Module 2

Object-Oriented Programming with Visual C++

1 Classes & Objects

1.1 Class

A class is the blueprint from which objects are instantiated. Every object instantiated from a class will have the same set of attributes and operations. You can define and use a complete class within a single C++ source file. To share classes across multiple source files, you need to separate a class definition into two sections; declaration and implementation.

Project Information

```
Project Name: Class1
Project Type: Win32 Console Application | Empty Project
Project     : C:\CPPDEV\SRC
Project     : Module2
```

Application main function: Class1\main.cpp

```
#include <stdio.h>

void main() {
}
```

For this example, we will be defining a class named **Time** to store and manage time information. To share this class between multiple source files, you can add a **Time.h** header for the class declaration and separate **Time.cpp** for the class implementation. To define a new class, use the **class** keyword followed by the class name in the same manner as that for a **struct**. Note that we named the following class as **CTime** rather than just **Time**. This is just a common naming convention to easily identity a symbol representing a class.

Declaring a class: Time.cpp

```
//  Time.h - class declaration

class CTime {
};
```

In order to use a class, all you need is the class declaration. Include the declaration to instantiate one or more objects from the class. A class cannot be declared more than once. A compilation error will occur if you include the declaration directly or indirectly more than once from the same source file.

Including the class declaration: main.cpp

```
#include <stdio.h>
#include "Time.h"
#include "Time.h"    // class redefinition error
```

It will be difficult to ensure that the declaration will be included only once as you may use multiple header files that will include the same declarations. Use **#pragma once** directive to ensure that the header file is processed once no matter how many times it has been included.

Ensuring header is processed only once: Time.h

```
//  Time.h - class declaration

#pragma once

class CTime {

};
```

1.2 Object

An object is an instance of a class. Objects instantiated from one class will have the same set of attributes but may contain different values since each object is allocated its own memory. In fact a C++ object is nothing more than just a block of memory that has been associated with a class so that the compiler knows how much memory to allocate and how that memory can be accessed. Variables have always been used to allocate memory since even the C programming language. Instancing an object in C++ is really just the same as declaring a basic variable. However, the context of the declaration will determine what area of memory will be used for the object. Objects can be instantiated in the static memory, call stack memory or dynamic memory heap. The amount of memory allocated is determined by the number and the type of attributes declared in the class. Since there is no such thing as a zero memory allocation, even if a class does not have any attributes, the minimum size of an object is 1 byte. This can be proven by using a **sizeof** operator with the class or an instance of that class.

Instancing and checking size of object: main.cpp

```
CTime time1;

void main() {
	size_t size1 = sizeof(CTime);
	size_t size2 = sizeof(time1);
	printf("size1=%u\nsize2=%u\n", size1, size2);
}
```

1.3 Class Members

There are no benefit to instance objects that have no attributes since the main purpose of doing so is to allocate memory for those attributes. For our class, we will now add three fields to represent a time; **hour**, **minute** and **second**. Time cannot be negative and we need minimum one byte to hold the value for each attribute. Since there is no byte data type in C++, we will define our own type based on the **unsigned short** type. Use member variables to allocate memory for each attribute. You will need to add a **public** access modifier to the member variables to allow them to be accessible to functions outside of the object since the default access is **private** where only functions within that class have access to those members. Build and run the same program now and you can see that the size of a **CTime** object has changed to 6 bytes.

Declaring member variables: Time.h

```
//  Time.h - class declaration

#pragma once

typedef unsigned short ushort;

class CTime {
public:
    ushort  hour;
    ushort  minute;
    ushort  second;
};
```

You can now store and retrieve values from the attributes of a **CTime** object by using the member variables. Since different object has their own memory, each object can store their own values without affecting each other.

Accessing object attributes: main.cpp

```
CTime time1;
CTime time2;

void main() {
    time1.hour  = 10;
    time1.minute= 20;
    time1.second= 30;
    time2.hour  = 11;
    time2.minute= 22;
    time2.second= 33;
    char * format = "%02u:%02u:%02u\n";
    printf(format, time1.hour, time1.minute, time1.second);
    printf(format, time2.hour, time2.minute, time2.second);
}
```

Allowing external functions to have direct access to the member variables will be dangerous since they can store any values they wish regardless of whether the value is valid or not. For example, following assignments even though are considered as invalid will not be detected during the assignment but may cause issues later during runtime and the developer will have trouble finding out the original cause of the issue. This is why the default access is **private** rather than **public**. Instead of removing **public** keyword, you should replace it with **private**, even though is the default, to avoid accidentally changing members to public by accidentally declaring a public member before them.

Attributes should be private: Time.h

```
class CTime {
private:
    ushort  hour;
    ushort  minute;
    ushort  second;
};
```

You will get compilation errors if you try to build the application now since our code is still attempting to access the attributes which are now private. Instead of allowing external functions to access attributes directly, they will be forced to use operations provided by the object to store and access the attributes. In this way, we have control over what can be stored in those attributes by validating the input values before storing them. You can implement operations in a class using member functions. Member functions must of course be marked **public** in order for them to be callable by external functions. You will need two functions per attribute; one to return the value of the attribute and one to store the value into the attribute. Following are declarations of functions that we will complete in the class implementation file.

