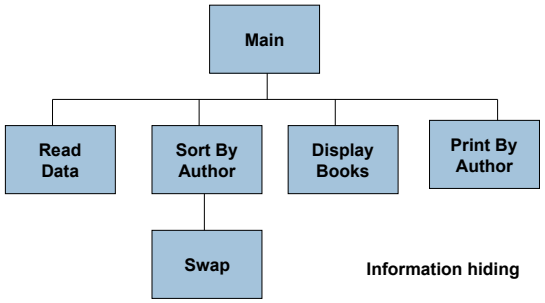
ABSTRACT DATA TYPE (ADT)

Class and objects

Functional Abstraction



Abstraction

- **Abstraction** is the **separation** of the essential qualities of an object from the details of how it works or is composed
  - Focuses on **what, not how**
  - Necessary for managing large, complex software projects

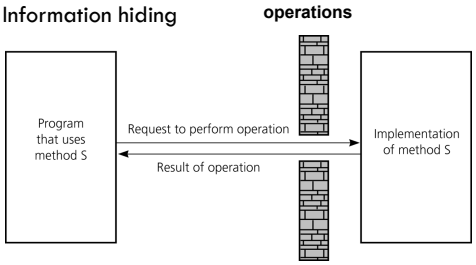
Data Abstraction

- **Data abstraction** separates the logical properties of a data type from its implementation

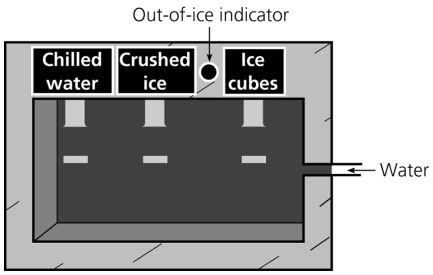
LOGICAL PROPERTIES	IMPLEMENTATION
What are the possible values?	How can this be done in C++?
What operations will be needed?	How can data types be used?

Data Abstraction

- Abstract Data Type (ADT)
  = Data + Operations on Data
- Information hiding



An Example ADT



A dispenser of chilled water, crushed ice, and ice cubes

Abstract Data Type (ADT)

- **An abstract data type is a data type whose properties (domain and operations) are specified (*what*) independently of any particular implementation (*how*)**

For example . . .

ADT Specification Example

TYPE

Time

DOMAIN

Each Time value is a time in hours, minutes, and seconds.

OPERATIONS

Set the time

Print the time

Increment by one second

Compare 2 times for equality

Determine if one time is "less than" another

Another ADT Specification

TYPE

ComplexNumber

DOMAIN

Each value is an ordered pair of real numbers (a, b) representing a + b*i*

Another ADT Specification, cont...

OPERATIONS

Initialize the complex number
Write the complex number
Add
Subtract
Multiply
Divide
Determine the absolute value of a complex number

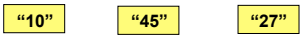
ADT Implementation

- **ADT implementation**
  - Choose a specific data representation for the abstract data using data types that already exist (built-in or programmer-defined)
  - Write functions for each allowable operation

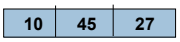
Several Possible Representations of ADT `Time`

3 int variables

3 strings



3-element int array



Choice of representation depends on time, space, and algorithms needed to implement operations

Some Possible Representations of ADT `ComplexNumber`

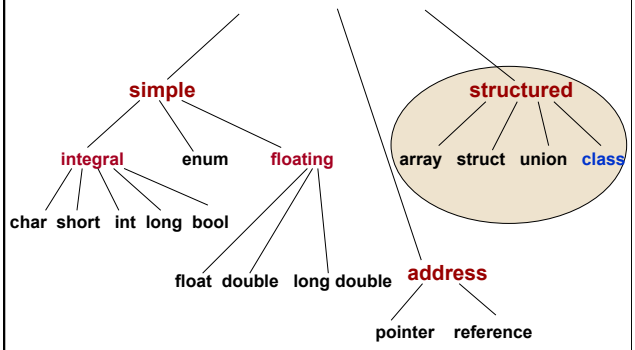
struct with 2 float members

-16.2	5.8
.real	.imag

2-element float array

-16.2	5.8

C++ Data Types



C++ `class` Type

- Facilitates re-use of C++ code for an ADT
- Software that uses the class is called a client
- Variables of the class type are called class objects or class instances
- Client code uses class's public member functions to manipulate class objects

