Chapter 2

Data Design and Implementation

Outline

- ***Different Views of Data: logical, application and implementation***
  - ***ADT***
- ***C++ built-in types and Class type, OOP review***
- ***Exception Management***
- ***Algorithms comparison***

Data

- Representation of information in a manner suitable for communication or analysis by humans or machines
- Data are nouns of programming world:
  - The objects that are manipulated
  - The information that is processed

Abstract Data Type (ADT)

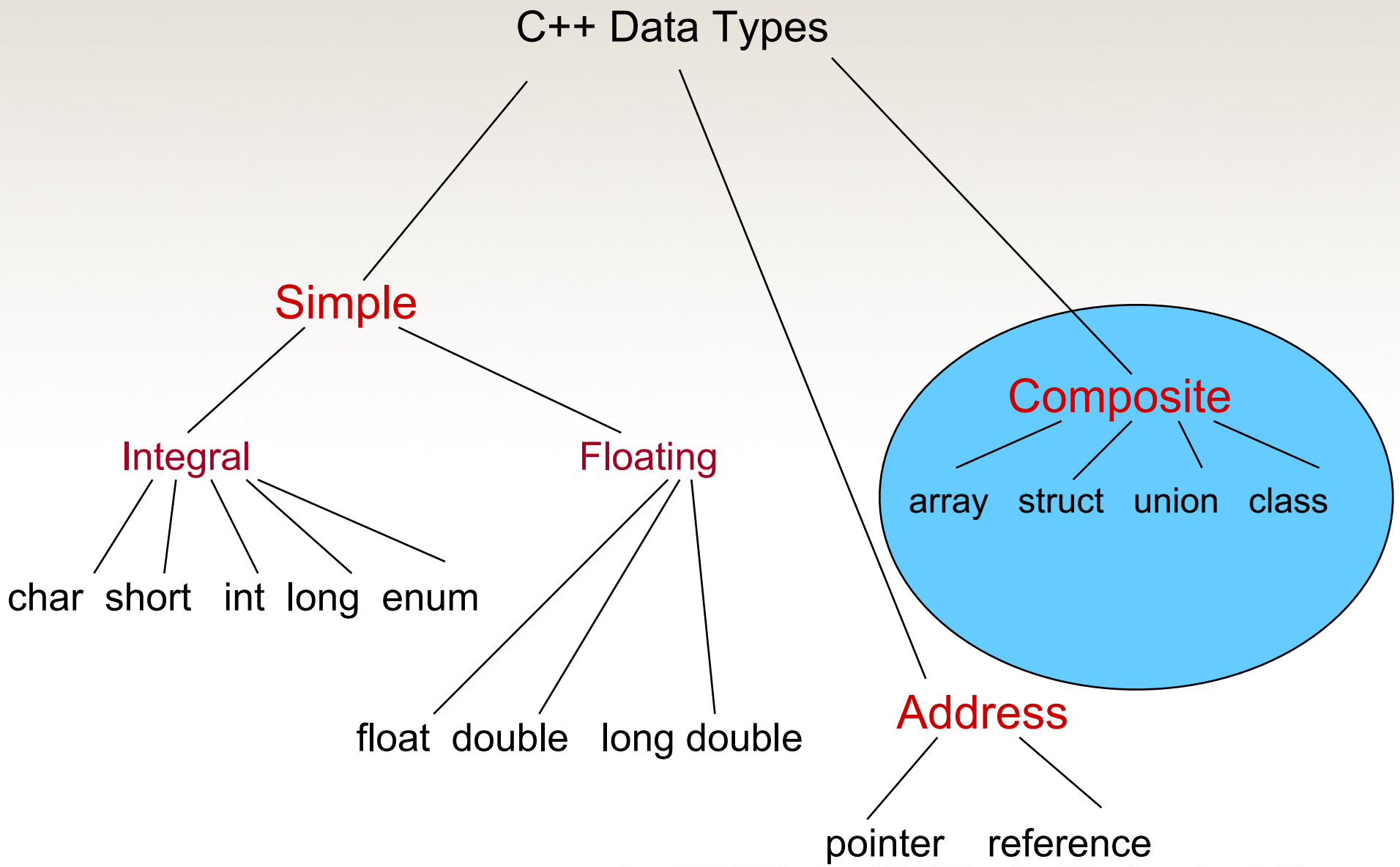
- A data type whose (logical) properties (domain and operations) are specified independently of any particular implementation.
- Different views of ADT:
  - *Application (or user) level:* modeling real-life data in a specific context.
  - *Logical (or ADT) level:* abstract view of the domain and operations. WHAT
  - *Implementation level:* specific representation to hold the data items, and implementation of operations. HOW