Declaring operations: Time.h

```
class CTime {
private:
    ushort  hour;
    ushort  minute;
    ushort  second;
public:
    ushort GetHour();
    ushort GetMinute();
    ushort GetSecond();
    void PutHour(ushort hour);
    void PutMinute(ushort minute);
    void PutSecond(ushort second);
};
```

Functions that are used to access data from an object are commonly called as accessor functions. The following are the implementations of the accessors for **CTime** class. Include the declaration to be matched up against the implementation. The scope resolution (**::**) operator will be required to associate each member implemented to the correct class.

Implementing member functions: Time.cpp

```
//  Time.cpp - class implementation

#include "Time.h"

ushort CTime::GetHour() { return hour; }
ushort CTime::GetMinute() { return minute; }
ushort CTime::GetSecond() { return second; }
```

Input values should be validated. If it is invalid, you should generate an exception to indicate failure by using a **throw** keyword. An exception is a signal of an error that can be detected and handled by the caller. It is up to you to decide what exactly to throw as an exception. It is common to throw an error code or message. For our example, we will throw error messages.

Validation and generating exceptions

```
void CTime::PutHour(ushort hour) {
    if (hour > 23) throw "Invalid hour value.";
    this->hour = hour;
}

void CTime::PutMinute(ushort minute) {
    if (minute > 59) throw "Invalid minute value.";
    this->minute = minute;
}

void CTime::PutSecond(ushort second) {
    if (second > 59) throw "Invalid second value.";
    this->second = second;
}
```

Note we also had to use **this** keyword to differentiate between a member variable and an input argument since they have the same name. The keyword explicitly returns a pointer to the current object so you use the pointer access (**->**) operator rather than the member (**.**) access operator to reference the members. This is implicitly done if there is no conflict between naming of the member variables with input arguments or local variables. This can also be resolved by using a different name for arguments and local variables but an easier way is to have a special naming convention for member variables. It is quite common for programmers to prefix member variables with **m_** to easily differentiate them to avoid naming conflicts later on.

Naming convention for member variables: Time.h

```
class CTime {
private:
    ushort  m_hour;
    ushort  m_minute;
    ushort  m_second;
          :
};
```

Code placed into implementation file is pre-compiled and a function call is required to execute the code. If the code is simple like the get functions you implemented for the above class, you can inline them in the header file. What this means is that all calls to the function is replaced by compiled code instead. This also means that anyone can change the code since header files are distributed in source. Performance is improved because the overhead of a function call has been removed but the performance gain will only be noticeable when a function is required to be called thousands to millions of times within a second. This kind of fine optimizations is usually only required when developing computer games.

Inline functions

```
class CTime {
             :
      ushort GetHour() { return m_hour; }
      ushort GetMinute() { return m_minute; }
      ushort GetSecond() { return m_second; }
             :
};
```

Using accessor functions: main.cpp

```
time1.PutHour(10);
time1.PutMinute(20);
time1.PutSecond(30);
time2.PutHour(11);
time2.PutMinute(22);
time2.PutSecond(33);
char * format = "%02u:%02u:%02u\n";
printf(format, time1.GetHour(), time1.GetMinute(), time1.GetSecond());
printf(format, time2.GetHour(), time2.GetMinute(), time2.GetSecond());
```

Program can now build successfully and passing invalid values would throw exceptions immediately which can easily be debug and solved if the value is provided by the programmer. If the value is provided at runtime by the user, you will have to trap and handle the error as shown below by using a **try catch** block. You must know exactly what is thrown in order to be able to catch and process the error.

Structured exception handling: main.cpp

```
try {
    time1.PutHour(22);  // Success. Continue with next statement
    time1.PutMinute(66);// Failure. Jump to catch section
    time1.PutSecond(44);// Will not be executed
}
catch (char *e) {
    printf("Error : %s\n", e);
}
```

If you use Visual C++, you can make accessor functions easier to call by making them behave like attributes by using property declarations. A property is a virtual attribute then when accessed is translated to accessor function calls when compiled. Use **__declspec** keyword to declare a **property** and assign the functions for the **get** and **put** operations. You can change the code to use the properties instead of calling the accessor functions directly. Remember this is not a standard C++ feature and can only be applied to Visual C++.

Declaring properties: Time.h

```
class CTime {
        :
public:
    __declspec(property(get=GetHour,put=PutHour)) ushort Hour;
    __declspec(property(get=GetMinute,put=PutMinute)) ushort Minute;
    __declspec(property(get=GetSecond,put=PutSecond)) ushort Second;
};
```

Using properties: main.cpp

```
time1.Hour = 10;
time1.Minute = 20;
time1.Second = 30;
char * format = "%02u:%02u:%02u\n";
printf(format, time1.Hour, time1.Minute, time1.Second);
```

1.4 Passing Objects

You can pass objects as arguments by value, pointer or reference. If you pass by value, a copy of the object will be made. If you pass by pointer or pass by reference then no copy will be made. This ensures the function is accessing the original object. You do not have to worry about the copy, as the compiler will automatically generate code to destroy the copy at the end of the function.

