public member functions
   bool Find(int element);
   bool IsEmpty();
   void List()

private:                       \\ Private declarations
   int element_list[MAX_ELEMENTS]; \\ Data
   int number_of_elements;

   int FindPlace(int element); \\ Private member function
};
```

C++ Classes: CList

```
class CList                          \\ Declaration of CList
void Add(int element);               \\ Public member functions
void Delete(int element);
bool Find(int element);
bool IsEmpty();
void List();
```

Is everything the user needs to know to use the class CList

```
int main( void )
{
    CList list;

    list.Add(2);
    list.Add(5);
    list.Delete(2);
    list.List();
    return 0;
}
```

C++ Classes Declaration

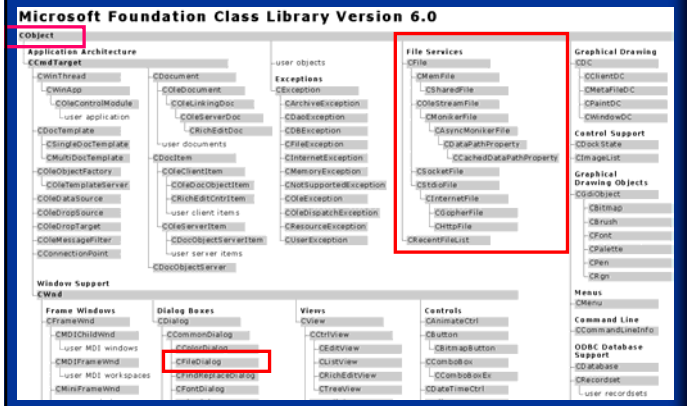
```
Class CMyClass
{
public:
      CMyClass();
      ~CMyClass();

      void memberfunction_this(int element);
      void memberfunction_that(int element);
      …
      int my_data;
      int my_array[10];
      int *my_pointers;
      …

private:
      int my_private_memberfunctions();
      …
      int my_private_data;
      …

