III. Classes (Chap. 3)

As we have seen, C++ data types can be classified as:

- Fundamental (or simple or scalar):
  A data object of one of these types is a single object.
  `int`, `double`, `char`, `bool`, `complex`, and the related types (`unsigned`, `short`, etc.)
  enumerations
- Structured:
  These store collections of data.
  arrays, structs, unions, classes, valarrays, bitsets, and the containers and adapters in STL

We have studied all of the fundamental types (except `complex`) and the data structures C++ gets from C — arrays, structs, and unions. We will now look at classes in detail; pointers (and linked structures that use pointers) and vectors, stacks, queues, and lists from STL will be considered soon.

A. Structs vs. Classes

Similarities between structs and classes

1. Both can be used to model objects with different attributes (characteristics) represented as **data members** (also called **fields** or **instance variables**). They can thus be used to process non-homogeneous data sets.
2. They have essentially the same syntax.

Differences between structs and classes

1. C does not provide classes; C++ provides both structs and classes.
2. Members of a struct by default are **public** (can be accessed outside the struct by using the dot operator).

   In C++ they can be explicitly declared to be **private** (cannot be accessed outside the struct).

   Members of a class by default are private unless explicitly declared to be public.

   Thus, choosing which to use is not based on their capabilites. It is common practice to use classes to prevent users of a new data type from (directly) accessing the data members. (We can also enforce this with structs, but this is not their default nature.)

Differences between "traditional" (C) structs and OOP (C++) structs and classes

C++'s structs and classes are extensions of C's structs. They can be used to model objects that have:

- Attributes (characteristics) represented as **data members**

and

- Operations (behaviors) represented as **function members** (also called **methods**).

Terminology:

It is common to call the two parts of a class *data members* and *member functions* . ("Data members" and "function members" is really more correct.) We will use the terms interchangeably.

This is an important difference because it leads to a whole new style of programming — object-oriented rather than procedural. Objects can now be **self-contained**, carrying their own operations around with them — commonly called the **I can do it myself** principle — instead of having to be shipped off to some external function that operates on them and sends them back.

1. Declaring a Class

a. Usual Form:

```
class ClassName
{
  public:
    Declarations of public members
  private:
    Declarations of private members
};
```

Notes:

1. The data members are normally placed in the **private** section of a class; the function members in the **public** section.

2. Some programmers prefer to put the private section first because this is the default access for classes so the `private:` specifier could be omitted. However, we will put the public *interface* part of the class first and the *hidden* private details last.

3. Although not commonly done, a class may have several private and public sections; the keywords `private:` and `public:` mark the beginning of each.

b. Access

(i) A particular instance of a class is called an **object** :

`ClassName object_name``;`

(ii) Private members can be accessed **only within the class** (except by **friend** functions described later).

(iii) Public members can be accessed **within or outside the class**.

To access them outside the class, one must use the **dot operator**:

`object_name.public_member_name`

c. Where are class declarations placed?

Usually in a header file whose name is *`ClassName`*`.h`. The library is then called a **class library**.

2. Example: Declaring a class `Time` — Version 1

```
/** Time.h -------------------------------------------------------------
 This header file defines the data type Time for processing time.
 Basic operations are:
   Set:     To set the time
   Display: To display the time
--------------------------------------------------------------------*/

#include <iostream>
using namespace std;

class Time
{
/******** Member functions ********/
public:

/* Set sets the data members of a Time object to specified values.
 *
 *  Receive:  hours, the number of hours in standard time
 *            minutes, the number of minutes in standard time
 *            AMPM ('A' if AM, 'P' if PM
 *  Return:   The Time object containing this function with its
 *            myHours, myMinutes, and myAMorPM members set to hours,
 *            minutes, and am_pm, respectively, and myMilTime to
 *            the equivalent military time
 *******************************************************************/

void Set(unsigned hours, unsigned minutes, char am_pm);
/* Display displays time in standard and military format using
 * output stream out.
 *
 *  Receive:      ostream out
 *  Output:       The time represented by the Time object containing
 *                this function
 *  Passes back:  The ostream out with time inserted into it
 ********************************************************************/

void Display(ostream & out) const;


/********** Data Members **********/
private:
   unsigned myHours,
            myMinutes;
   char myAMorPM;        // 'A' or 'P'
   unsigned myMilTime;   // military time equivalent

}; // end of class declaration
```

Notes:

1. The "`my`" in names of data members is simply to remind us of the "I-can-do-it-myself" nature of a class object.

2. The `const` at the end of `Display()`'s prototype makes it a **const function**, which means that it cannot modify any of the data members. It is good practice to protect the data members in this way from accidental modification.