Example: ADT List

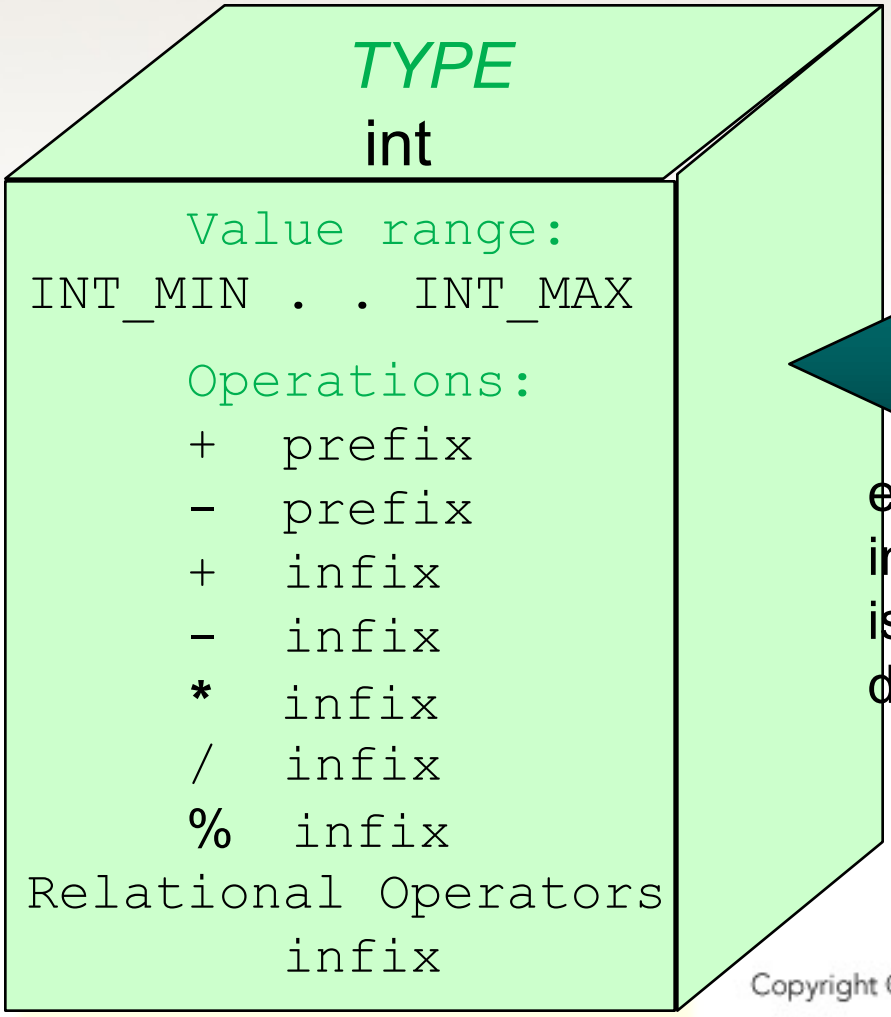
- (***Application or user level)*** modeling real-life list, a homogeneous collection of elements with a linear relation.
  - there is one first element,
  - every element except first one has a unique predecessor
  - every element except last one has a unique successor
- (***Logical level)*** operations supported: PutItem, GetItem, DeleteItem, GetNext, …
- (***Implementation level)*** implemented using array, linked list, or other; codes for operations

Basic ADT Operations

- Constructor -- creates a new instance (object) of an ADT.
- Transformer -- changes state of one or more of the data values of an instance.
- Observer -- allows us to observe the state of one or more of the data values without changing them.
- Iterator -- allows us to process all the components in a data structure sequentially.