Passing objects to functions: Class1\main.cpp

```
void Display(CTime obj) { // copy of original object
    char *format = "%2u:%02u:%02u\n";
     printf(format, obj.Hour, obj.Minute, obj.Second);
}

void SetToMidnight(CTime *pObj) { // address of original object
    pObj->Hour = 0; pObj->Minute = 0; pObj->Second = 0;
}

void SetToAfternoon(CTime &obj) { // reference to original object
    obj.Hour = 12; obj.Minute = 0; obj.Second = 0;
}

void main() {
    SetToMidnight(&time1);
    SetToAfternoon(time2);
    Display(time1);
    Display(time2);
}
```

Sometimes we still want to pass an object by pointer or by reference to a function that does not change the object. This is because we do not want a copy of the object to be made. This will consume additional memory and CPU time to make and store the copy. However if we do not pass by value, there might be a chance that the object might be changed inside the function which may not be what we intended. To solve this problem, we can pass pointers and references as constants using the **const** keyword. You can only use a constant pointer or a constant reference to call constant functions. Constant functions are by default not allowed to change the content of the object.

Passing a constant reference

```
void Display(const CTime& obj) { // pass as constant reference
    char *format = "%2u:%02u:%02u\n";
     printf(format, obj.Hour, obj.Minute, obj.Second);
}
```

The above code cannot be compiled yet since the **GetHour**, **GetMinute** and the **GetSecond** functions have not been marked as **const**. These functions have no problems being marked as constants since they do not change the attributes of the object. There is still a loophole though; any attributes marked as **mutable** can be changed in constant functions. Once you have updated the class as shown below, you can now build the program successfully.

Declaring constant functions: Class1\Time.h

```
ushort GetHour() const { return m_hour; }
ushort GetMinute() const { return m_minute; }
ushort GetSecond() const { return m_second; }
```

While there are no issues passing an object by pointer or reference, there may be problems when you do not return an object by value. When you return an object by value, it is guaranteed that a copy of the object is returned instead of the original. Thus it does not matter what happens to the original. Following is a function that returns a local object by reference. The problem is that the local variables and objects are created on call stack memory. Stack memory will be recycled at the end of the function so all memory and objects will be destroyed automatically when the function returns. Thus you are returning a reference to something that no longer exists.

Returning a local object by reference

```
CTime& GetObject1() {
    CTime time;     // local object
    time.Hour = 10;
    time.Minute = 20;
    time.Second = 30;
    return time;    // reference to destroyed object
}
```

You can decide to return a dynamic object instead. Dynamic objects are created on the dynamic heap by using the **new** keyword and will not be destroyed until **delete** keyword is used. You have to use a pointer to contain the address of the dynamic object is order to access the object or destroy it later.

Returning a dynamic object

```
CTime *GetObject2() {
    CTime *pTime;       // local pointer
    pTime = new CTime;  // stores address of dynamic object
    pTime->Hour = 10; pTime->Minute = 20; pTime->Second = 30;
    return pTime;       // return address of dynamic object
}
```

The problem is the caller must remember to destroy the object when no longer in use or otherwise causing a memory leak. The address to the dynamic object must be kept in order to destroy the object with **delete** keyword. It will also be difficult to know whether a dynamic object has been destroyed or not since the address is still in the pointer variable. Thus it is a best practice then once it has been destroyed the pointer has to be invalidated by initializing it to address 0, commonly called as a null pointer. When you pass pointers around, functions can check to make sure the pointer is not null before using it.

Using dynamic objects

```
CTime* p1 = GetObject2();    // store address to dynamic object
Display(*p1);                // convert pointer to reference
delete p1;                   // destroy dynamic object
p1 = 0;                      // invalidate the pointer
p1 = NULL;                   // using the NULL macro (use nullptr in C++/11)
```

Header files for Standard C Library have already defined the **NULL** macro. You can use this macro instead to describe clearly to the programmer looking at the code that you are invalidating a pointer. If you need to use dynamic objects very often, you should also define a **SAFE_DELETE** macro to help destroy the object and invalidate the pointer for you. In C++/11 you can now use the **nullptr** keyword to replace NULL macro.

Defining your own macros: Time.h

```
#ifndef NULL
#define NULL 0
#endif

#ifndef SAFE_DELETE
#define SAFE_DELETE(ptr)\
      delete ptr; \
      ptr = NULL;
#endif
```

The important thing to remember here is to never return pointers or references to objects created in call stack memory. No problems for those objects created in static or dynamic memory. Static memory is safest since the memory will be released only at the end of the program. Do not assume that all local variables or objects will only use stack memory. You can use the **static** keyword for local variables to use static memory instead. Then there will not be an issue to return a pointer or reference to this type of object.