3. Why not make all members public?

   So they can't be accessed outside the class.

   Why? Otherwise programmers may use them in programs, other classes, libraries, . . . However, the data members often are changed to improve storage, simplify algorithms for operations, etc., and all programs, classes, . . . that access them directly must then be modified.

   Results:
   - Increased upgrade time.
   - Increased programmer cost
   - Decreased programmer productivity
   - Reduced profits due to

     — Software late getting on the market — lose out to comptetitors
     — Loss of customer confidence in software reliability

   Therefore:
   Always define data members of a class as private.

   Keeping the data members "hidden" forces programs to interact with an object through its member functions, which thus provide the **interface** between programs and the class. If this does not change, then programs that use an object will not require change.

3. Implementation of a Class

Usually, only the prototypes of the member functions are placed inside the class declaration to avoid cluttering up the intterface — definitions are outside.

However, when a declaration of some public item such as a type, constant, or function is inside a class declaration — another name for "function prototype" is "function declaration" — and is then referenced or defined outside the class declaration, the compiler must be informed where the declaration/prototype is.

This is accomplished using the **scope operator `::`** which has the form

`ClassName::ItemName`

This is referred to as the **qualified** or **full name** of *`ItemName`*.

Example:

```
class Something
{
 public:
   static const int CAPACITY = 100;
   typedef double ArrayType[CAPACITY];

   void Print(ArrayType a, int itsSize);
     . . .
}
. . .

Something::ArrayType x = {0};

for (int i = 0; i < Something::CAPACITY; i++)
    . . .

void Something::Print(Something::ArrayType a, int itsSize)
{ . . . }
```

Traditionally, definitions of member functions have been put in an implementation file *`ClassName`*`.cpp` corresponding to the class' header file. This is done to enforce **data abstraction —** separating the interface of the ADT from its implementation details. (Unfortunately, the class data members, which store data and are therefore part of the implementation, must be in the `.h` file.)

With the increasing use of **templates**, however, this practice is becoming less common because current compiler technology doesn't permit this split for templates — everything has to be in the same file. Thus the reason for dropping the "`.h`" from standard class libraries. They're really class-template libraries, and there are therefore no corresponding "`.cpp`" files.

4. Example: Definitions of Member Functions for class `Time` — Version 1

```
/** Time.cpp -- implements the Time member functions **/

#include "Time.h"

/*** Utility Functions -- Prototypes ***/

int ToMilitary(unsigned hours, unsigned minutes, char am_pm);

//----- Function to implement the Set operation -----

void Time::Set(unsigned hours, unsigned minutes, char am_pm)
{
  // Check class invariant
  if (hours >= 1 && hours <= 12 &&
      minutes >= 0 && minutes <= 59 &&
      (am_pm == 'A' || am_pm == 'P'))
  {
    myHours = hours;
    myMinutes = minutes;
    myAMorPM = am_pm;
    myMilTime = ToMilitary(hours, minutes, am_pm);
  }
  else
    cerr << "*** Can't set time with these values ***\n";
    // Object's data members remain unchanged
}

//----- Function to implement the Display operation -----

void Time::Display(ostream & out) const
{
  out << myHours << ':'
      << (myMinutes < 10 ? "0" : "") << myMinutes
      << ' ' << myAMorPM << ".M.  ("
      << myMilTime << " mil. time)";
}
```

```
/*** Utility Functions -- Definitions ***/

/* ToMilitary converts standard time to military time.
 *
 *  Receive: hours, minutes, am_pm
 *  Return:  The military time equivalent
 ********************************************************/

int ToMilitary (unsigned hours, unsigned minutes, char am_pm)
{
  if (hours == 12)
    hours = 0;

  return hours * 100 + minutes + (am_pm == 'P' ? 1200 : 0);
}
```