Public:
      … etc.
};
```

```
CMyClass Examp;
Examp.my_data = 1;
Examp.memberfunction_this(3);
```

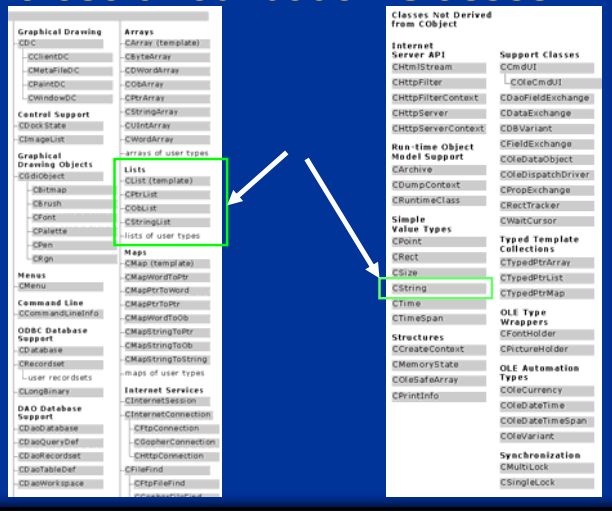
Using Existing C++ Classes

- CString
- CFile
- CList
- Microsoft Foundation Class Library (MFC)
- Etc.

Microsoft Foundation Classes



Microsoft Foundation Classes



CString

Construction
- CString Constructs **CString** objects in various ways.

The String as an Array
- GetLength Returns the number of characters in a **CString** object. For multibyte characters, counts each 8-bit character; that is, a lead and trail byte in one multibyte character are counted as two characters.
- IsEmpty Tests whether a **CString** object contains no characters.
- Empty Forces a string to have 0 length.
- GetAt Returns the character at a given position.
- operator [] Returns the character at a given position — operator substitution for **GetAt**.
- SetAt Sets a character at a given position.
- operator LPCTSTR Directly accesses characters stored in a **CString** object as a C-style string.

Assignment/Concatenation
- operator = Assigns a new value to a **CString** object.
- operator + Concatenates two strings and returns a new string.
- operator += Concatenates a new string to the end of an existing string.

CList: member functions

Construction
- **CList** Constructs a **CList** object.

Head/Tail Access
- **GetHead** Returns the head element of the list (cannot be empty).
- **GetTail** Returns the tail element of the list (cannot be empty).

Operations
- **RemoveHead** Removes the element from the head of the list.
- **RemoveTail** Removes the element from the tail of the list.
- **AddHead** Adds an element (or all the elements in another list) to the head of the list (makes a new head).
- **AddTail** Adds an element (or all the elements in another list) to the tail of the list (makes a new tail).
- **RemoveAll** Removes all the elements from this list.

Status
- **GetCount** Returns the number of elements in this list.
- **IsEmpty** Tests for the empty list condition (no elements).

CList

- The **CList** class supports ordered lists of non-unique objects accessible sequentially or by value.
- **CList** lists behave like doubly-linked lists.
- **template< class** *TYPE*, **class** *ARG_TYPE* **>class CList : public CObject**

CList: member functions

Iteration
- **GetHeadPosition** Returns the position of the head element of the list.
- **GetTailPosition** Returns the position of the tail element of the list.
- **GetNext**Gets the next element for iterating.
- **GetPrev** Gets the previous element for iterating.

Retrieval/Modification
- **GetAt** Gets the element at a given position.
- **SetAt** Sets the element at a given position.
- **RemoveAt** Removes an element from this list, specified by position.

Insertion
- **InsertBefore** Inserts a new element before a given position.
- **InsertAfter** Inserts a new element after a given position.

Searching
- **Find** Gets the position of an element specified by pointer value.
- **FindIndex** Gets the position of an element specified by a zero-based index.

CFile

- **CFile** is the base class for Microsoft Foundation file classes. It directly provides unbuffered, binary disk input/output services, and it indirectly supports text files and memory files through its derived classes. **CFile** works in conjunction with the **CArchive** class to support serialization of Microsoft Foundation Class objects.
- The hierarchical relationship between this class and its derived classes allows your program to operate on all file objects through the polymorphic **CFile** interface. A memory file, for example, behaves like a disk file.
- Use **CFile** and its derived classes for general-purpose disk I/O. Use **ofstream** or other Microsoft iostream classes for formatted text sent to a disk file.
- Normally, a disk file is opened automatically on **CFile** construction and closed on destruction. Static member functions permit you to interrogate a file's status without opening the file.
- For more information on using **CFile**, see the article Files in MFC in *Visual C++ Programmer's Guide* and File Handling in the *Run-Time Library Reference*.

CFile Class Members

Data Members
- m_hFile Usually contains the operating-system file handle.

Construction
- CFile Constructs a **CFile** object from a path or file handle.
- Abort Closes a file ignoring all warnings and errors.
- Duplicate Constructs a duplicate object based on this file.
- Open Safely opens a file with an error-testing option.
- Close Closes a file and deletes the object.

CFile Input/Output

- Read Reads (unbuffered) data from a file at the current file position.
- ReadHuge Can read more than 64K of (unbuffered) data from a file at the current file position. Obsolete in 32-bit programming. See Read.
- Write Writes (unbuffered) data in a file to the current file position.
- WriteHuge Can write more than 64K of (unbuffered) data in a file to the current file position. Obsolete in 32-bit programming. See Write.
- Flush Flushes any data yet to be written.

CFile Position

- Seek Positions the current file pointer.
- SeekToBegin Positions the current file pointer at the beginning of the file.
- SeekToEnd Positions the current file pointer at the end of the file.
- GetLength Retrieves the length of the file.
- SetLength Changes the length of the file.

CFile Locking and Status

Locking

- LockRange Locks a range of bytes in a file.
- UnlockRangeUnlocks a range of bytes in a file.

Status

- GetPosition Retrieves the current file pointer.
- GetStatus Retrieves the status of this open file.
