Data Structures and Algorithms

Dr. Salwa Osama

Chapter 1

Introduction to Abstract Data Types Revision

Course Content

Week1	Chapter 1. Introduction and revision (Recursion, Pointers, Arrays, Classes, Abstract Data Types)
Week2	Chapter 2. Lists (Array based, Dynamic Allocation, Linked List)
Week3	Chapter 3. Stacks
Week4	Chapter 4. Queues
Week5	Chapter 5. Templates (ADT Implementation, Standard Template Library)
Week6	Revision
Week7	Midterm
Week8	Chapter 6. Trees (and binary trees)
Week9	Chapter 7.Algorithm Efficiency
Week10	Chapter 8. Searching Algorithms
Week11	Chapter 9. Sorting Algorithms
Week12	Chapter 10.Graphs & digraphs
Week13	Revision
Week14	Practicle

Grading Criteria

- Midterm → 30
- Final → 50
- Practical Exam →10
- Quizzes → 5
- Assignments →5

Contents

- A first look at ADTs and Implementations
- C++'s Simple Data Types
- Programmer-Defined Data Types
- Arrays
- Pointers
- Recursion
- Classes

Objectives

- Distinguish between ADTs and implementations of ADTs
- Review C++'s simple data types & ADTs they model
- Look at simple mechanisms to define new data types
- Review Arrays
- Take a review at pointers and pointer operations
- Review recursion
- Review classes

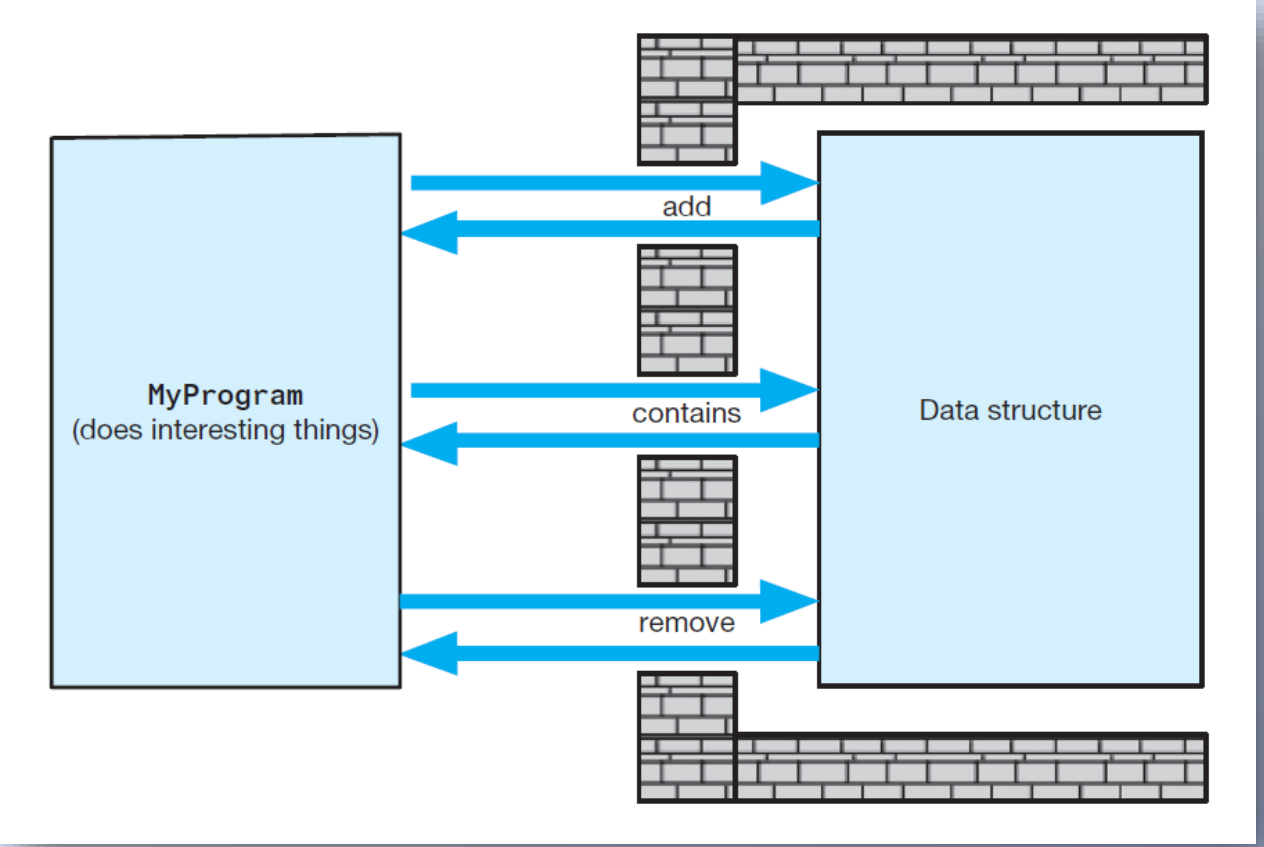
First Look at ADTs & Implementations

- For a programming task we must identify
  - The collection of data items
  - Basic operations to be performed on them
- Taken together (data items & operations)
  - are called an **A**bstract **D**ata **T**ype (ADT)
- Implementation
  - Storage structures for the data items
  - Algorithms for the operations

Abstract **D**ata **T**ypes (ADT)

- Typical operations on data
  - Add data to a data collection.
  - Remove data from a data collection.
  - Ask questions about the data in a data collection.
- An ADT : a collection of data *and* a set of operations on data
- A data structure: an implementation of an ADT within a programming language

Abstract **D**ata **T**ypes (ADT)



- A wall of ADT operations isolates a data structure from the program that uses it

C++ Simple Data Types

Integers

- Unsigned integers (The set of *nonnegative* integers)
  - `unsigned short, unsigned, unsigned long`
  - Sometimes called *whole numbers or cardinal numbers*
  - Represented in 2, 4, or 8 bytes
  - For example, 58 can be represented as a 16-bit binary numeral

$$58 = 0000000000111010_2$$

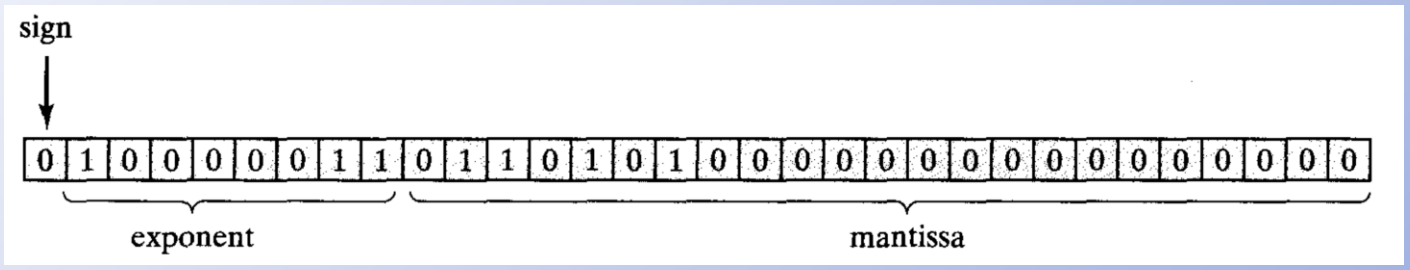
and stored in two bytes:

- Signed integers
  - `short, int, long`
  - represented in two's complement

C++ Simple Data Types

Real Data

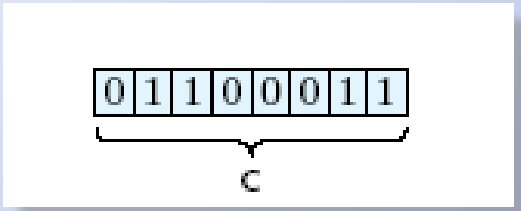
- Types *float* and *double* in C++, extended-precision type is *long double*.
- Use single precision (IEEE Floating-Point)
- Store:



  - sign of ***mantissa*** in leftmost bit (0 = +, 1 = –)
  - represent exponent in next 8 bits (exponent + 127)
  - bits $b_2b_3 \ldots b_{24}$ ***mantissa*** in rightmost 23 bits.

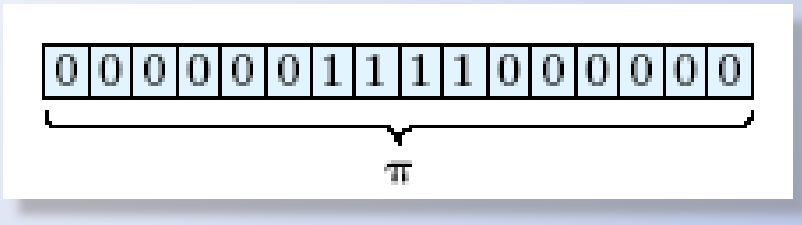
C++ Simple Data Types

Character Data



ASCII code for C 99

- 1 byte for ASCII, EBCDIC
- 2 bytes for Unicode (java, Python) or C++ *wide character type*



Unicode code for π 960

- Operations ==, <, >, etc. Using numeric code

C++ Simple Data Types
Boolean Data

- Values { false, true }
- Could be stored in bits, usually use a byte
- Operations &&, ||
- In C++
  - bool type
  - int (boolVal) evaluates to
    - ✓0 if false
    - ✓1 if true Otherwise

Programmer-Defined Data Types
typedef

- Typedefs
  - Mechanism usable to create a new type
  - Give new name to existing type
- Example:

```
typedef int counter;
```

  - Now either `double` or `real` can be used.

Programmer-Defined Data Types
enum

- Enumerations
  - Mechanism for creating types whose literals are identifiers
  - Each identifier associated with unique integer

```
enum Color{RED, ORANGE, YELLOW, GREEN, BLUE,
  INDIGO, VIOLET};
```


```
Color Shade = BLUE;
cout<<Shade<<endl;          // Output 4
```

Programmer-Defined Data Types

enum

- Also, possible to specify explicit values to give the enumerators

```
enum NumberBase { BINARY = 2,
                  OCTAL = 8,
                  DECIMAL = 10,
                  HEXADECIMAL = 16};
```

Arrays

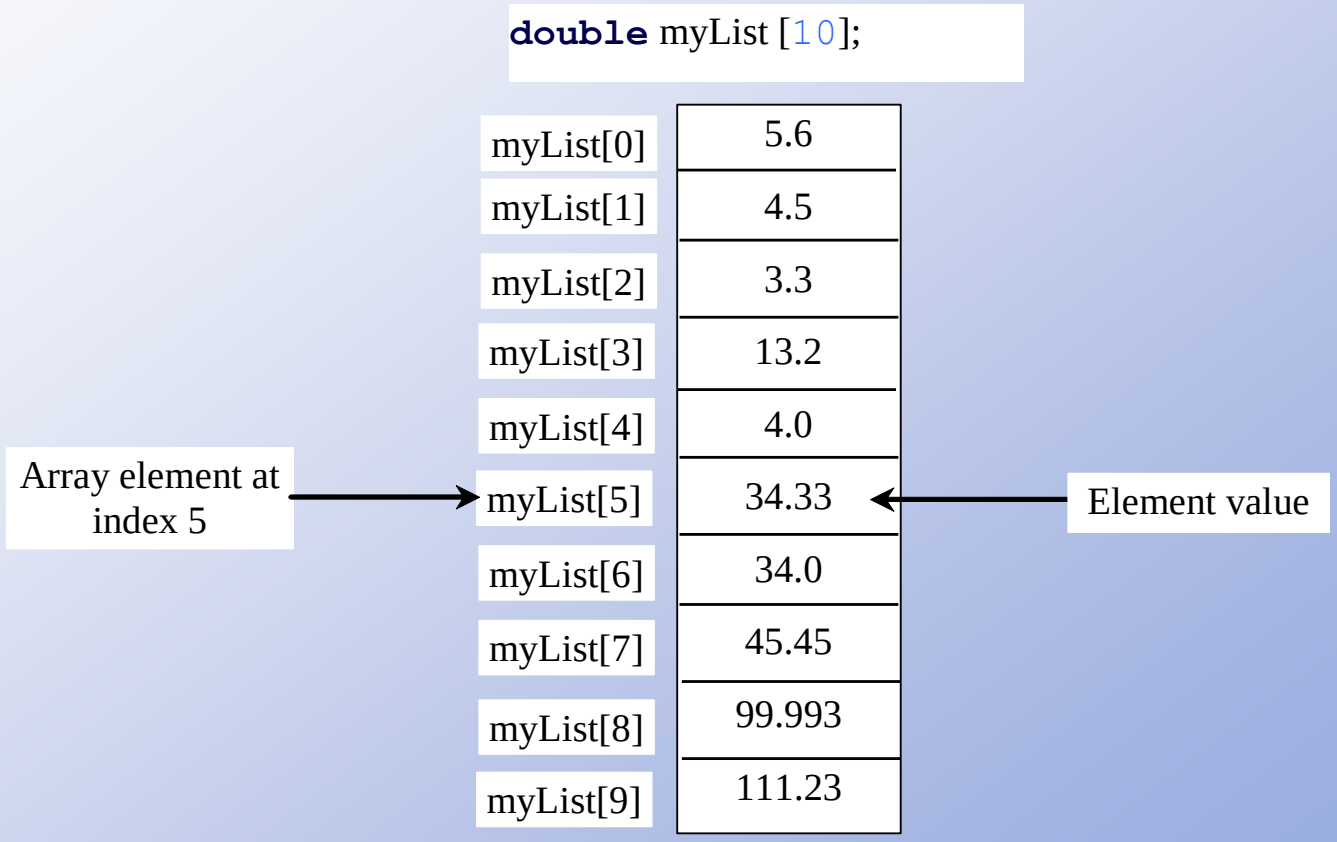
- Array: a collection of a fixed number of components wherein all the components have the same data type
- In a one-dimensional array, the components are arranged in a list form
- Syntax for declaring a one-dimensional array:

```
dataType arrayName[intExp];
```

`intExp` evaluates to a positive integer call Index

Introducing Arrays

Array is a data structure that represents a collection of the same types of data.