5. Testing the class

```
// Test driver

#include "Time.h"
#include <iostream>
using namespace std;

int main()
{
  Time mealTime;

  mealTime.Set(5, 30, 'P');

  cout << "We'll be eating at ";

  mealTime.Display(cout);

  cout << endl;
}
```

Execution

```
We'll be eating at 5:30 P.M.  (1730 mil. time)
```

Again, note the difference from the procedural approach. Rather than package up the object and send it off to some function for processing, we **send a message to the object to operate on itself.** To set my digital watch to 5:30 P.M., I don't wrap it up and mail it off to Casio and have them do it; rather, I push a button! To display the time, I don't wrap up my watch and mail if off to Casio and have them tell me what time it is. Ridiculous! I have it display the time to me itself, perhaps pushing a button to turn on the backlight so I can see it .

6. Some Notes

a. Member functions: "Inside" an object so don't pass object to them as a parameter. Another way to view this:

> *They receive the class object to be operated on implicitly, rather than explicitly via a parameter.*

Non-member functions: "Outside" an object, so *to operate on an object, they must receive it via a parameter*.

b. Public items like types and constants declared inside a class declaration must be qualified with the class name when used outside the class:

```
ClassName::ItemName
```

Constants are usually specified to be static so this is a global class property that can be accessed by all objects of that class type rather than having each such object carry around it's own copy of that constant.

c. Nontrivial member function:

Usually: Prototypewithin the class
Define outside the class; must qualify its name:

```
ClassName::FunctionName(. . .)
```

d. Simple member function:

Usually: Specify that it be an **`inline`** function, which *suggests* to the compiler that it replace a function call with the actual code of the function with parameters replaced by arguments, thus avoiding the usual overhead of a function call. This can be done in two ways:

1. Prototype the function inside the class declaration as usual, but **define it as inline below the class declaration in the header file, qualifying its name as usual**:

In *ClassName*.h

```
class ClassName
{
  // Public section -- function members
          . . .
        RetType SimpleFun(param_list);
          . . .
  // Private section -- data members
            . . .
};

inline RetType ClassName::SimpleFun(param_list)
{
    // function body
}
```

2. **Simply define the function inside the class declaration.** In this case, it need not be prototyped, its name need not be qualified, and the compiler will treat it as an inline function:

    In *ClassName*.h

```
 class ClassName
 {
   // Public section -- function members
        . . .
      RetType SimpleFun(param_list)
      {
         // function body
      }
             . . .
   // Private section -- data members
 };
```

    But use this method only for simple functions to avoid interface clutter.

e. In `Set()`, we tested whether the arguments are valid:

```
  if (hours >= 1 && hours <= 12 &&
      minutes >= 0 && minutes <= 59 &&
      (am_pm == 'A' || am_pm == 'P'))
 {
   myHours = hours;
   myMinutes = minutes;
   . . .
 {
 else . . .
```

This is to ensure that the following **class invariant** is true:

1 ≤ `myHours` ≤ 12 && 0 ≤ `myMinutes` ≤ 59 &&
`myAMorPM` == 'A' or 'P' && 0 ≤ `myMilTime` ≤ 2359

This class invariant is intended to guarantee that the data members always contain valid values so that other function members can be sure of this Thus, whenever an operation modifies any of the data members, we should always check that the class invariant still holds.

An alternative way to test this is to use the **`assert()`** mechanism (from **`<cassert>`**)— at least during debugging — which:

- Accepts a boolean condition;
- If that condition is true, execution continues as usual.
- If the condition is false, execution halts and an error message is displayed.

```
 #include <cassert>
using namespace std;
     . . .
//----- Function to implement the Set operation -----
void Time::Set(unsigned hours, unsigned minutes, char am_pm)
{
  assert(hours >= 1 && hours <= 12 && minutes <= 59 &&
         (am_pm == 'A' || am_pm == 'P'));
   myHours = hours;
   myMinutes = minutes;
     . . .
}
```

Testing:

If we change `driver.cpp` as: `mealTime.Set(13, 30, 'P');`
execution terminates with the following message:

```
Time.cpp :11: failed assertion `hours >= 1 && hours <= 12 &&
minutes <= 59 && (am_pm == 'A' || am_pm == 'P')'
IOT trap
```