- GetFileName Retrieves the filename of the selected file.
- GetFileTitle Retrieves the title of the selected file.
- GetFilePath Retrieves the full file path of the selected file.
- SetFilePath Sets the full file path of the selected file.

CFile Static

- Rename Renames the specified file (static function).
- Remove Deletes the specified file (static function).
- GetStatus Retrieves the status of the specified file (static, virtual function).
- SetStatus Sets the status of the specified file (static, virtual function).

Structured Programming

- Layout (your own style but consequent)
- Comment to clarify your code
- Naming: variables, functions, classes, etc. should be meaningful
- Appropriate control structures (if then, while, for) should be used
- Functions, and member functions should make the right division of tasks (careful top-down design)
- Data abstraction

Debugging

- A two-step process:

1. correct compile-time errors that prevent you from building your project, such as incorrect syntax, misspelled keywords, or type mismatches.
2. use the debugger to detect and correct logic errors and errors in sequencing, branching, and interaction among program components.

Debug vs Release

- Win32 Debug: Full symbolic debugging information in Microsoft format No optimization (optimization generally makes debugging more difficult)
- Win32 Release: No symbolic debugging information Optimized for maximum speed

Change debug options:

- to output line numbers only,
- to generate a mapfile
- to redirect output.
- Etc.

Debugging Tools

- Visual C++ provides a variety of tools to help with the varied tasks of tracking down errors in the code and program components. The debugger interface provides
  - special menus,
  - windows,
  - dialog boxes, and
  - spreadsheet fields.
  - Drag-and-drop: moving debug information between components.
  - Interactive: the debugger is paused in break mode, meaning the debugger is waiting for user input after completing a debugging command (like break at breakpoint, step into/over/out/to cursor, break at exception, break after Break command or Restart).

Debugger Menu's

- **<Build><Start Debug>** contains a subset of the commands on the full **Debug** menu.
- These commands (**Go**, **Step Into**, **Run To Cursor** and **Attach to Process**) start the debugging process
- The **Debug** menu appears in the menu bar while the debugger is running (even if it is stopped at a breakpoint).
- From the **Debug** menu, you can control program execution and access the QuickWatch window. When the debugger is not running, the **Debug** menu is replaced by the **Build** menu.
- The **View** menu contains commands that display the various debugger windows, such as the Variables window and the Call Stack window.
- From the **Edit** menu, you can access the **Breakpoints** dialog box, from which you can insert, remove, enable, or disable breakpoints.

Debug Commands

- **Go** Executes code from the current statement until a breakpoint or the end of the program is reached, or until the application pauses for user input. (Equivalent to the Go button on the toolbar.)
- **Step Into** Single-steps through instructions in the program, and enters each function call that is encountered.
- **Run to Cursor** Executes the program as far as the line that contains the insertion point. This is equivalent to setting a temporary breakpoint at the insertion point location.
- **Attach to Process** Attaches the debugger to a process that is running. Then you can break into the process and perform debugging operations like normal.

Debug Commands Control Program Execution

- **Go** Executes code from the current statement until a breakpoint or the end of the program is reached, or until the application pauses for user input. (Equivalent to the Go button on the Standard toolbar.) When the Debug menu is not available, you can choose Go from the Start Debug submenu of the Build menu.
- **Restart** Resets execution to the first line of the program. This command reloads the program into memory, and discards the current values of all variables (breakpoints and watch expressions still apply). It automatically halts at the main( ) or WinMain( ) function.
- **Stop Debugging** Terminates the debugging session, and returns to a normal editing session.
- **Break** Halts the program at its current location.

Debug Commands Control Program Execution

- **Step Into** Single-steps through instructions in the program, and enters each function call that is encountered. When the Debug menu is not available, you can choose Step Into from the Start Debug submenu of the Build menu.
- **Step Over** Single-steps through instructions in the program. If this command is used when you reach a function call, the function is executed without stepping through the function instructions.
- Step Out Executes the program out of a function call, and stops on the instruction immediately following the call to the function. Using this command, you can quickly finish executing the current function after determining that a bug is not present in the function.
- **Run to Cursor** Executes the program as far as the line that contains the insertion point. This command is equivalent to setting a temporary breakpoint at the insertion point location. When the Debug menu is not available, you can choose Run To Cursor from the Start Debug submenu of the Build menu.
- **Step Into Specific Function** Single steps through instructions in the program, and enters the specified function call. This works for any number of nesting levels of functions.

Debug Menu Commands

