



CS 215: Introduction to Program Design, Abstraction and Problem Solving

Chapter 9 Classes

Basic Operations

Insertion

Deletion

Searching for a key

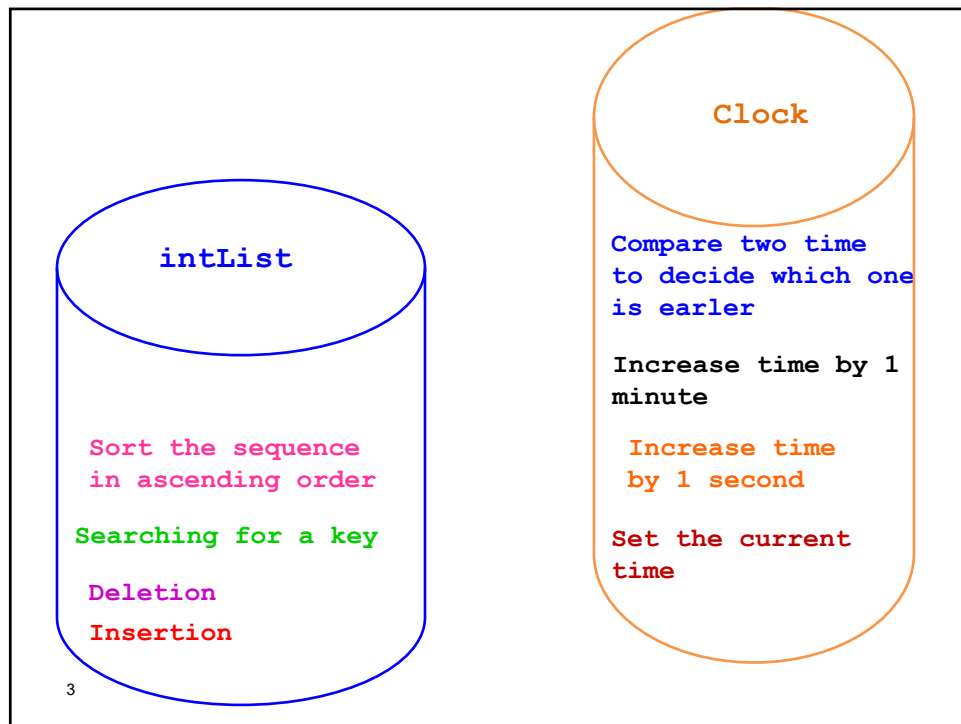
Set the current time

Increase time
by 1 second

Compare two time
to decide which one
is earlier

Increase time by 1 minute

Sort the sequence in ascending order



Objects to the rescue

- Computer scientists noticed that most often functions were working on related data so they invented *objects*, where they keep the data and the functions that work with them together.
- Some new terminology:
 - ▶ The data stored in an object are called *data members* or *member variables*.
 - ▶ The functions that work on data members are called *methods* or *member functions*.

Abstract Data Types

- A typical program consists of a set of data values and operations. In many cases, some of the [data values](#) and [operations](#) are closely related and hence can be grouped together to describe a common object.
 - Data values (state or attributes)
 - Operations on these data values (behaviors)

TYPE

IntList

DOMAIN

Each IntList value is a collection of up to 100 separate integer numbers.

OPERATIONS

- Insert an item into the list
- Delete an item from the list
- Search the list for an item
- Sort the list into ascending order
- Print the list

TYPE

TimeClock

DOMAIN

Each TimeType value is a time of the day in the form of hours, minutes, and seconds

OPERATIONS

- Set the time
- Print the time
- Increment the time by one second
- Compare two times for equality
- Determine if one time comes before another

Class Fundamentals

- A [class](#) is a template that defines a set of objects with the same behavior
 - Class specifies both the data and the code that operates on that data.
 - [Data members](#) of the class
 - [Functions](#) of the class.
- An [object](#) is an instance of a class.
 - A class is a set of objects with the same behavior
 - A class is a logical abstraction. A physical representation of a class does not exist until an object of that class is declared.