A third alternative is to **throw an exception** that the calling function can **catch** and take appropriate action:

```
//----- Function to implement the Set operation -----
void Time::Set(unsigned hours, unsigned minutes, char am_pm)
{
  // Check class invariant
  if (hours >= 1 && hours <= 12 &&
      minutes >= 0 && minutes <= 59 &&
      (am_pm == 'A' ||am_pm == 'P'))
  {
     . . .
  }
  else
  {
     char error[] = "*** Illegal initializer values ***\n";
     throw error;
  }
}
```

To catch this exception, a calling function might contain

```
try
{
  mealTime.Set(13, 30, 'P');
  cout << "This is a valid time\n";
}
catch (char badTime[])
{
  cout << "ERROR: " << badTime << endl;
  exit(-1);
}
cout << "Proceeding. . .\n";
```

When executed, the output produced will be

```
ERROR: *** Illegal initializer values ***
```

7. Class Constructors

a. Recall that *constructing* an object consists of:

(1) **allocating memory** for the object, and
(2) **initializing** the object.

In our example, after the declaration

```
Time mealTime;
```

memory has been allocated for `mealTime`, but it's data members are not initialized (and are likely to contain "garbage" values). It would be better if:

- the programmer could specify initial values for `mealTime`
- default values were used if no initial values are specified.

b. This can be accomplished using **constructor functions**.
Properties:

(1) Their primary role (for now) is to **initialize the data members** of an object with values (either default values or values provided as arguments).

(2) Their names are always the same as the **class name**.

(3) They are always function members and are (almost always) prototyped in the public section.

(4) They do not return a value; they have **no return type** (not even `void`).
For this reason, documentation that describes their behavior commonly specifies:
1. What values they receive (if any) via parameters:
2. Preconditions: Conditions that must be true before the function is called.
and
3. Postconditions: Conditions that must be true when the function terminates.

(5) Often they are quite simple and can be inlined in either of the two ways described earlier.

(6) Constructors get called **whenever an object is declared.**

(7) If no constructor is given in the class, **the compiler supplies a default constructor** which allocates memory and initializes it with some default (possibly garbage) value.

A *default constructor* is one that is used when the declaration of an object contains no initial values:

```
ClassName object_name;
```

(8) If we supply a constructor for a class, then we must also provide a default constructor.

c. Example: Constructors for `Time` class

In `Time.h`

```
. . .
class Time
{
/******** Member functions ********/
public:

/***** Class constructors *****/

/* --- Construct a class object (default).
 *  Precondition:  A Time object has been declared.
 *  Postcondition: The Time object is initialized to 12:00 A.M.;
 *                 that is, the myHours, myMinutes, and myAMorPM
 *                 members are initialized to 12, 0, 'A', respectively,
 *                 and myMilTime to 0.
 **********************************************************************/

Time();

/* --- Construct a class object (explicit values).
 *  Precondition:  A Time object has been declared.
 *  Receive:       Initial values initHours, initMinutes, and
 *                 initAMPM
 *  Postcondition: The myHours, myMinutes, and myAMorPM members
 *                 of theTime object are initialized to initHours,
 *                 initMinutes, and initAMPM , respectively, and
 *                 myMilTime to the corresponding military time.
 **********************************************************************/

Time(unsigned initHours, unsigned initMinutes, char initAMPM);

  . . .
// other member function prototypes
  . . .
/********** Data Members **********/
private:
  . . .
}; // end of class declaration

inline Time::Time()
{
  myHours = 12;
  myMinutes = 0;
  myAMorPM = 'A';
  myMilTime = 0;
}
```

or

class Time

```
/******** Function Members ********/
public:

/***** Class constructors *****/

/* --- Construct a class object (default).
...
----------------------------------------------------------------------*/

Time()
{
  myHours = 12;
  myMinutes = 0;
  myAMorPM = 'A';
  myMilTime = 0;
}

/* --- Construct a class object (explicit values).
...
-------------------------------------------------------------------*/
  . . .

// other member function prototypes
  . . .
/********** Data Members **********/
private:
  . . .
}; // end of class declaration
```