C++ Data Type int



encapsulated inside the box is implementation details

Representation of

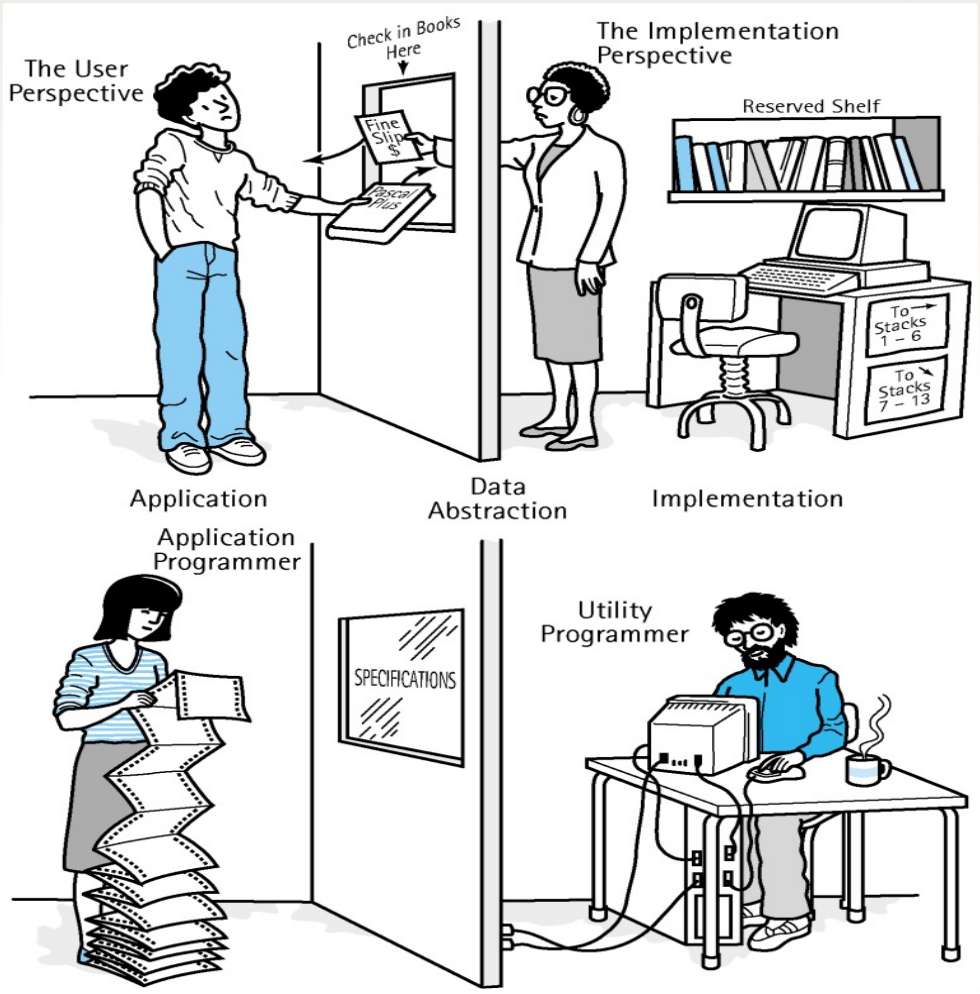
int

as 16 bits two's complement

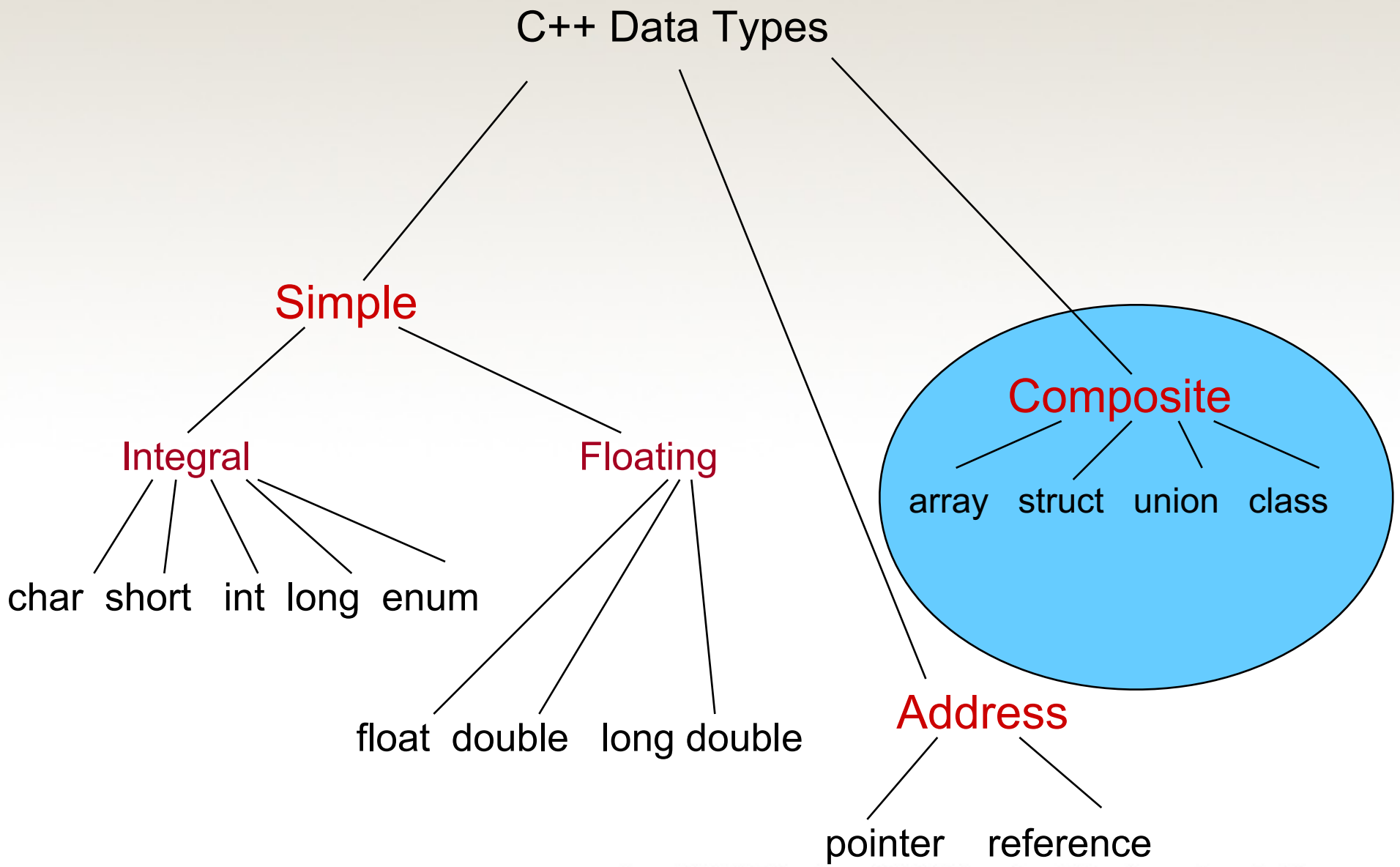
and

Implementation of Operations

Communication between Application Level and Implementation Level



- ***Application (or user) level:*** Library of Congress, or Baltimore County Public Library.
- ***Logical (or ADT) level:*** domain is a collection of books; operations include: check book out, check book in, pay fine, reserve a book.
- ***Implementation level:*** representation of the structure to hold “books”, and the coding for operations.



Composite Data Types

stores a collection of individual data components under one variable name, and allows individual components to be accessed.