Class Fundamentals – Cont'd

- A class normally consists of one or more functions that manipulate the attributes that belong to a particular object of the class
- Data members of a class
 - Attributes represented as variables in a class declaration
 - Declared inside a class declaration but outside the bodies of the class's function declarations
 - When each object maintains its own copy of an attribute, the data members are also known as *instance variables* – each object (instance) of the class has a separate instance of the variables in memory
- Member functions of a class
 - Sequence of instructions that accesses the data members of an object
 - You manipulate an object by calling its member functions

Encapsulation

- The data members are **encapsulated**, which means they are hidden from other parts of the program and accessible only through their own member functions.
- When we want to change the way that an object is implemented, only a small number of functions need to be changed, and they are the ones in the object.
- Because most real-world programs need to be updated often during their lifetime, this is an important advantage of object-oriented programming.
 - ▶ Program evolution becomes much more manageable!

Encapsulation

- When you use `string` objects, you did not know their data members.
 - ▶ Encapsulation means that they are hidden from you.
 - ▶ But you were allowed to call member functions such as `substr`, `length`, and so on.
 - ▶ You were given an interface to the object.
 - ▶ All those member functions and operators are the interface to the object.



Thinking in OOP???

- ① Real-world objects share two characteristics: they all have state and behavior. Objects in C++ are conceptually similar to real-world objects: they also consist of state and behavior.
- ② The process of providing a public interface, while hiding the implementation details, is called encapsulation.
- ③ A class is the blueprint from which individual objects are created, and it describes a set of objects with the same behavior.
- ④ Hiding internal data from the outside world, and accessing it only through publicly exposed functions is known as data encapsulation.
- ⑤ A software object's state is stored in data members.
- ⑥ A software object's behavior is exposed through Member functions.

Classes

- In C++, a programmer doesn't implement a single object. Instead, the programmer implements a class.
- A class describes a set of objects with the same behavior.
 - ▶ You would create the `car` class to represent cars as objects.
- To define a class, you must:
 - ▶ Define the **data members** to hold the object's attributes.
 - ▶ Implement the **member functions** to specify the behavior.

Designing the class

- We will design a **Clock** class.
- By observing a real clock working, we realize our clock design needs member functions to do the following:
 - ▶ Set time to hh:mm:ss.
 - ▶ Increase time by seconds, minutes or hours.
 - ▶ compare two time representations of clock.

These activities will be our public interface.
- The public interface is specified by declarations in the class definition. The data members are defined there also.

Designing the class

- To define a class, you write:

```
class NameOfClass
{
    public:
        // the public interface
    private:
        // the data members
}; ← Don't miss the semi-colon!
```

- ▶ Any part of the program should be able to call the member functions – so they are in the public section.
- ▶ Data members are defined in the private section of the class. Only member functions of the class can access them. They are hidden from the rest of the program.

Designing the class

- Here is an example of the `Clock` class definition:

```
class Clock
{
    public:
        void setClock(int hh, int mm, int ss);
        void incrementSeconds(int ss);
        void incrementMinutes(int mm);
        void incrementHours(int hh);
        void printTime() const;
        int compareTime(Clock C) const;
    private:
        // data members will go here
};
```

- ▶ The public interface has the six activities that we decided this class should support.
- ▶ Notice that these function prototypes are declarations. They will be defined (implemented) later.

Functions

- There are two kinds of member functions:

- ▶ **Mutators (setters):** a function that modifies the data members of the object.

- ▶ **Accessors (getters):** a function that queries a data member of the object. It returns the value of a data member to its caller.

```
class Clock {
```

```
public:
```

```
void setClock(int hh, int mm, int ss);
```

```
void incrementSeconds(int ss);
```

```
void incrementMinutes(int mm);
```

```
void incrementHours(int hh);
```

Mutators

```
void printTime() const;
```

```
int compareTime(Clock C) const;
```

Accessors

```
private:
```

```
// data members will go here
```

```
};
```

Mutators

- You call the member functions by first creating a variable of type `clock` and then using *the dot notation*:

- ▶ `Clock clock1;`

...

```
clock1.setClock(12, 30, 47);
```

...

```
clock1.incrementHours(2);
```

An object of Clock class

- ▶ Because these are mutators, the data stored in the class can (and will) be changed.