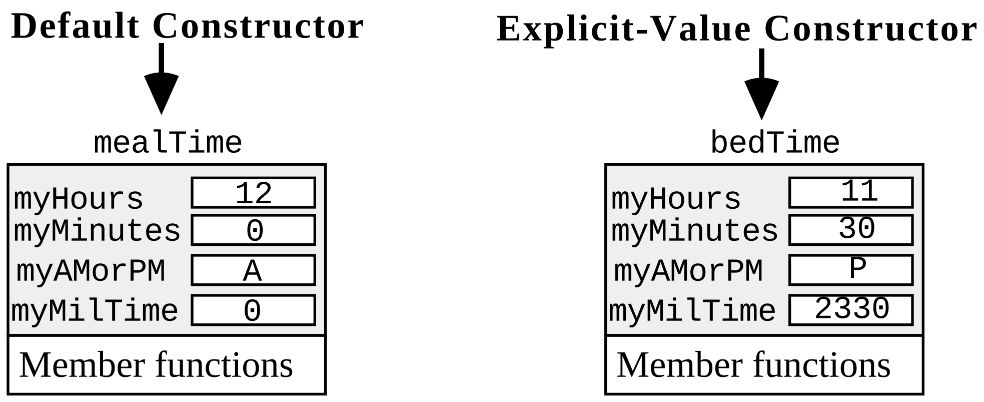
Add to `Time.cpp`

```
#include <cassert>
using namespace std;
 . . .

//----- Function to implement the explicit-value  constructor -----

Time::Time(unsigned initHours, unsigned initMinutes, char initAMPM)
{
  // Check class invariant
  assert(initHours >= 1 && initHours <= 12 &&
         initMinutes >= 0 && initMinutes <= 59 &&
         (initAMPM == 'A' || initAMPM == 'P'));

  myHours = initHours;
  myMinutes = initMinutes;
  myAMorPM = initAMPM;
  myMilTime = ToMilitary(initHours, initMinutes, initAMPM);
}
```

Testing # 1

```
Time mealTime,
     bedTime(11,30,'P');
```

Creates and initializes 2 Time objects:



```
mealTime.Display(cout);
cout << endl;
bedTime.Display(cout);
cout << endl;
```

Execution:

```
12:00 A.M. (0 mil. time)
11:30 P.M. (2330 mil. time)
```

Testing # 2

```
Time mealTime,
     bedTime(13,0,'P');
```

Execution:

```
Time.cpp:12: failed assertion
`initHours >= 1 && initHours <= 12 &&
 initMinutes <= 59 &&
(initAMPM == 'A' || initAMPM == 'P')'
 IOT trap
```

Note: We could combine both constructors into a single constructor function by using **default arguments**:

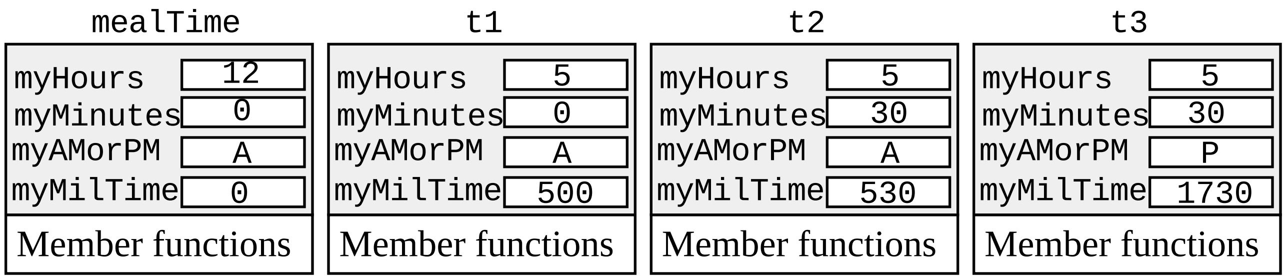
Replace constructors in `Time.h` with:

```
/* --- Construct a class object.
   Precondition:  A Time object has been declared.
   Receive:       Initial values initHours, initMinutes, and
                  initAMPM (defaults 12, 0, 'A')
   Postcondition: The myHours, myMinutes, and myAMorPM members of
                  the Time object are initialized to initHours,
                  initMinutes, and initAMPM , respectively.
-----------------------------------------------------------------*/

Time(unsigned initHours = 12, unsigned initMinutes = 0,
      char initAMPM = 'A');
```

Testing:

```
Time mealTime,
     t1(5), t2(5, 30), t3(5, 30, 'P');
```

Creates 4 Time objects:



```
mealTime.Display(cout);
cout << endl;
t1.Display(cout); cout << endl;
t2.Display(cout); cout << endl;
t3.Display(cout); cout << endl;
```

Execution:

```
12:00 A.M. (0 mil. time)
5:00 A.M. (500 mil. time)
5:30 A.M. (530 mil. time)
5:30 P.M. (1730 mil. time)
```

Question: What happens with the declaration

```
Time t(5, 'P');
```

Will it create `Time` object `t` with values `5, 0, 'P'` in its data members?