- **Exceptions** Displays the Exceptions dialog, which you can use to specify how you want the debugger to handle your program exceptions.
- **Threads** Displays the Threads dialog, which you can use to suspend, resume, or set focus to progam threads.
- **Show Next Statement** Shows the next statement in your program code. If source code is not available, displays the statement within the Disassembly window.
- **QuickWatch** Displays the Quick Watch window, where you can work with expressions.

Debugger Windows

- Output Information about the build process, including any compiler, linker, or build-tool errors, as well as output from the **OutputDebugString** function or the **afxDump** class library, thread termination codes, loading symbols notification and first-chance exception notifications.
- Watch Names and values of variables and expressions.
- Variables Information about variables used in the current and previous statements and function return values (in the **Auto** tab), variables local to the current function (in the **Locals** tab), and the object pointed to by **this** (in the **This** tab).
- Registers Contents of the general purpose and CPU status registers.
- Memory Current memory contents.
- Call Stack Stack of all function calls that have not returned.
- Disassembly Assembly-language code derived from disassembly of the compiled program.

Set Break Points

- at a source-code line
- at the beginning of a function
- at the return point of a function
- at a label
- Viewing, disabling, and removing breakpoints

Debugger Dialog Boxes

- **Breakpoints** A list of all breakpoints assigned to your project. Use the tabs in the Breakpoints dialog box to create new breakpoints of various types.
- **Exceptions** System and user-defined exceptions for your project. Use the **Exceptions** dialog box to control how the debugger handles exceptions.
- **QuickWatch** A variable or expression. Use QuickWatch to quickly view or modify a variable or expression or to add it to the Watch window.
- **Threads** Application threads available for debugging. Use the Threads dialog box to suspend and resume threads and to set focus.
- Drag and drop: If you expand an object (Obj, for example) in the Variables window, you can drag a member of that object (such as Obj.child) to the Watch window.

Viewing Values

- of a variable or expression or the contents of a register
- of a variable using QuickWatch
- of a variable or expression or the contents of a register in the Watch window
- View type information for a variable in the Watch window
- View a variable in the Variables window
- View type information for a variable in the Variables window
- Display meaningful values for your custom data type

Changing Values

When the program is paused at a breakpoint or between steps, the value of any non-**const** variable or contents of any register can be changed.

- Modify the value of a variable or contents of a register using QuickWatch
- Modify the value of a variable or contents of a register using the Watch window
- Modify the value of a variable in the Variables window

Edit and Continue

- With Edit and Continue you can make changes to your source code while the program is being debugged (with some limitations).
- You can apply code changes while the program is running or halted under the debugger.

Running to a Location

- until a breakpoint is reached
- to the cursor or cursor location in object code or in the call stack
- to a specified function
- Set the next statement to execute
- Set the next disassembled instruction to execute

Stepping into Functions

- Run the program and execute the next statement
- Step into a specific function

ASSERT

ASSERT(*booleanExpression* **)**

- *booleanExpression:* Specifies an expression (including pointer values) that evaluates to nonzero or 0.
- ASSERT evaluates its argument, if the result is 0, the macro prints a diagnostic message and aborts the program. If the condition is nonzero, it does nothing.
- The diagnostic message has the form: assertion failed in file <name> in line <num> where *name* is the name of the source file, and *num* is the line number of the assertion that failed in the source file.
- This function is available only in the Debug version of MFC. In the Release version of MFC, **ASSERT** does not evaluate the expression and thus will not interrupt the program.
- If the expression must be evaluated regardless of environment, use the **VERIFY** macro in place of **ASSERT**.

Example

```
// example for ASSERT
CPerson* pperson = new CPerson("James", 21 );  //CPerson derived from CObject

ASSERT( pperson != NULL )                    // Terminates if pperson equals NULL
ASSERT( pperson->IsKindOf( RUNTIME_CLASS( CPerson) ) )
// Terminates program only if pperson is not a CPerson*.
```

VERIFY (interrupts in both Release and Debug programs)

VERIFY(*booleanExpression* **)**

- *booleanExpression:* Specifies an expression (including pointer values) that evaluates to nonzero or 0.
- In the debug version of MFC, the **VERIFY** macro evaluates its argument. If the result is 0, the macro prints a diagnostic message and halts the program. If the condition is nonzero, it does nothing.
- The diagnostic message has the form: assertion failed in file <name> in line <num> where *name* is the name of the source file and *num* is the line number of the assertion that failed in the source file.
- In the release version of MFC, **VERIFY** evaluates the expression but does not print or interrupt the program. (For example the function call will be made (which is not the case in the ASSERT case))