Returning a static object by reference

```
CTime& GetObject1() {
    static CTime time;  // static object
    time.Hour = 10;
    time.Minute = 20;
    time.Second = 30;
    return time;    // reference to static memory is ok
}
```

For our example, we wish to allow outside operations to use one single function call to set the entire time. So we will add a **PutTime** function and a **GetTime** function. We can pass arguments to first function by value since it does not change the arguments but the second function needs to return time information within the arguments so for this function we will pass by reference.

Declaring new class functions: Time.h

```
class CTime {
        :
    void PutTime(ushort hour, ushort minute, ushort second);
    void GetTime(ushort& hour, ushort& minute, ushort& second);
};
```

Implementing the functions: Time.cpp

```
void CTime::PutTime(ushort hour, ushort minute, ushort second) {
    Hour = hour; Minute = minute; Second = second; }

void CTime::GetTime(ushort& hour, ushort& minute, ushort& second) {
    hour = Hour; minute = Minute; second = Second; }
```

Using simpler method to set time: main.cpp

```
time1.PutTime(10, 20, 30); Display(time1);
time2.PutTime(11, 22, 33); Display(time2);
```

1.5 Overloading Functions

In C++, you can have multiple functions with the same function name as long as they do not have the same number of arguments or the arguments are of a different data type. The following example shows overloading of **PutTime** and **GetTime** functions based on different number of arguments and their data types. The compiler can then distinguish which function you wish to call based on the arguments that you passed in.

Overloading the PutTime and GetTime function: Time.h

```
class CTime {
        :
    void PutTime(ushort hour, ushort minute, ushort second);
    void PutTime(ushort hour, ushort minute);
    void PutTime(ushort hour);
    void GetTime(ushort& hour, ushort& minute, ushort& second);
    void GetTime(ushort *pHour, ushort *pMinute, ushort *pSecond);
};
```

Implementing overloaded functions: Time.cpp

```
void CTime::PutTime(ushort hour, ushort minute) { PutTime(hour, minute, 0); }
void CTime::PutTime(ushort hour) { PutTime(hour, 0); }
void GetTime(ushort *pHour, ushort *pMinute, ushort *pSecond) {
    *pHour = Hour; *pMinute = Minute; *pSecond = Second;
}
```

Calling overloaded functions: main.cpp

```
time1.PutTime(10, 20, 30);   Display(time1);
time1.PutTime(10, 20);       Display(time1);
time1.PutTime(10);           Display(time1);

ushort h, m, s;
time1.GetTime(h, m, s);          // GetTime(ushort&,ushort&,ushort&);
time1.GetTime(&h, &m, &s);       // GetTime(ushort*,ushort*,ushort*);
```

Sometimes overloading can get abused. Following example overloads **PutTime** and **GetTime** further to support setting and retrieving time based on the total number of seconds. Even though the data type is different, sometimes it would be impossible for the compiler to know which function to call.

Additional overloaded functions: Time.h

```
typedef unsigned long ulong;

class CTime {
        :
    ulong GetTime();
    void PutTime(ulong value);
        :
```

Implementing overloaded functions: Time.cpp

```
ulong CTime::GetTime() {
    return (ulong)(m_hour * 3600 + m_minute * 60 + m_second); }

void CTime::PutTime(ulong value) {
    m_hour = (ushort)((value / 3600) % 24);
    m_minute = (ushort)((value % 3600) / 60);
    m_second = (ushort)(value % 60); }
```

Ambiguous function calls: main.cpp

```
ushort v1 = 10;
ulong v2 = 32768;
time1.PutTime(v1);               // PutTime(ushort); no ambigiuty
time1.PutTime(v2);               // PutTime(ulong); no ambigiuty
time2.PutTime((ushort)10);       // ambigious, typecasting required
time2.SetTime((ulong)20);        // ambigious, typecasting required
```

Ambiguity can be resolved using typecasting. However the basic principles of object-oriented programming to reduce complexity not increase it. Forcing the programmer to typecast in order to call the right function is not feasable. In will be better to forgo overloading and create new functions. Rename the functions we added to **GetValue** and **PutValue** instead. Since it accepts one value, you can also declare a property called **Value** for it.

1.6 Default Arguments

You can reduce overloading for functions that have a different number of arguments by providing default values for arguments. However, since the default arguments are set during declaration, it is possible they can be changed by other programmers. This may cause inconsistency when a programmer uses different header files to compile their projects and the default arguments in the header files are not the same.

Using default arguments: Time.h

```
class CTime {
        :
      void PutTime(ushort hour = 0, ushort minute = 0, ushort second = 0);
        :
};
```

Calling function with default arguments: main.cpp

```
time1.PutTime(10,20,30);// PutTime(10,20,30)
time1.PutTime(10,20);   // PutTime(10,20, 0)
time1.PutTime(10);      // PutTime(10, 0, 0)
time1.PutTime();        // PutTime( 0, 0, 0)
```

1.7 Constructors

If you do not have a constructor or a destructor the compiler will generate them if the class is not the root class. A compiler will not generate any constructor or destructor if you already define your own. The purpose of a constructor is to initialize or prepare an object to be used while the constructor is used to finalize or clean up the object when it is no longer in use. A destructor may not always be required but at least one constructor is required if you have attributes. While static memory is always initialized, call stack and dynamic memory are not auto initialized. This can be proven by the following code. The constructor will always be called when an object is created and the destructor will always be called just before the object is destroyed.