No — compilation error.

All parameters with default arguments must appear after all parameters without default arguments.

9. Copy Operations

Two default copy operations are provided:

1. Copy in **initialization**
2. Copy in **assignment**

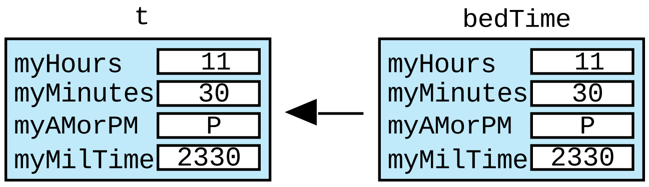
Each makes a (byte-by-byte) copy of the data members of the object.

Examples:

```
Time t = bedTime;
Time t(bedTime);
```

Both:

1. Allocate memory for `t`
2. Copy data members of `bedTime` into them so `t` is a copy of `bedTime` :



```
Time t = Time(11, 30, 'P');
```

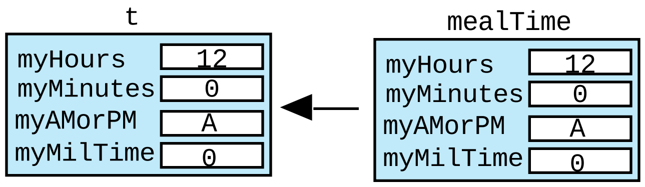
also does: Right side calls the explicit-value constructor to construct a (temporary) `Time` object and then copies it into `t`.

Note: These are *not* assignments; a default ***copy constructor*** is called.

There is a default copy operation for assignment.
Example:

```
t = mealTime;
```

copies the members of `mealTime` into `t`, replacing any previous values:



9. Access (Extractor) Functions

Data members are private; they cannot be accessed outside the class. It is often necessary, however, to make the values stored in some or all of these members accessible. For this, **access** (or **extractor**) member functions can be provided.

Example:

Problem: To add extractors to class `Time`.
(We will do this only for the `myHours` member; the others are essentially the same.)

Specification:
Receives: A `Time` object
Returns: The value stored in the `myHours` member of that object

As usual, the specification tells us how to prototype the function:

- If we declare it as a member function, then it will be "inside" the `Time` object and so no parameters (`Time` or otherwise) will be needed.
- The function returns `myHours`, which is an integer.

In addition, because this function simply retrieves the value stored in a data member, it is simple enough to inline.

Also because it does not modify any data members it should be prototyped (and defined) as a **const** function.

Add to `Time.h`

```
class Time
{
/********** Data Members **********/
   . . .
/******** Member functions ********/
public:
   . . .
/***** Data Retrievers *****/

/* Hour Accessor
 *  Receive: The Time object containing this function (implicitly)
 *  Return:  The value stored in the myHours member of the Time
 *           object containing this function
 ****************************************************************/

unsigned Hour() const;

// and similar functions for myMinutes, myAMorPM, and myMilTime retrieval
   . . .
/********** Data Members **********/
 private:
   . . .
}; // end of class declaration
   . . .

//----- Definition of Hour()
inline unsigned Time::Hour() const;
{ return myHours; }
```

Testing:

```
Time mealTime;                                  ||   Execution:
   . . .                                        ||
cout << "Hour: " << mealTime.Hour() << endl;||   Hour: 12
   . . .                                        ||
```

9. Output and Input — Overloading Operators — Friend Functions

Add output operation to a class early so that it can be used for debugging.