Accessors

- Because accessors should never change the data in an object, you should make them **const**.

```
class Clock
{
public:
    void setClock(int hh, int mm, int ss);
    void incrementSeconds(int ss);
    void incrementMinutes(int mm);
    void incrementHours(int hh);
    void printTime() const;
    int compareTime(Clock C) const;
private:
    // data members will go here
};
```

This statement will print the current time:

```
clock1.printTime();
```

Encapsulation

- Each Clock object must store the hours, minutes and seconds to represent the current time. Assume that we use the 24-hour clock.

```
class Clock
{
public:
    // interface
private:
    // Declare data members of the class
    // 0<= hours<24; 0<= minutes or seconds < 60
    int hours, minutes, seconds;
};
```

- Every Clock object has a separate copy of these data members.

```
Clock clock1;
Clock clock2;
```

Encapsulation

- Because the data members are private, this won't compile:

```
int main()
{
    ...
    clock1.hours = clock1.hours + 1;
    // Error—use incrementHours(1) instead
    ...
}
```

- ▶ A good design principle: **Never have any public data members.**
- ▶ Then your data is encapsulated by your class.
- ▶ One benefit of the encapsulation mechanism is that we can make guarantees.

Encapsulation and functions as guarantees

- We can write the mutator for **hours** so that **hours** cannot be set to a negative value or a number larger than 24
 - ▶ If **hours** were public, it could be directly set to an invalid value by some misguided programmer.
- There is a second benefit of encapsulation that is particularly important in larger programs:
 - ▶ Implementation details often need to change over time.
 - ▶ You want to be able to make your classes more efficient or more capable, without affecting the programmers that use your classes.
 - ▶ As long as those programmers do not depend on the implementation details, you are free to change them at any time.

The interface

- The interface should not change even if the details of how they are implemented change.



- A driver switching to an electric car does not need to relearn how to drive.

Implementing the member functions

- To specify that a function is a member function of your class you must write `clock::` in front of the member function's name:

```
class Clock
{
    public:
        ...
    private:
        ...
};
```

You should NOT write `clock::` in the class declaration

Only specify that when you implement the functions outside the class declaration.

```
void Clock::setClock(int hh, int mm, int ss)
{
    hours = hh;
    minutes = mm;
    seconds = ss;
}
```

Coding with the instructor...

A quick start of Lab 7

- Download **Clock.h** from the following link:
<https://www.cs.uky.edu/~yipike/CS215/Clock.h>
- Separate class declaration in header file and implementation in .cpp file
- Complete the definition for the class named **Clock**
- Complete the main function according to **Lab7** description

Implicit parameters

- If we have two **Clock** objects, which clock time is **setClock** working on?

▶ **Clock clock1;** implicit parameters

...

clock1.setClock(12, 30, 47);

- ▶ The variable to the left of the **dot operator** is implicitly passed to the member function.

▶ **void Clock::setClock(int hh, int mm, int ss)**
{
 implicit parameter.hours = hh;
 implicit parameter.minutes = mm;
 implicit parameter.seconds = ss;
}

(Note the grey part is never shown in your code, it is for understanding only)

Calling a member function from a member function

- We have already written the `setClock` member function, however, this function does not guarantee that data members are in the correct range, and we want to call another member function to adjust time:

```
void Clock::setClock(int hh; int mm; int ss)
{
    hours = hh;
    minutes = mm;
    seconds = ss;
    adjustClock();
}
```

- ▶ When one member function calls another member function on the same object, do NOT use the dot notation.

implicit parameter.`adjustClock()` ;

Designing the class

- Let us complete the declaration of the `Clock` class:

```
class Clock
{
public:
    void setClock(int hh, int mm, int ss);
    void incrementSeconds(int ss);
    void incrementMinutes(int mm);
    void incrementHours(int hh);
    void printTime() const;
    int compareTime(Clock C) const;
private:
    int hours, minutes, seconds;
    void adjustClock();
};
```

- ▶ Notice that `adjustClock()` member function serves as a service function to make sure after calling any mutator function, the data members are always in the correct range. So we can put it in the private part.

```

void Clock::adjustClock()
{
    int adjust = 0;
    if (seconds >= 60)
    {
        adjust = seconds / 60;
        seconds = seconds % 60;
        minutes = minutes + adjust;
    }
    if (minutes >= 60)
    {
        adjust = minutes / 60;
        minutes = minutes % 60;
        hours = hours + adjust;
    }
    if (hours >= 24)
        hours = hours % 24;
}

```

Constructors

- A constructor is a member function that initializes the data members of an object.