Only static memory is auto-initialized: main.cpp

```
CTime time3;    //  local object
Display(time1); //  00:00:00 guaranteed
Display(time3); //  ??:??:??
```

To ensure consistency, we must initialize the attributes using a constructor and not depend on where an object is created. The constructor is a function that has the same name as the class name. The following shows the declaration and implementation of our default constructor to initialize the attributes.

Constructor declaration: Time.h

```
class CTime() {
public:
    CTime();// constructor function
            :
};
```

Constructor implementation: Time.cpp

```
CTime::CTime() {
    m_hour = m_minute = m_second = 0;
}
```

You can overload constructors. Since we have to explicitly call constructors, we can choose exactly which constructor to call. A destructor is automatically called so you can only have one destructor per class. In the following, we declare two additional custom constructors to provide different ways to initialize the objects from the class.

Overloading constructors: Time.h

```
class CTime() {
public:
    CTime();
    CTime(ushort hour, ushort minute, ushort second);
    CTime(ulong value);
            :
};
```

Implementing custom constructors

```
CTime::CTime(ushort hour, ushort minute, ushort second) {
    SetTime(hour, minute, second);  // may throw exception
}

CTime::CTime(ulong value) {
    PutValue(value);    // does not throw exception
}
```

Be careful with constructors that may generate an exception. They should never be called through dynamic objects. If an exception occurs, the address for that object will not be returned and you will not be able to destroy it thus causing a memory leak. It would be better not to have constructors that throw exceptions if possible. Even though our attributes are simple values and not objects, you can initialize them using same syntax as aggregated objects. The constructor of an outer object can call constructors of inner objects using extend (**:**) operator as shown below. This proves that constructors can be extended. Constructors for simple values accept a single argument representing the value.

Calling constructors for aggregated objects

```
CTime::CTime() : m_hour(0), m_minute(0), m_second(0) { }
```

The following demonstrate the different syntax for calling all constructors. Local and global objects use the same syntax while dynamic objects will use a different syntax. Inline objects uses same syntax as dynamic objects except without the **new** keyword. Inline objects are used once in a statement and automatically destroyed at the end of the statement. The following shows the different syntax.

Calling overloaded constructors: main.cpp

```
CTime t1;                // CTime()
CTime t2(10,20,30);      // CTime(ushort,ushort,ushort)
CTime t3(32767);         // CTIme(ulong)
Display(t1); Display(t2); Display(t3);
CTime *p1 = new CTime();// dynamic objects
CTime *p2 = new CTime(10, 20, 30);
CTime *p3 = new CTime(32767);
Display(*p1); Display(*p2); Display(*p3);
SAFE_DELETE(p1);
SAFE_DELETE(p2);
SAFE_DELETE(p3);
Display(CTime());        // inline objects
Display(CTime(10,20,30));
Display(CTime(32767));
```

To pass objects by value, the compiler will always generate a copy constructor. A copy constructor is a constructor that accepts a reference to an object of the same type. If you storing values and not pointers to dynamic memory then do not need to create your own copy constructors and let the compiler generate one for you. Whenever you pass or return objects by value, this constructor will be called for the new object to copy the content from the original object. Even though our CTime class does not need one, we will implement our own just to demonstrate the syntax and purpose for this constructor.

Overloading constructors: Time.h

```
class CTime() {
public:
    CTime();
    CTime(CTime& obj);  // copy constructor
    CTime(ushort hour, ushort minute, ushort second);
    CTime(ulong value);
};
```

Implementing the copy constructor: Time.cpp

```
CTime::CTime(CTime& obj) {
    m_hour = obj.m_hour;
    m_minute = obj.m_minute;
    m_second = obj.m_second;
}
```

Any constructor accepts a single value is an assignment constructor. What this means is that you can use an assignment operator to call the constructor as well as create an inline object by specifying the value. This can be disabled by marking the constructor with the **explicit** keyword. The copy constructor is also an assignment constructor since it also accepts one argument.

Calling assignment constructor

```
CTime t4 = 32768;       // CTime(32768);
CTime t5 = t4;          // CTime(t4); copy constructor
Display(32768);         // Display(CTime(32768));
Display(t4);
Display(t5);
```

1.8 Using Windows API

To perform complex tasks, we may need to interact with the operating system. The Windows API is a group of dynamic link libraries that exports functions that we call into the operating system. Include the **windows** header file to access the general services of the Windows operating system. To support compilation to both ANSI and Unicode, you can also include the **tchar** header file. Following example shows how to access the current universal and the local time from the operating system. It is also possible to use the Standard C Library to perform the same operation so that you program can be compiled to run across multiple operating systems.