UNSTRUCTURED

Components are not organized with respect to one another.

e.g., struct and classes

STRUCTURED

organization of components determines method used to access individual components.

e.g., arrays

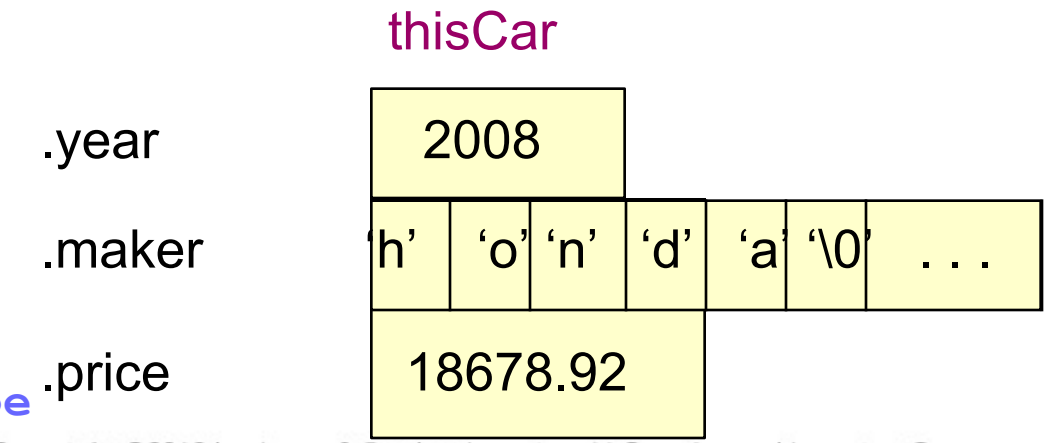
Records (struct in C++)

- A record: a composite data type made up of a finite collection of often times heterogeneous elements, called *members* or *fields*.
- to access individual member:

  recordVar.fieldIdentifier

  e.g., thisCar.year = 2008;

```
struct  CarType
{
  int      year;
  char     maker[10];
  float    price;
};

CarType  thisCar; //CarType
  variables
```



Composite type variables

- Can be assigned
  - each fields/members are assigned
  - thatCar = thisCar;
  - thatCar will be a bit-by-bit copy of thisCar

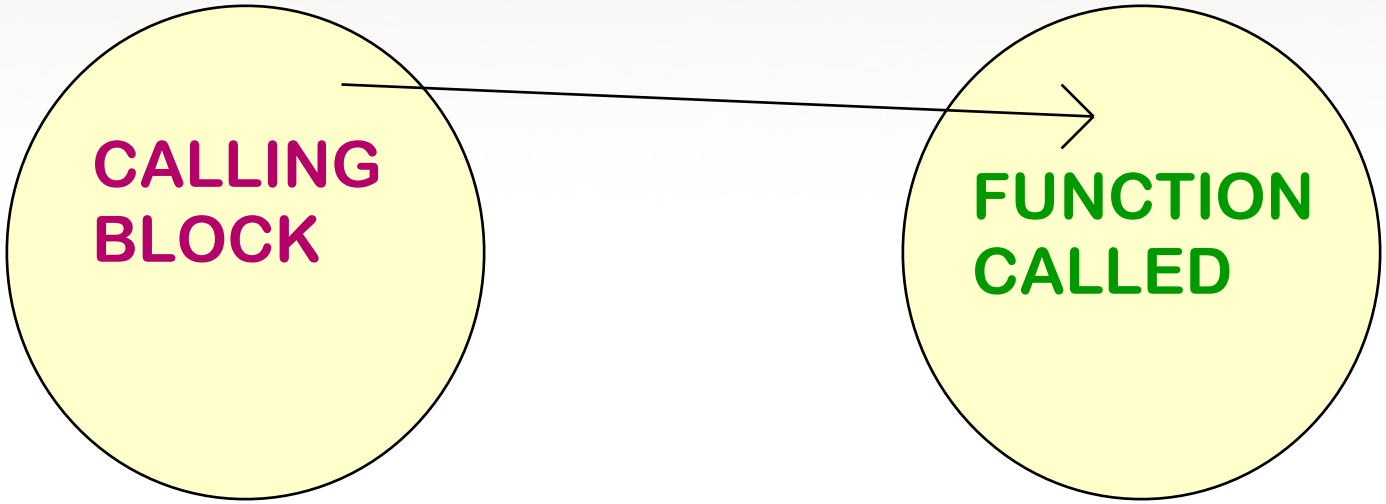
```
struct  CarType
{

  int     year;
  char    maker[10];
  float   price;

};

CarType  thisCar; //CarType
  variables
CarType thatCar;
```

- Can be passed as a parameter to a function (either by value or by reference)
- Can be returned from a function, e.g.,

```
//Prompts user to enter year, price, and maker of a car
// return a CarType value
   CarType ReadUserInput();
```

Pass-by-value

**sends a copy
of the contents of
the actual argument**



SO, actual argument cannot be changed by the function.