Separate Specification and Implementation


```
// Specification file "time.h"
// Specifies the data and function members
class Time
{
public:
                . . .

private:
                . . .
 };
```

class Time Specification

18

```
//  Specification file (Time.h)

class  Time      // Declares a  class data type
{
                 //  does not allocate memory

public :         // Five public function members
    void  Set (int  hours , int  mins , int  secs);
    void  Increment ();
    void  Write ()  const;
    bool  Equal (Time   otherTime)  const;
    bool  LessThan (Time   otherTime)  const;
private :        // Three private data members
    int  hrs;
    int  mins;
    int  secs;
};
```

Things to remember

19

- The class declaration creates a data type and names the members of the class
- It does not allocate memory for any variable of that type!
- Client code still needs to declare class variables
- Data members are generally private
- Function members are generally declared public
- Private class members can be accessed only by the class member functions (and friend functions), not by client code

Client Code Using Time

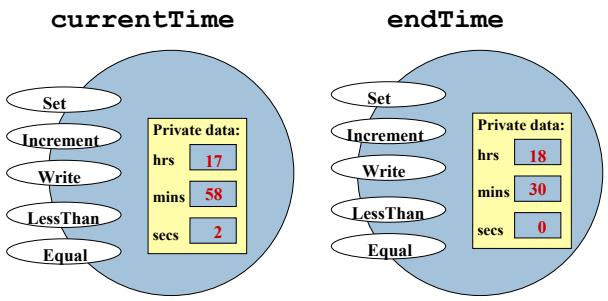
20

```
#include   "time.h";
using  namespace  std;
int  main ()
{
    Time  currentTime;
    Time  endTime;
    bool  done  =  false;

    currentTime.Set (5, 30, 0);
    endTime.Set (18, 30, 0);
    while  (! done)
    {
        currentTime.Increment ();
        if  (currentTime.Equal (endTime))
           done  =  true;
    };
}
```


Time Class Instance Diagrams

21



Class Constructors

22

- A **class constructor** – a member function whose purpose is to initialize the private data members of a class object
- The name of a constructor is always the name of the class, and there is no return type for the constructor

Class Constructors

23

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

Specification of `Time` Class Constructors

24

```
class  Time     // Time.h
{ public :      //  7 function members
    void Set(int hours, int minutes,
             int seconds);
    void Increment();
    void Write()  const;
    bool Equal(Time otherTime)  const;
    bool LessThan(Time otherTime)  const;
     // Parameterized constructor
    Time (int initHrs, int initMins,
          int initSecs);
    Time(); // Default constructor

  private :     //  3 data members
    int hrs;
    int mins;
    int secs;
};
```

Implementation of `Time` Default Constructor

```
Time::Time ()
{
    hrs  = 0;
    mins = 0;
    secs = 0;
}
```

Parameterized Constructor

26

```
Time::Time( /* in */   int   initHrs,
            /* in */   int   initMins,
            /* in */   int   initSecs)
{
    hrs  =  initHrs;
    mins =  initMins;
    secs =  initSecs;
}
```

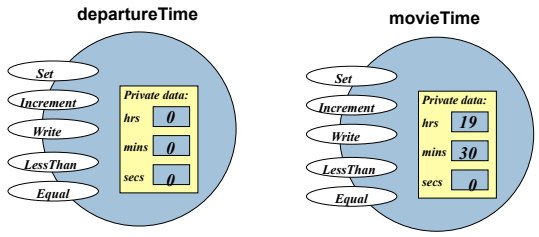
Automatic invocation of constructors occurs