Including Windows API header files: Time.h

```
#include <windows.h>
#include <tchar.h>

class CTime {
        :
    void SetUniversalTime();
    void SetLocalTime();
        :
};
```

Accessing universal and local time: Time.cpp

```
void CTime::SetUniversalTime() {
    SYSTEMTIME st;
    GetSystemTime(&st);
    m_hour  = (ushort)st.wHour;
    m_minute= (ushort)st.wMinute;
    m_second= (ushort)st.wSecond;
}

void CTime::SetLocalTime() {
    SYSTEMTIME st;
    GetLocalTime(&st);
    m_hour  = (ushort)st.wHour;
    m_minute= (ushort)st.wMinute;
    m_second= (ushort)st.wSecond;
}
```

Using simple functions to access operating system: main.cpp

```
CTime t1;
CTime t2;
t1.SetUniversalTime();
t2.SetLocalTime();
Display(t1);
Display(t2);
```

1.9 Static Members

Not all of the member variables and member functions have to be object-oriented. It is possible to also have static variables and static functions in a class. Functions and variables that are static can be accessed through the class rather than through an object. Thus you do not have to instantiate any object to use static members since they are not accessed as part of any object. Thus **this** keyword is invalid within static functions. The following are two static functions to simplify creating an object to access the current time.

Declaring static functions: Time.h

```
class CTime {
        :
    static CTime GetUniversalTime();
    static CTime GetLocalTime();
        :
};
```

Implementing static functions

```
CTime CTime::GetUniversalTime() {
    CTime time;
    time.SetUniversalTime();
    return time;
}

CTime CTime::GetLocalTime() {
    CTime time;
    time.SetLocalTime();
    return time;
}
```

There will be a naming conflict when you try to compile the **CTime** class. In the **SetLocalTime**, we are calling a global **GetLocalTime** function but since C++ always references the local scope before the global scope, it will actually select the member function instead. Use scope resolution operator without a name to reference members of the global namespace. You should now be able to build the class successfully.

Accessing members from global namespace: Time.cpp

```
void CTime::SetLocalTime() {
    SYSTEMTIME st;
    ::GetLocalTime(&st);
        :
}
```

Static variables can be used to share memory between objects. For example we want to implement an object counter to count objects have already been instantiated but have not yet been destroyed. This is useful for discovering memory leaks. Note that static variables are only declared in class but memory has to be allocated separately in a source file. We will increment the counter in the constructor and decrement the counter in the destructor. The destructor is the same name as the constructor except of a tilde (**~**) operator that is used to indicate the opposite of the constructor. To allow the counter to be fetched without having an object, the function should be marked static.

Declaring static variable and destructor: Time.h

```
class CTime {
        :
    static int m_count;
public:
    ~CTime();   // destructor
    static int GetCount();
};
```

Implementing destructor and static members: Time.cpp

```
int CTime::m_count; // static memory allocation

CTime::CTime() : m_hour(0), m_minute(0), m_second(0) { ++m_count; }
static int CTime::GetCount() { return m_count; }
CTime::~CTime() { --m_count; }
```

Using the object counter

```
char *format = "Count = %ld\n";
printf(format, CTime::GetCount());  // should be 0
CTime *p1 = new CTime();
CTime *p2 = new CTime();
printf(format, CTime::GetCount());  // should be 2
delete p1;
delete p2;
printf(format, CTime::GetCount());  // should be 0
```

2 Exception Handling

2.1 Exception Class

Before C++, functions commonly return error codes to indicate an operation's success or failure. However, to handle such errors, the program has to test the return value of every function that you call. In C++, error handling has become more simple and structured. In C++, functions can use a **throw** statement to signal an error that is called as an exception. A caller can use a **try catch** structure to detect and handle the exception. In the **throw** statement, you can pass a single element that can be a value, reference or a pointer. This can be used to provide information about the actual error that occurred. However rather than throwing a simple value, we may wish to throw objects instead. An object can provide more information about a runtime error that a simple value or string. In the following example, we will declare **CException** class to store an error code, and text strings representing the source and description of the error. The information can be stored in the object through the constructor and retrieved by using accessors.

Declaring an exception class Exception.h

```
#pragma once;
#include <windows.h>
#include <tchar.h>
class CException {
private:
    int m_error;
    LPTSTR m_source;
    LPTSTR m_message;
public:
    CException(int error, LPCTSTR source, LPCTSTR message);
    int GetError() const { return m_error; }
    LPCTSTR GetSource() const { return m_source; }
    LPCTSTR GetMessage() const { return m_message; }
    ~CException(); };
```

The following represents the implementation code of the class. When the source and message are passed to the constructor, instead of just copying the pointers to the character strings, we wish to make a copy of the strings. This will allow an exception object to be kept without caring whether if the strings are dynamically allocated which could have been released before the exception object is used. However we need to first find out the length of the string using **_tcslen** macro so that we know how much memory to allocate. We can then copy strings to the dynamically allocated memory. The destructor will be used to release the memory.