It is convenient to overload `operator<<` for a `Time` object so we can write

```
cout << "We'll be eating at " << mealTime << endl;
```

instead of

```
cout << "We'll be eating at " ;
mealTime.Display(cout);
cout << endl;
```

a. Overloading operators:

— In C++, operator `Δ` can be implemented with the function **`operatorΔ()`**

— If a member function of a class C, and `a` is of type C, the compiler treats `a Δ b` as

`a.operatorΔ(b)`

— If not a member function of a class C the compiler treats `a Δ b` as

`operatorΔ(a, b)`

b. Overloading Output Operator <<

Can `operator<<()` be a member function? No, because the compiler will treat

```
cout << t
```

as

```
cout.operator<<(t)
```

which would mean that `operator<<()` would have to be a member of class `ostream`!

Putting the prototype

```
ostream & operator<<(ostream & out, const Time & t);
```

inside the class declaration causes a compiler error like:

```
`Time::operator <<(ostream &, const Time &)' must take
  exactly one argument
```

because making `operator<<()` a *member* function of `Time` means that it already has the `Time` object containing it as an (implicit) parameter, so it can't have two more.

Option 1: Put its prototype in the header file `Time.h` but outside the class declaration.
and it's definition in `Time.cpp`

Actually,
because it is so simple, we inline it by putting it's definition in `Time.h`:

```
class Time
{
 public:    // documentation omitted to save space here
   Time();
   Time(unsigned initHours, unsigned initMinutes, char initAMPM);
   int Hour() const{ return myHour; }
   int Minute() const{ return myMinute; }
   char AMPM() const{ return myAMorPM; }
   int MilTime() const{ return myMilTime; }
   void Display(ostream & out);
 private:
   unsigned myHours,
            myMinutes;
   char myAMorPM;        // 'A' or 'P'
   unsigned myMilTime;   // military time equivalent
}; // end of class declaration

/* --- operator<< displays time in standard and military format
   using ostream out.
   Receives:     An ostream out and a Time object t
   Output:       The time represented by the Time object t
   Passes back:  The ostream out with t inserted into it.
   Return value: out
-------------------------------------------------------------------*/
inline ostream & operator<<(ostream & out, const Time & t)
{
   t.Display(out);
   return out;
}
```

- Why 1st parameter a reference parameter?
  The ostream gets modified so must be passed back.

- Why 2nd parameter a const reference parameter?
  To avoid the overhead of having to copy a class object.

- Why return a reference to `out`?
  So we can chain output operators. For example:

```
                                                  Output
cout << t1 << endl << t2 << endl;          5:00 A.M. (500 mil. time)
                                           5:30 A.M. (530 mil. time)
```

Because `<<` is left-associative, this is evaluated as
```
  operator<<(cout, t1) << endl << t2 << endl;
```
So first function must return `cout` so expression becomes
```
  cout << endl << t2 << endl;
```
which is evaluated as
```
     operator<<(cout, endl) << t2 << endl;
```
. . .

Option 2: Replace `Display()` with `operator<<` :

Replace the prototype of `Display()` in `Time.h`

```
class Time
{
   . . .
 public:
   . . .

  /***** I/O Functions *****/

  /* --- operator<< displays time in standard and military format
     using ostream out.
     Receives:     An ostream out and a Time object t
     Output:       The time represented by t
     Passes back:  The ostream out with representation of t
                   inserted into it.
     Return value: out
  ------------------------------------------------------------------*/

  friend ostream & operator<<(ostream & out, const Time & t);

};
```

And replace the definition of `Display()` in `Time.cpp`

```
//----- Function to implement ostream output -----

ostream & operator<<(ostream & out, const Time & t)
{
   out << t.myHours << ':'
       << (t.myMinutes < 10 ? "0" : "") << t.myMinutes
       << ' ' << t.myAMorPM << ".M.  ("
       << t.myMilTime << " mil. time)";

   return out;
}
```

A function that a class names as a *friend* is a: **non-member function to which the class has granted permission to access members in its private sections.**

Note: Because a friend function is not a function member:

- It's definition is not qualified using the class name and the scope operator (`::`).
- It receives the time object on which it operates as a parameter
- It uses the dot operator to access the data members.

b. Input

To add an input operator to our `Time` class, we proceed in much the same way as for output. We could either:

1. Add a member function `ReadTime()` that reads values and stores them in the data members of a `Time` object; then call it from non-member function `operator>>()`
2. Declare `operator>>()` to be a friend function so that it can access the data members of a `Time` object and store input values in them.