Array Initializers

Declaring, creating, initializing in one step:

dataType arrayName[arraySize] = {value0, value1, ..., value*k*};

double myList[4] = {1.9, 2.9, 3.4, 3.5};

Initializing arrays with random values

The following loop initializes the array myList with random values between 0 and 99:

```
for (int i = 0; i < ARRAY_SIZE; i++) {
    myList[i] = rand() % 100;
}
```

Finding the Largest Element

- Use a variable named *max* to store the largest element. Initially *max* is *myList[0]*.
- To find the largest element in the array *myList*, compare each element in *myList* with *max*, update *max* if the element is greater than *max*.

```
double max = myList[0];
for (int i = 1; i < ARRAY_SIZE; i++) {
    if (myList[i] > max)
        max = myList[i];
}
```

Finding the smallest index of the largest element

```
double max = myList[0];
int indexOfMax = 0;
for (int i = 1; i < ARRAY_SIZE; i++) {
  if (myList[i] > max) {
    max = myList[i];
    indexOfMax = i;
  }
}
```

What About if I need Largest index of the largest element?

Passing Arrays to Functions

- Just as you can pass single values to a function, you can also pass an entire array to a function with its size.
- Arrays are passed by reference only.
- The symbol & is not used when declaring an array as a formal parameter.
- Array can't be returned from a function.

```
#include <iostream>
using namespace std;

void printArray(int list[], int arraySize); // Function prototype

int main()
{
  int numbers[6] = {1, 4, 3, 6, 8, 9};
  printArray(numbers, 6); // Invoke the function

  return 0;
}

void printArray(int list[], int arraySize)
{
  for (int i = 0; i < arraySize; i++)
  {
    cout << list[i] <<  " ";
  }
}
```

Dynamic Arrays

- Array that created during the execution of a program (during run time)
- Example:

```
int *p;
p = new int[10];
```

- C++ allows us to use array notation to access these memory locations
- The statements:

```
p[0] = 25;
p[1] = 35;
```

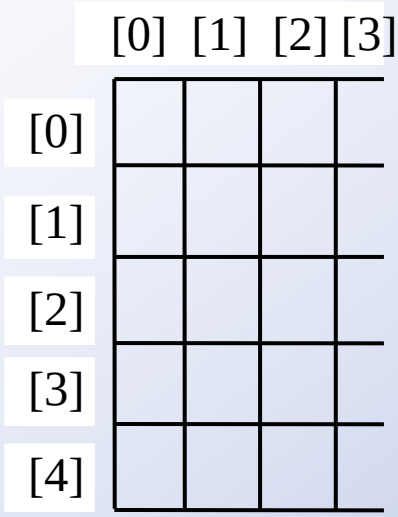
store `25` and `35` into the first and second array components, respectively

Two-dimensional Arrays

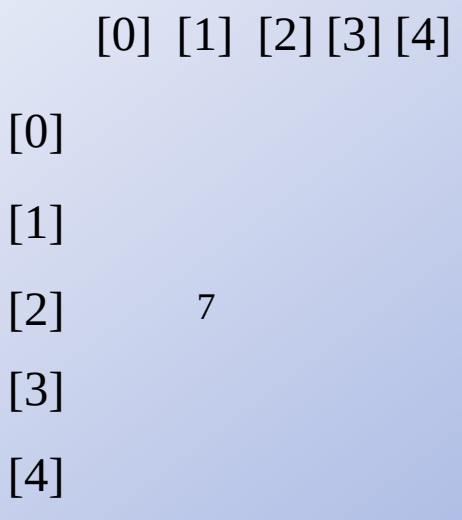
```
// Declare array ref var
```

elementType arrayName[rowSize][columnSize];

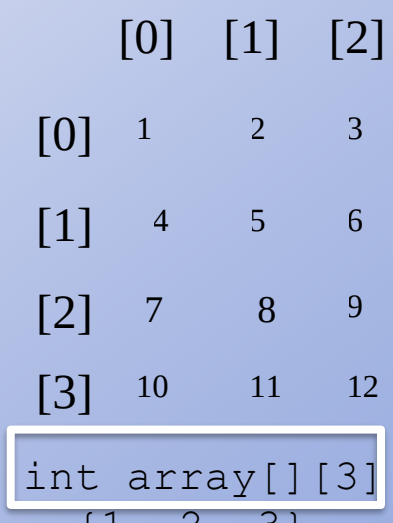
int matrix[5][5];




```
int matrix[5][5];
```




```
matrix[2][1] = 7;
```




```
int array[][3] ={
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```

Initializing Arrays with Random Values

- The following loop initializes the array with random values between 0 and 99:

```
for (int row = 0; row < rowSize; row++)
{
  for (int column = 0; column < columnSize;
    column++)
  {
    matrix[row][column] = rand() % 100;
  }
}
```

Summing All Elements

- To sum a two-dimensional array, you must sum each element in the array using a loop like the following:

```
int Sum=0;
for (int row = 0; row < rowSize; row++) {
  for (int column = 0; column < columnSize; column++) {
    Sum+=matrix[row][column] ;
  }
}
```

Summing Elements by Column

- For each column, use a variable named *total* to store its sum. Add each element in the column to *total* using a loop like this:

```
for (int column = 0; column < columnSize;
    column++)
{
  int total = 0;
  for (int row = 0; row < rowSize; row++)
    total += matrix[row][column];
  cout << "Sum for column " << column << " is " <<
    total << endl;
}
```

What About if I need to sum all elements by row?

Passing Two-Dimensional Arrays as Parameters to Functions

- **Function Prototype:**

  void PrintArray (int a[ ] [3]);

- **Function Calling:**

  Int x[5] [3];

  PrintArray (x);

- **Function definition:**

  void PrintArray (int m[ ] [3]){

  }

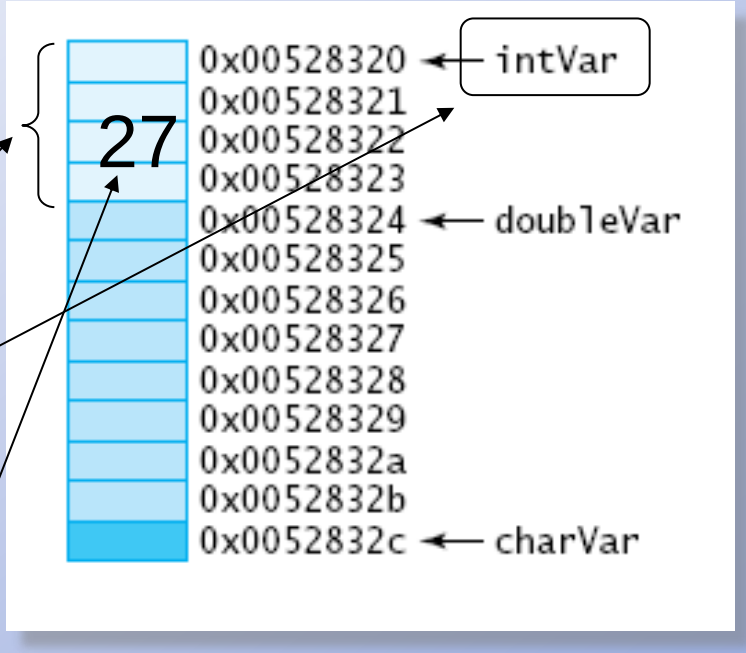
Passing Two-Dimensional Arrays to Functions