Exception class implementation: Exception.cpp

```
CException::CException(int error, LPCTSTR source, LPCTSTR message) {
    m_error = error;
    size_t len1 = _tcslen(source) + 1;
    size_t len2 = _tcslen(message) + 1;
    m_source = new TCHAR [len1];
    m_message = new TCHAR [len2];
    _tcscpy_s(m_source ,len1, source);
    _tcscpy_s(m_message,len2, message);
}

CException::~CException() {
    delete m_source;
    delete m_message;
}
```

If you do allow an exception object to be copied, you must implement a custom copy constructor since default copy constructor will only copy the pointers to the character strings and not make a duplicate of them. Destroying the copy would release the same memory use by the original object.

Declaring a copy constructor

```
class CException {
public:
    CException(CException& object);
            :
}
```

When the copy constructor is called, we will re-allocate memory to store a copy of the strings from the original object. This will ensure that when the destructor of the copy is called, the destructor would release the memory used by the copy and does not touch the memory used by the original object.

Implementing the copy constructor

```
CException::CException(CException& object) {
    m_error = object.m_error;
    size_t len1 = _tcslen(object.source) + 1;
    size_t len2 = _tcslen(object.message) + 1;
    source = new TCHAR [len1];
    message = new TCHAR [len2];
    _tcscpy_s(source,len1,object.source);
    _tcscpy_s(message,len2,object.message);
}
```

2.2 Exception Handling

We have demonstrated how to generate a runtime error using a **throw** statement. To handle any runtime errors, an application would place the statements where errors can potentially occur within a **try** block. One or more **catch** blocks can be used to handle any exceptions that occur from the **try** block. The value that is thrown can be declared as a parameter to the **catch** section to allow the error information to be retrieved and process. The type of the parameter must be the same type as value thrown to catch the error.

Handling different types of exceptions

```
try {
    func1();// may throw a char*
    func2();// may throw an int
}
catch(char *e) { printf("Error message: %s\n", e); }
catch(int e) { printf("Error code: %ld\n", e); }
```

In our **CTime** class, **SetHour**, **SetMinute** and **SetSecond** functions throws strings to indicate an error. We will now modify the functions to throw exception objects instead. This method allows the application to obtain more information on the runtime error. When you throw an object, you can throw a dynamic or an inline object. The main difference is that you have to destroy the dynamic object manually when you have finished accessing it which is commonly but not compulsorily done in the **catch** section. Throwing an inline object ensures that that the compiler will write code to generate and destroy the copy at the end of the **catch** section. Note that you should catch the inline object by reference to stop making another copy.

Throwing dynamic object

```
throw new CException(-1,_T("main()"),_T("An error has occurred"));
```

Throwing an inline object

```
throw CException(-1,_T("main()"),_T("An error has occurred"));
```

Catching a dynamic object

```
catch(CException *pError) {
    _tprintf(_T("Error message: %s\n"), pError->GetMessage());
     delete pError; } // releasing dynamic object
```

Catching an inline object

```
catch(CException& error) {
    _tprintf(_T("Error message: %s\n"), error.GetMessage()); }
```

If it is not necessary to maintain the exception object after the **catch** section has completed its processing of the error, then throwing a temporary object will be more efficient. We will use this method for the following functions.

Throwing exception objects

```
void CTime::SetHour(ushort hour) {
    if(hour > 23) throw CException(-1,
           _T("SetHour(hour)"), _T("Invalid hour value."));
    m_hour = hour; }

void CTime::SetMinute(ushort minute) {
    if(minute > 59) throw CException(-1,
           _T("SetMinute(minute)"), _T("Invalid minute value."));
    m_minute = minute; }

void CTime::SetSecond(ushort second) {
    if(second > 59) throw CException(-1,
           _T("SetSecond(second)"), _T("Invalid second value."));
    m_second = second; }
```

Catching exception objects

```
try {
    time1.SetHour(10);      // will succeed
    time1.SetMinute(30);    // will succeed
    time1.SetSecond(60);    // will fail!
    puts("Done!");  }       // never executed
catch(CException& ex) {
    _tprinf(_T("Error code %ld in %s : %s\n"),
        ex.GetError(), ex.GetSource(), ex.GetMessage()); }
puts("End!"); // executes after catch
```

You can consider expanding the **CException** class to include also the filename and the line number where the error occurred. This will provide more detailed information for the programmer to locate the error. You do not have to supply the filename or the line number since you can use the **__FILE__** macro and **__LINE__** macro supplied by the compiler to embed the filename and line number into the compiled code. You can also define your macro to simplify the validation and generation of exceptions.

Example usage of FILE and LINE macros

```
if (value < 0) throw CException(-1,_T("SetValue()"),
     _T("Invalid input value."),_T(__FILE__), __LINE__);
```

Simplify validation and throwing exception using macros

```
#define ASSERT(valid, error, source, message) \
    if (!(valid)) throw CException(error, _T(source), _T(message), \
        _T(__FILE__), __LINE__);
```

3 Overloading Operators

3.1 Array Operator

To allow an application to be able to retrieve each value separately, we can treat the class variables as an array, where the array (**[]**) operator can be used together with an index to retrieve a value. Note that it has been marked to return a reference rather then a value that will allow the value to be modified. However, this would defeat hiding data members and providing accessor functions. To create read-only arrays, only return values. Just as an example, we will return references to show you how data members can be modified through an array operator.