10. Adding Relational Operators:

We will describe how to add only one of the relational operators — less than — the others are similar.

Specification:
Receives: Two `Time` objects
Returns: True if the first `Time` object is less than the second; false otherwise.

Question: Should it be a member function?

From an internal perspective: I compare myself with another `Time` object and determine if I am less than that other object

From an external perspective: Two `Time` objects are compared to determine if the first is less than the second.

Answer: Either will work, but in keeping with the OOP "I-can-do-it-myself" principle of making objects self-contained, we usually opt for using member functions whenever possible.

In this case, we might better rephrase our specification as:

Receives: A `Time` object (and the current object implicitly)
Returns: True if I (the `Time` object containing this function) am less than the `Time` object received; false otherwise.

Add to `Time.h`:

```
  ...
class Time
{
 public:   // member functions
  ...
/***** Relational operators *****/
/* --- operator< determines if one Time is less than another Time
   Receive:  A Time t (and the current object implicitly)
   Return:   True if time represented by current object is < t.
------------------------------------------------------------------*/

bool operator<(const Time & t);
 ...
}; // end of class declaration

inline bool Time::operator<(const Time & t)

{ return myMilTime < t.myMilTime; }
```

For the external perspective:

```
  ...
class Time
{
 public:   // member functions
  ...
/***** Relational operators *****/
/* --- operator< determines if one Time is less than another Time
   Receive:  Two Times t1 and t2
   Return:   True if time t1 is less than time t2/
------------------------------------------------------------------*/

friend bool Time::operator<(const Time & t1, const Time & t2);
 ...
}; // end of class declaration

inline bool operator<(const Time & t1, const Time & t2)

{ return t1.myMilTime < t2.myMilTime; }
```

12. Adding Increment/Decrement Operators:

Specification:
Receives: A `Time` object (perhaps implicitly)
Returns: The `Time` object with minutes incremented by 1 minute.

Question: Should it be a member function? Yes

Add to `Time.h`:

```
/***** Increment operator *****/
/* --- Advance() increments a Time by 1 minute.
   Receive:   Current time object (implicitly)
   Pass back: The Time object with its minutes incremented by 1.
--------------------------------------------------------------------*/
void Advance();
```

Add to `Time.cpp`:

```
//----- Function to implement Advance() -----
void Time::Advance()
{
  myMinutes++;
  myHours += myMinutes / 60;
  myMinutes %= 60;
  if (myMilTime == 1159)
     myAMorPM = 'P';
  else if (myMilTime == 2359)
     myAMorPM = 'A';
  // else no change
  myMilTime = ToMilitary(myHours, myMinutes, myAMorPM);
}
```

We could overload `operator++()` but how do we distinguish between prefix ++ and postfix ++?

Solution: In C++,
`operator++()` with no parameters is prefix
`operator++(int)` with one int parameter is postfix
[The `int` parameter is not used in the definition.]

13. Problem of Redundant Declarations

A class like `Time` might be used in a program, libraries, other class libraries, and so it could easily happen that it gets included several times in the same file —

e.g.,
Program needs `Time` class, so it `#include`s `"Time.h"`
Program also needs library `Lib`, so it `#include`s `"Lib.h"` . . . but `Lib.h` also `#include`s `"Time.h"`

This would cause "redeclaration" errors during compiling.

How do we prevent the declarations in `Time.h` from being included more than once in a file?

Use **conditional compilation**

Wrap the declarations in `Time.h` inside preprocessor directives like the following:

[The preprocessor scans through a file removing comments, `#includ`ing files, and processing other directives (which begin with `#`)before the file is passed to the compiler.]

```
#ifndef TIME  ←¾¾¾  Usually the name of the class in all caps
#define TIME
     :
     :
#endif
```

The first directive tests to see whether the identifier `TIME` has been defined.
If it has not:
Processing proceeds to the second directive, which defines `TIME` (to be 1), and then continues on through what follows and on to the `#endif` and beyond.
If it has been defined:
The preprocessor removes all code that follows until a `#elif`, `#else`, of `#endif` directive is encountered.

Thus, the first time the preprocessor encounters a class declaration like `Time`, it defines the name `TIME`. If it encounters the class declaration again, since `TIME` has been defined, all code between `#ifndef TIME` and `#endif` is stripped, thus removing the redeclaration.