```
#include <iostream>
using namespace std;

const int COLUMN_SIZE = 4;

int sum(const int a[][COLUMN_SIZE], int rowSize)
{
  int total = 0;
  for (int row = 0; row < rowSize; row++)
  {
    for (int column = 0; column < COLUMN_SIZE; column++)
    {
      total += a[row][column];
    }
  }

  return total;
}

int main()
{
  const int ROW_SIZE = 3;
  int m[ROW_SIZE][COLUMN_SIZE];
  cout << "Enter " << ROW_SIZE << " rows and "
    << COLUMN_SIZE << " columns: " << endl;
  for (int i = 0; i < ROW_SIZE; i++)
    for (int j = 0; j < COLUMN_SIZE; j++)
      cin >> m[i][j];

  cout << "\nSum of all elements is " << sum(m, ROW_SIZE) << endl;

  return 0;
}
```

- You can pass a two-dimensional array to a function; however, C++ requires that the column size to be specified in the function declaration.
- Here is an example with a function that returns the sum of all elements in a matrix.

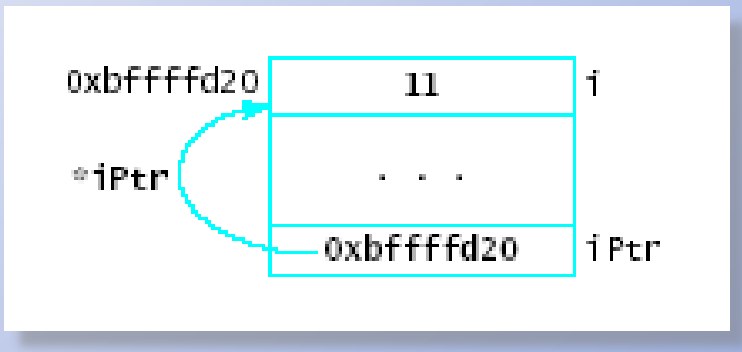
Pointers

- When regular variables are declared
- int intVar=27;
  - Memory allocated for value of specified type
  - Variable name associated with that memory location
  - Memory initialized with values provided (if any)



Basic Pointer Operations

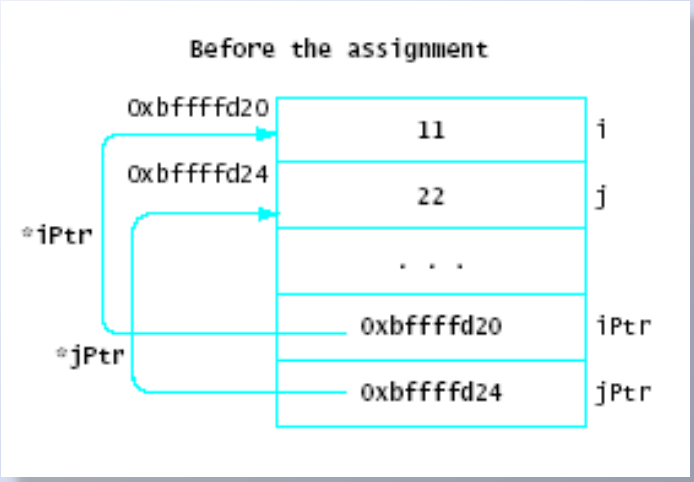
- Dereferencing and indirection
  - Pointer variable stores address of a location
  - Accessing contents of that location requires dereferencing operator *

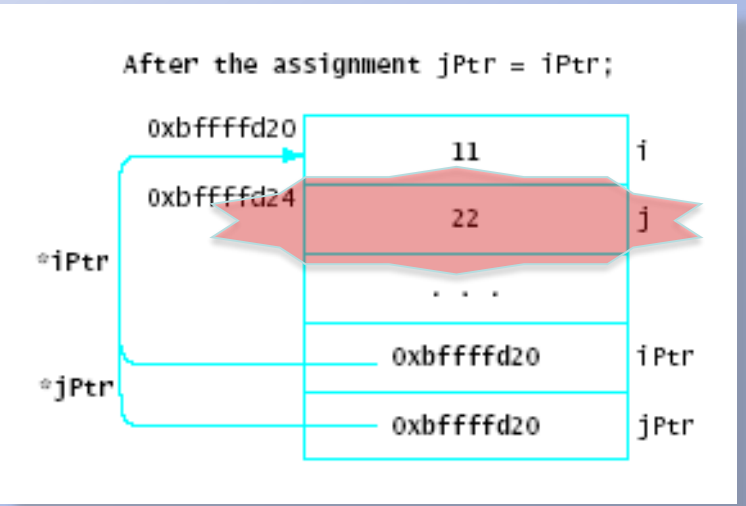


Basic Pointer Operations

- Assignment
  - Pointer variables can be assigned the values of other pointer variables bound to same type

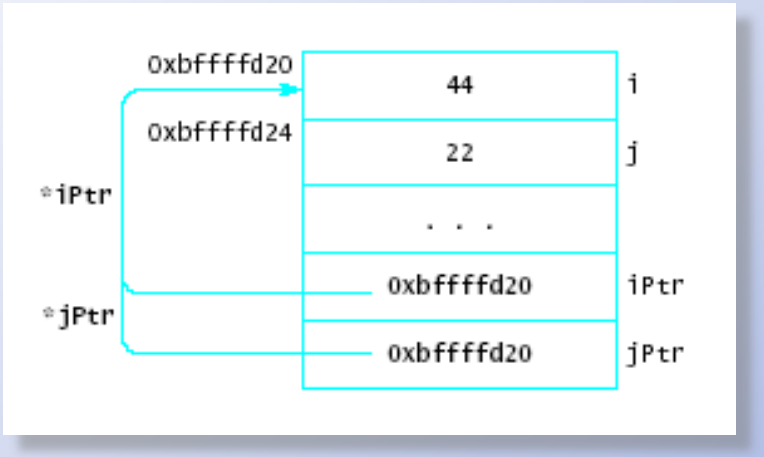
jPtr=iPtr;





Basic Pointer Operations

- Consider `*jPtr = 44;`
  - Changes value that both pointers reference



Aliasing in programming refers to the situation where **two or more variables refer to the same memory location**.

  - Not good programming practice, hard to debug
  - Known as *aliasing problem*

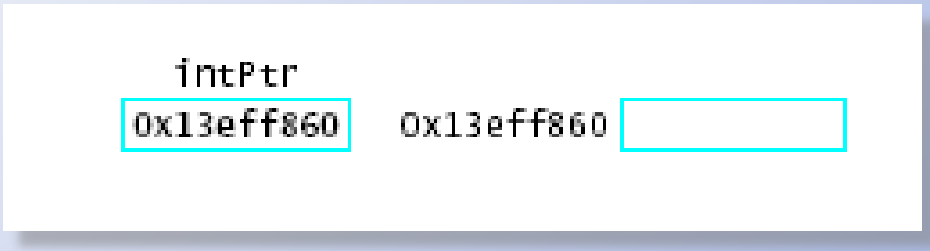
Basic Pointer Operations

- Comparison
  - Relational operators used to compare two pointers
  - Must be bound to same type
  - Most common == and !=
  - The null (0) address may be compared with any pointer variable

Dynamic Memory Allocation

- The ***new*** operation
- Example

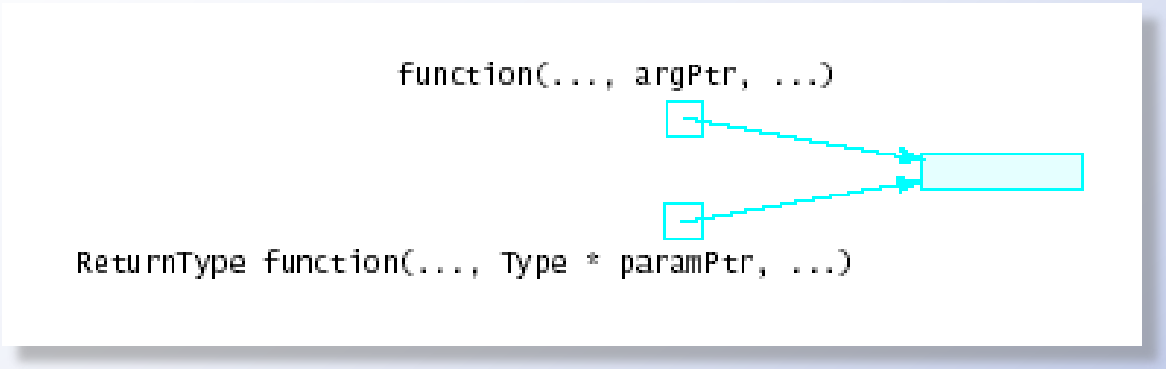
```
int * intPtr;
intPtr = new int;
```



  - An anonymous variable
  - Cannot be accessed directly

Pointer Arguments

- Pointers can be passed as arguments to functions



- This is logically equivalent to reference parameters
  - In fact, this is how early C++ compilers accomplished reference parameters