Declaration of array operator (Time.h)

```
class CTime {
        :
    ushort& operator [] (int index);
};
```

Implementation of array operator (Time.cpp)

```
ushort& CTime::operator [] (int index) {
    switch(index) {
        case 0 : return m_hour;
        case 1 : return m_minute;
        case 2 : return m_second;
        default: throw CException(-1,
                 _T("CTime::operator[](int)"),
            _T("Index out of range."),
                 _T(__FILE__), __LINE__);
    }
}
```

Using array operator

```
void main() {
    CTime time;
    time[0] = 10; time[1] = 20; time[2] = 30;
    char *format = "%02u:%02u:%02u\n";
    printf(format, time[0], time[1], time[2]);
}
```

3.2 Unary & Binary Operators

If you do allow mathematical operations on your data type class, you can overload the mathematical operators and assignment operators to perform the operations. These are binary operators where a source is applied to a target that is an object from our class type while unary operators operate on only one operand that represent the target of an operation instead. In the following example, we have overloaded the plus assignment (**+=**) operator to allowed one time object to be added to another.

Overloading binary operators (Time.h)

```
class CTime {
        :
    CTime& operator += (CTime& time);
};
```

Implementation of a binary operator: Time.cpp

```
CTime& CTime::operator += (CTime& time) {
    PutValue(GetValue() + time.GetValue()); return *this;
}
```

Note that **this** keyword represents a pointer to the current object. To pass as it as a reference instead of a pointer you can use ***this** instead. Operators return data for the next operation since multiple operators can be use in succession in one expression. In the example below, we have also overloaded the unary pre-increment (**++**) operator. This operator does not supply any arguments but indicates that an increment operation is required on the target.

Class unary operator declaration (Time.h)

```
class CTime {
        :
    CTime& operator ++ ();
};
```

Class unary operator implementation: Time.cpp

```
CTime& CTime::operator ++ () {
    PutValue(GetValue()+1); return *this;
}
```

The above is an implementation of a pre-increment operator. There is also a post-increment operator declared in the following fashion. The parameter is not important but is there to distinguish between the two operators. A post-increment operation requires you to return the value or object before doing any operation on the data. We can do this by making a copy of the object before we update our data members. We can then return the original copy to the next operation.

Post-increment operator declaration: Time.h

```
class CTime {
        :
    CTime& operator ++ (int dummy);
};
```

Post-increment operator implementation: Time.cpp

```
CTime& CTime::operator ++ (int dummy) {
    static CTime temp = *this;  // make copy
    PutValue(GetValue()+1);
    return temp;// return copy
}
```

3.3 Global Operators

We can also overload global operators rather than the operators for the class only. In the following example, we have overloaded the global comparison operator to compare two time objects.

Overloading the global comparison operator: Time.cpp

```
bool operator == (CTime& time1,CTime& time2) {
    return time1.m_hour == time2.m_hour &&
        time1.m_minute == time2.m_minute &&
        time1.m_second == time2.m_second;
}
```

However since this is an external function, it is not allowed access to private members in the class. However, you can mark this function in the class as a **friend** function. The friend functions of a class are allowed access to **private** and **protected** members of a class as well.

Declaring global friend operator function: Time.h

```
class CTime {
        :
    friend bool operator == (CTime& time1,CTime& time2);
};
```

3.4 Overloading Type Operators

You can provide data conversion by overloading type operators in following format. By providing data conversion, we do not have to specially overload other classes to accept a new object type each time we add a new class. C++ will determine which conversion operator will be called. However, when it is ambiguous, we can rely on type casting to highlight the exact type operator function to call.

Format of a conversion operator

```
operator type() {
    return converted_value;
}
```

Providing conversion functions: Time.h

```
class CTime {
        :
    operator ulong();
    operator LPCTSTR();
        :
}
```

Implementing conversion functions: Time.cpp

```
CTime::operator ulong() { return GetValue(); }

CTime::operator LPCTSTR() {
    static TCHAR buffer[10];
    wsprintf(buffer,_T("%02u:%02u:%02u"),
        m_hour,m_minute,m_second);
    return buffer;
}
```

To call right function, typecasting may be required unless the compiler can identify an exact data type. For example, an **int** type conversion will be implied when assigning the object to an **int** variable so no typecasting will be required. However, when you output the value to the **cout** object, we need to typecast to identify the data type we wish to output the object as.

Calling conversion operator functions

```
void main() {
    CTime t(10,20,30);
    ulong v1 = t;   // typecasting not required
    LPCTSTR s1 = t; // typecasting not required
    _tprintf(_T("%lu\n%s\n"),
           (ulong)t,(LPCSTR)t); // typecasting required
}
```