- ▶ **The constructor is automatically called whenever an object is created.**

`Clock clock1;`

- ▶ By supplying a constructor, you can ensure that all data members are properly set before any member functions act on an object.
- What would be the value of a data member that was not properly set?
 - ▶ The garbage value!

Constructors

- Consider the following statements:

```
▶ Clock clock1;  
  clock1.incrementHours(2);  
  clock1.printTime(); // Unpredictable values
```

▶ The programmer forgot to call **setClock()** before increasing time by hours.

▶ Constructors are written to guarantee that an object is always fully and correctly initialized when it is defined.

- Constructors are declared in the class definition:

```
class Clock  
{  
  public:  
    Clock(); // A constructor  
    ... no return type, not even void  
};
```

▶ The name of a constructor is **identical to** the name of its class.

Defining the constructors

- Once you declared the constructors in class definition, you have to define (implement) it.

```
Clock::Clock()  
{  
  hours = 0;  
  minutes = 0;  
  seconds = 0;  
}
```

▶ To connect the definition with the class, you must use the same **::** notation. You should choose initial values for the data members so the object is correct. And remember, there is NO return type!

- A constructor with no parameters is called a **default constructor**.
 - Default constructors are called when you define an object and do not specify any parameters for the construction.
 - ▶ `Clock clock1;`

Constructors

- Constructors can have parameters, and constructors can be overloaded:

```
class Clock
{
public:
    // Sets time to 00:00:00
    Clock();
    // Sets time to hh:mm:ss
    Clock(int hh, int mm, int ss);
    // Member functions omitted
private:
    int hours, minutes, seconds;
};
```

Constructors

- When you construct an object, the compiler chooses the constructor that matches the parameters that you supply:

```
▶// Uses default constructor
```

```
    Clock joes_clock;
```

```
    // Uses Clock(int,int,int) constructor
```

```
    Clock lisas_clock(12, 30, 47);
```

- A common error is trying to use the constructor to reset

```
▶ Clock clock1;
```

```
...
```

```
    Clock1.Clock(); // Error, use setClock() instead
```


Initialization lists

- C++ provides another way of initializing member variables that allows us to initialize member variables when they are created rather than afterwards.
- This is done through use of an initialization list.

```
class Clock
```

```
{
```

```
public:
```

```
    Clock();
```

```
private:
```

```
    int hours;
```

```
    int minutes;
```

```
    int second;
```

```
};
```

Initializing the data members by explicitly implementing the constructor.

```
Clock::Clock()
```

```
{
```

```
    hours = 0;
```

```
    minutes = 0;
```

```
    seconds = 0;
```

```
}
```

Initializing the data members via initialization list.

```
Clock():hours(0), minutes(0), seconds(0){};
```

const-Correctness

- You should declare all accessor functions in C++ with the `const` reserved word.

```
class Clock
```

```
{
```

```
public:
```

```
    void printTime() const;
```

```
    int compareTime(Clock C) const;
```

```
    int getHours() const;
```

```
    int getMinutes() const;
```

```
    int getSeconds() const;
```

```
private:
```

```
    // data members will go here
```

```
};
```

]


Add more getters functions if you need

► After the parameters (both *declaration* and *implementation*)

► Your `const` function should not change any data members.

► It cannot call any non-`const` functions within its function body either!


Which of the following is a benefit of encapsulation?

- A. It allows a function defined in one class to be reused in another class;
-  B. It guarantees that an object cannot be accidentally put into an inconsistent state;
- C. It improves the performance of an application;
- D. It prevents functions that are internal to a class from changing private data members;

Consider the member function call

```
reg.add_item(2, 19.95);
```

Which of the following describes the role of `reg` in this call?

- A. It is the default parameter;
- B. It is an explicit parameter;
-  C. It is an implicit parameter;
- D. It is a constructor parameter;