```
void swap(int * first, int * second)
{
  int temp = *first;
  *first = *second;
  *second = temp;
}
```

and the preceding function call to

```
swap(&x, &y);
```

Recursive Definitions

- Recursion: solving a problem by reducing it to smaller versions of itself
- Provides a powerful way to solve certain problems which would be complicated otherwise

Recursive Definitions (cont'd.)

- Base case: the case for which the solution is obtained directly
  - Every recursive definition must have one (or more) base case(s)
  - The base case stops the recursion
- General case: must eventually reduce to a base case

Recursive Definitions (cont'd.)

- Example: factorials

$$1! = 1 \qquad (1)$$

$$n! = n \times (n\text{-}1)! \quad \text{if} \quad n > 0 \qquad (2)$$

  - Equation (1) is called the *base case*
  - Equation (2) is called the *general case*

Programming Example

- To find the binary representation of 35
  - Divide 35 by 2
  - The quotient is 17 and the remainder is 1
  - Divide 17 by 2
  - The quotient is 8 and the remainder is 1
  - Divide 8 by 2
  - The quotient is 4 and the remainder is 0
  - Continue this process until the quotient becomes 0

Programming Example

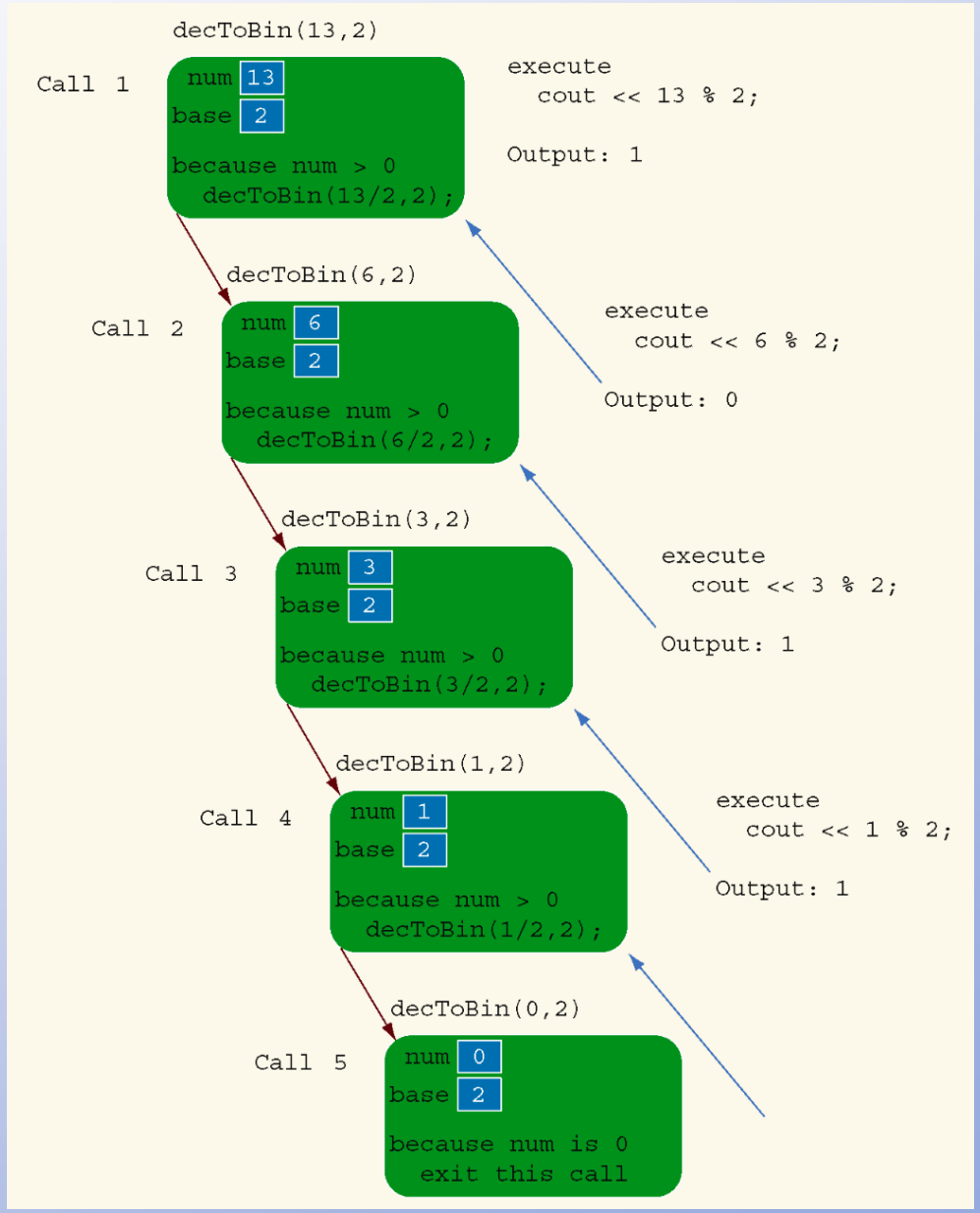
1. $binary(num) = num \rightarrow if\ num = 0$
2. $binary(num) = binary(num/2) \rightarrow followed\ by\ (num\%2) \rightarrow if\ num > 0$