27

In the client program:

```
Time departureTime;  // Default constructor invoked

Time movieTime (19, 30, 0); // Parameterized constructor
```



Example Using `const` with a Member Function

28

```
void Time::Write ()   const
{
    if  (hrs < 10)
        cout << '0';
    cout  << hrs  <<  ':';
    if  (mins < 10)
        cout << '0';
    cout  << mins  <<  ':';
    if  (secs < 10)
        cout << '0';
    cout  << secs;
}
```

Use of `const` with Member Functions

29

- When a member function does not modify the private data members:
- Use `const` in both the function prototype (in specification file) and the heading of the function definition (in implementation file)

Implementation File for Time

```
// Implementation file "time.cpp"
// Implements the Time member functions.
// Also must appear in client code
#include " time.h"
#include  <iostream>
    . . .
bool Time::Equal(/* in */  Time otherTime) const

// Postcondition:  Return value == true,
//     if this time equals otherTime,
//         otherwise == false
{
    return ((hrs == otherTime.hrs)
          && (mins == otherTime.mins)
          && (secs  == otherTime.secs));
}
```

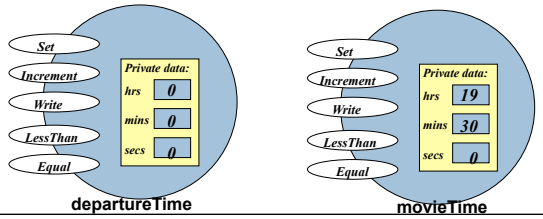
Equal member function in the client program

In the client program:

```
Time departureTime;  // Default constructor invoked

Time movieTime (19, 30, 0); // Parameterized constructor

if (departureTime.Equal(movieTime))
    cout << "Time to watch the movie!" << endl;
```



Practice Question

- **How to implement the member function:**

```
bool LessThan(Time otherTime) const;
```

Avoiding Multiple Inclusion of Header Files

- **Often several program files use the same header file containing typedef statements, constants, or class type declarations**
- **But, it is a compile-time error to define the same identifier twice within the same namespace**

Avoiding Multiple Inclusion of Header Files

- This preprocessor directive syntax is used to avoid the compilation error that would otherwise occur from multiple uses of #include for the same header file

```
#ifndef   Preprocessor_Identifier
#define  Preprocessor_Identifier
   .
   .
   .
#endif
```

Example Using Preprocessor Directive `#ifndef`


```
// time .h
// Specification file
#ifndef   TIME_H
#define  TIME_H

class Time
{
    public:
     . . .

    private:
     . . .
};
#endif
```

For compilation the class declaration in File time.h will be included only once

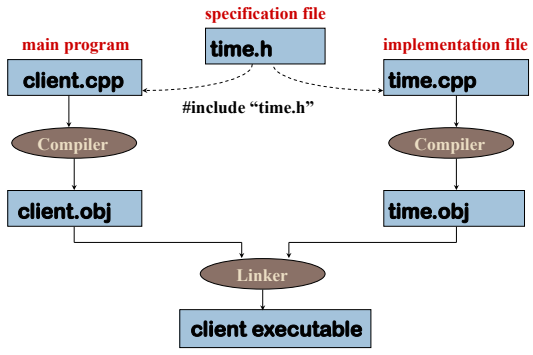
```
// time .cpp
// IMPLEMENTATION FILE
#include "time.h"

          . . .
```

```
// client.cpp
// Appointment program
#include "time.h"

 int   main (void)
 {
      . . .
 }
```

Separate Compilation and Linking of Files



Aggregate `class` Operations

- **Built-in operations valid on class objects are:**
  - Member selection using dot (.) operator ,
  - Assignment to another class variable using (=),
  - Pass to a function as argument (by value or by reference),

Aggregrate `class` Operations

- **Built-in operations valid on class objects a also:**
  - **Return as value of a function**

  Other operations can be defined as class member functions