```
11  void dec2Bin(int num){
12      if(num>0){
13          dec2Bin(num/2);
14          cout<<num%2;
15      }
16  }
```

Programming Example

$decToBin(13);$

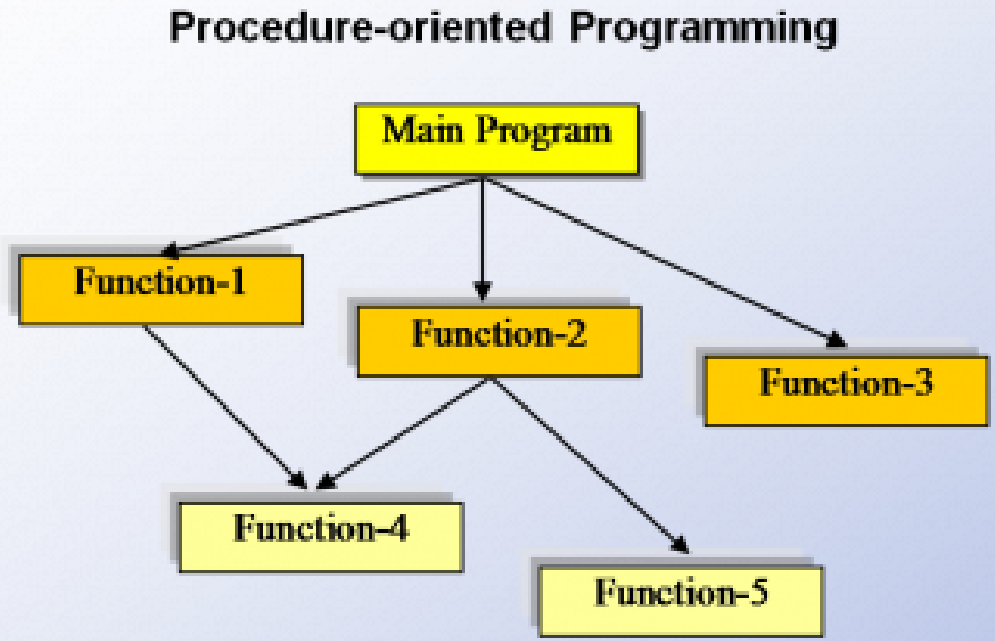
```
11  void dec2Bin(int num){
12      if(num>0){
13          dec2Bin(num/2);
14          cout<<num%2;
15      }
16  }
```

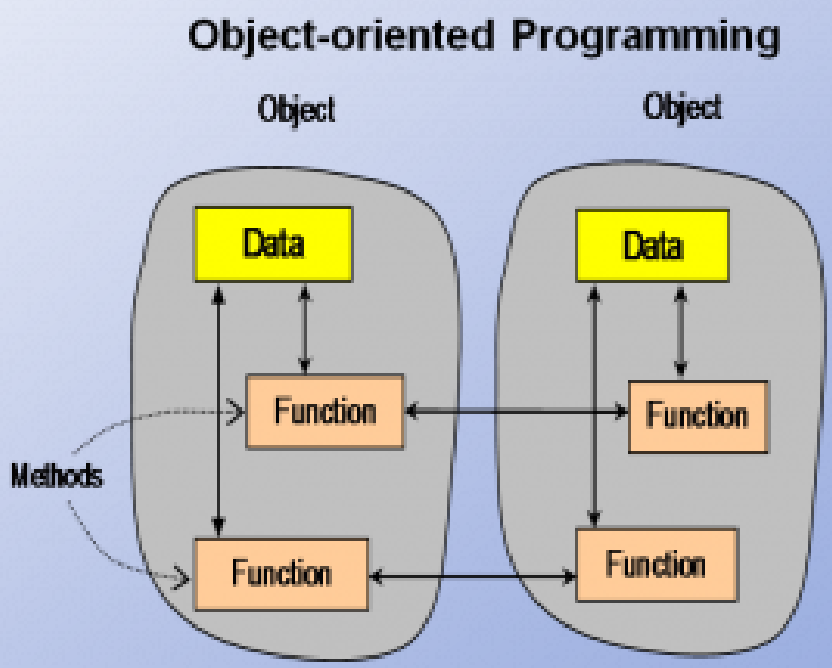


Infinite Recursion

- Infinite recursion: every recursive call results in another recursive call
  - In theory, infinite recursion executes forever
- Because computer memory is finite:
  - Function executes until the system runs out of memory (stack overflow)
  - Results in an abnormal program termination

Object Oriented Programming





Classes

- A class is the building block, that leads to Object Oriented programming. It is a user-defined data type.
- holds its own data members and member functions, which can be accessed and used by creating an instance of that class.
- For Example:
  - Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them have 4 wheels, Speed Limit, Mileage range etc.
  - So here, Car is the class and wheels, speed limits, mileage are their properties (member data).
  - A car can move, stop, move back which are its operations (member function)

Classes (continued)

- Class member can be a variable or a function
- If a member of a `class` is a variable
  - It is declared like any other variable
- In the definition of the `class`
  - You cannot initialize a variable when you declare it
- If a member of a class is a function
  - Function prototype is listed
  - Function members can (directly) access any member of the `class`

Class member Access Modifiers

- **`private`** (default)
  - Member cannot be accessed outside the `class`
- **`Public`**
  - Member is accessible outside the class
- **`Protected`**
  - member can be accessed within the class and from the derived class (subclass).

Classes Example

Suppose that we want to define a class to implement the time of day in a program. Because a clock gives the time of day, let us call this `class clockType`. Furthermore, to represent time in computer memory, we use three `int` variables: one to represent the hours, one to represent the minutes, and one to represent the seconds.

Suppose these three variables are:

```
int hr;
int min;
int sec;
```

We also want to perform the following operations on the time:

1. Set the time.
2. Retrieve the time.
3. Print the time.
4. Increment the time by one second.
5. Increment the time by one minute.
6. Increment the time by one hour.
7. Compare the two times for equality.

Classes Example

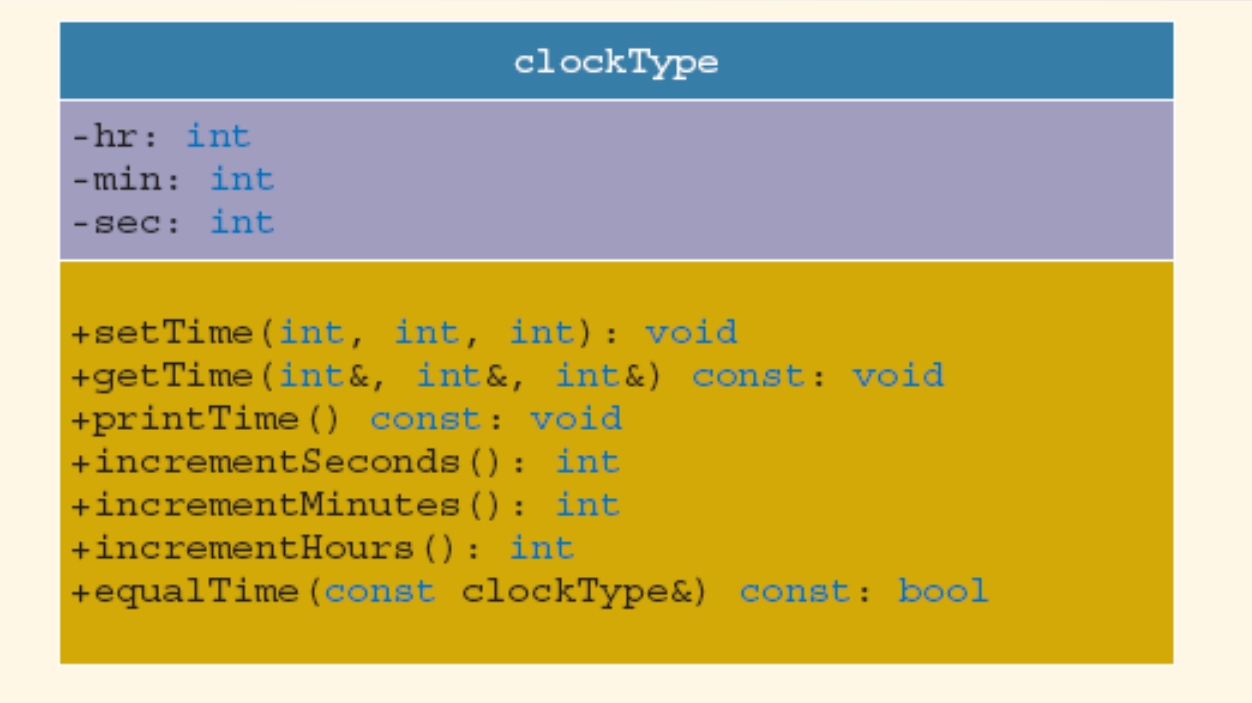
```
class clockType
{
public:
    void setTime(int, int, int);
    void getTime(int&, int&, int&) const;
    void printTime() const;
    void incrementSeconds();
    void incrementMinutes();
    void incrementHours();
    bool equalTime(const clockType&) const;

private:
    int hr;
    int min;
    int sec;
};
```

These functions cannot modify the member variables of a variable of type `clockType`

`const`: formal parameter can't modify the value of the actual parameter

`private` members, can't be accessed from outside the class

Unified Modeling Language Class Diagrams



+: member is `public`

-: member is `private`

#: member is `protected`

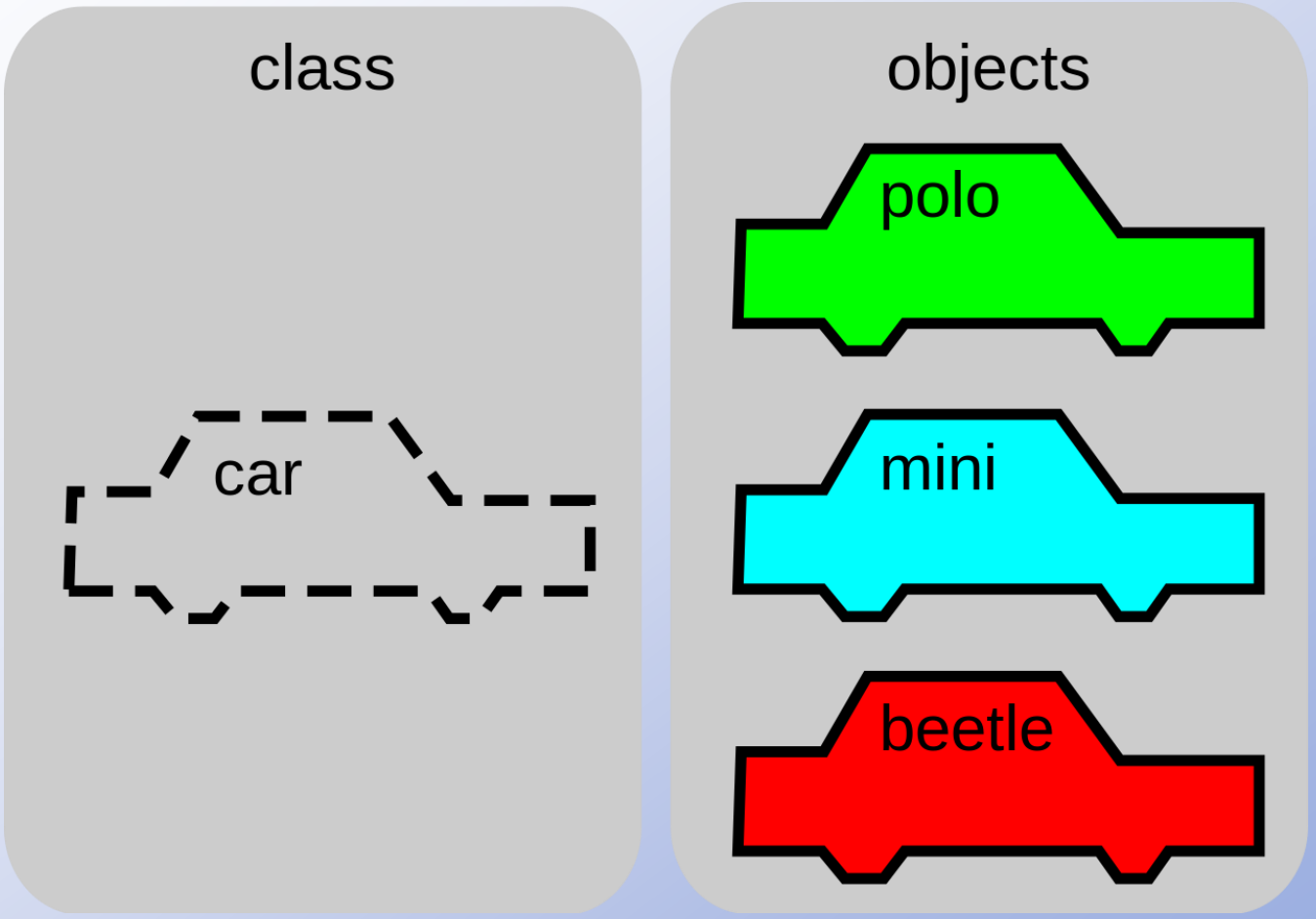
Variable (Object) Declaration

- Once a `class` is defined, you can declare variables of that type

```
clockType    myClock;
clockType    yourClock;
```

- A `class` variable is called a `class` object or `class` instance

Class versus Object



Accessing Class Members

- Once an object is declared, it can access the `public` members of the class
- Syntax:

```
classObjectName.memberName
```

  - The dot (`.`) is the **member access operator**
- If object is declared in the definition of a member function of the `class`, it can access the `public` and `private` members

Built-in Operations on Classes

- Most of C++'s built-in operations do not apply to classes
  - Arithmetic operators cannot be used on `class` objects unless the operators are overloaded
  - You cannot use relational operators to compare two `class` objects for equality
- Built-in operations valid for `class` objects:
  - Member access (`.`)
  - Assignment (=)

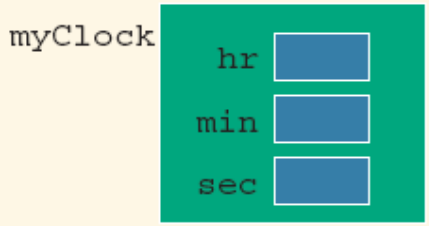
Functions and Classes

- Objects can be passed as parameters to functions and returned as function values
- As parameters to functions
  - Objects can be passed by value or by reference
- If an object is passed by value
  - Contents of data members of the actual parameter are copied into the corresponding data members of the formal parameter

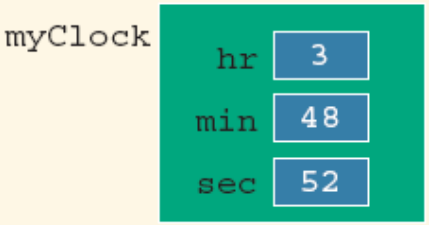
Implementation of Member Functions

```cpp
#include <iostream>
using namespace std;

class Car {
public:
    void show();  // Function declaration inside the class
};

// Function definition outside the class using the scope resolution operator
void Car::show() {
    cout << "This is a car." << endl;
}

int main() {
    Car myCar;
    myCar.show();  // Calling the method
    return 0;
}
```

Implementation of Member Functions (continued)



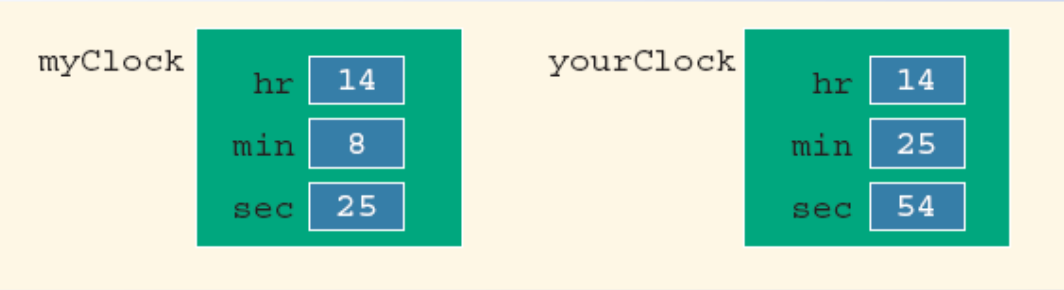

```
myClock.setTime(3, 48, 52);
```



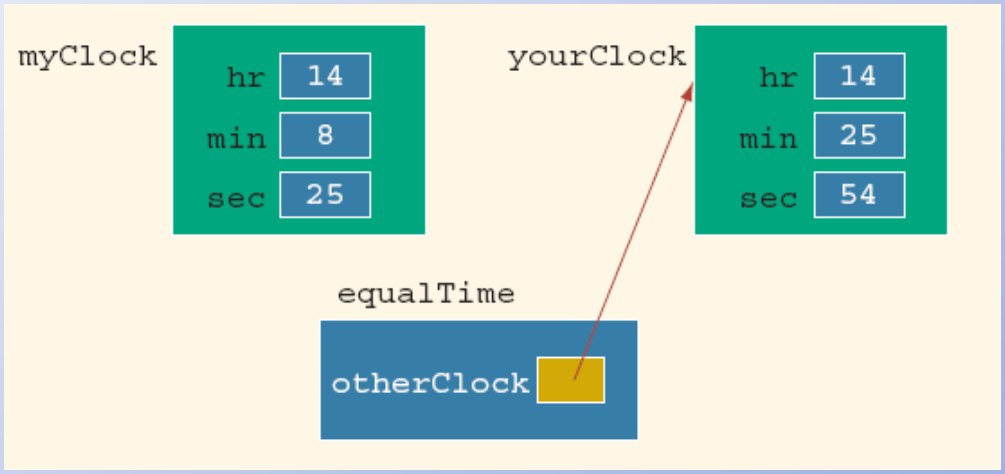
```
void clockType::getTime(int& hours, int& minutes,
                        int& seconds) const
{
    hours = hr;
    minutes = min;
    seconds = sec;
}

void clockType::printTime() const
{
    if (hr < 10)
        cout << "0";
    cout << hr << ":";

    if (min < 10)
        cout << "0";
    cout << min << ":";

    if (sec < 10)
        cout << "0";
    cout << sec;
}
```

```
void clockType::incrementHours()
{
    hr++;
    if (hr > 23)
        hr = 0;
}

void clockType::incrementMinutes()
{
    min++;
    if (min > 59)
    {
        min = 0;
        incrementHours(); //increment hours
    }
}

void clockType::incrementSeconds()
{
    sec++;

    if (sec > 59)
    {
        sec = 0;
        incrementMinutes(); //increment minutes
    }
}
```

```
bool clockType::equalTime(const clockType& otherClock) const
{
    return (hr == otherClock.hr
            && min == otherClock.min
            && sec == otherClock.sec);
}
```




```
if (myClock.equalTime(yourClock))
```



Constructors

- Use constructors to guarantee that data members of a class are initialized
- Two types of constructors:
  - With parameters
  - Without parameters (default constructor)
- The name of a constructor is the same as the name of the class
- A constructor has no type

Constructors (continued)

- A class can have more than one constructor
  - Each must have a different formal parameter
- Constructors execute automatically when a class object enters its scope
  - They cannot be called like other functions
  - Which constructor executes depends on the types of values passed to the class object when the class object is declared

Constructors (continued)

```
class clockType
{
public:
    void setTime(int, int, int);
    void getTime(int&, int&, int&) const;
    void printTime() const;
    void incrementSeconds();
    void incrementMinutes();
    void incrementHours();
    bool equalTime(const clockType&) const;
    clockType(int, int, int);  //constructor with parameters
    clockType();  //default constructor

private:
    int hr;
    int min;
    int sec;
};
```

```
clockType::clockType(int hours, int minutes, int seconds)
{
    if (0 <= hours && hours < 24)
        hr = hours;
    else
        hr = 0;

    if (0 <= minutes && minutes < 60)
        min = minutes;
    else
        min = 0;

    if (0 <= seconds && seconds < 60)
        sec = seconds;
    else
        sec = 0;
}

clockType::clockType()  //default constructor
{
   hr = 0;
   min = 0;
   sec = 0;
}
```

Can be replaced with:

```
setTime(hours, minutes, seconds);
```

Destructors

- Destructors are functions without any type
- The name of a destructor is the Tilt character '~' followed by class name
  - For example:

```
~clockType();
```

- A class can have only one destructor
  - The destructor has no parameters
- The destructor is automatically executed when the class object goes out of scope