Pass struct type by value

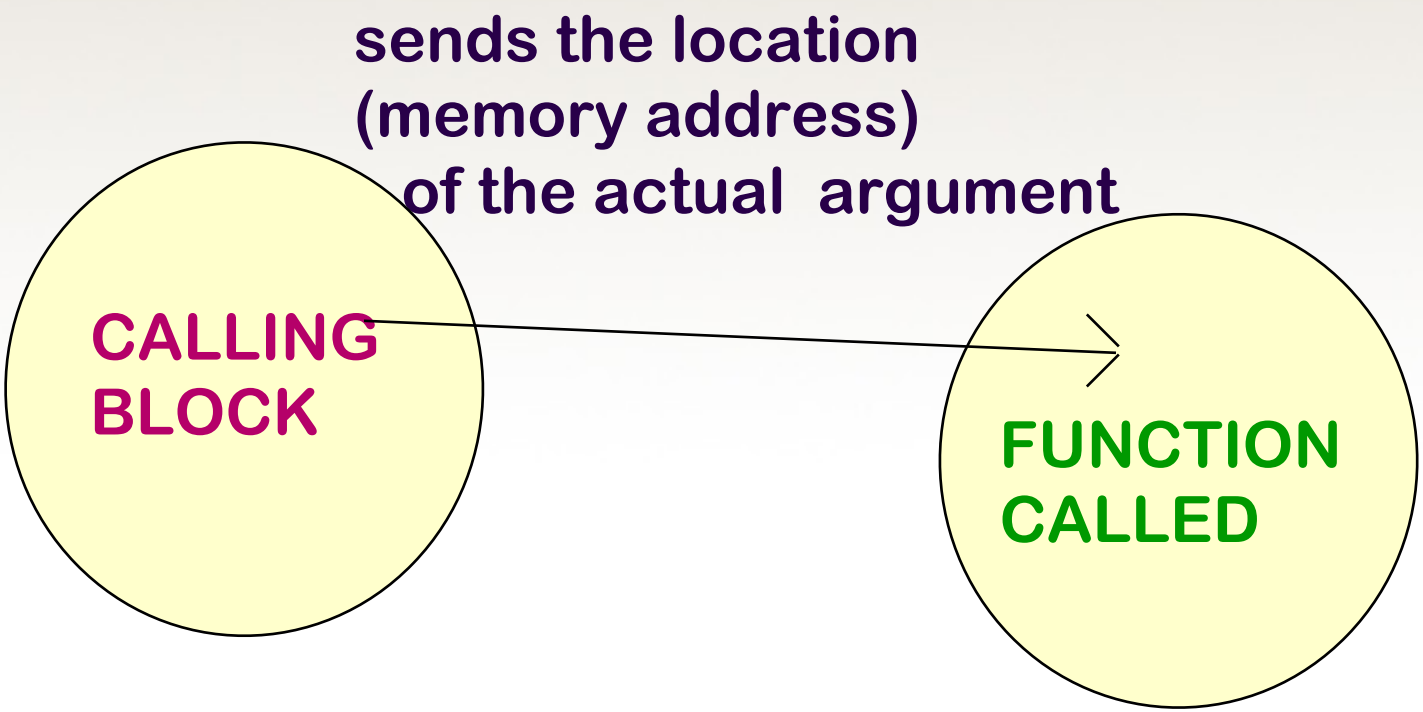
```
bool  LateModel(CarType  car,  int date)

//  Returns true if the car's model year is later than
//  or equal to date;  returns false otherwise.
{

   return  ( car.year  >=  date ) ;

};
```

SAMPLE CALL

Can we modify car in LateModel?

```
myCar.year=2000;
myCar.price = 12,000;
if ( LateModel(myCar, 1995) )
   std::cout << myCar.price << std::endl;
```

Pass-by-reference



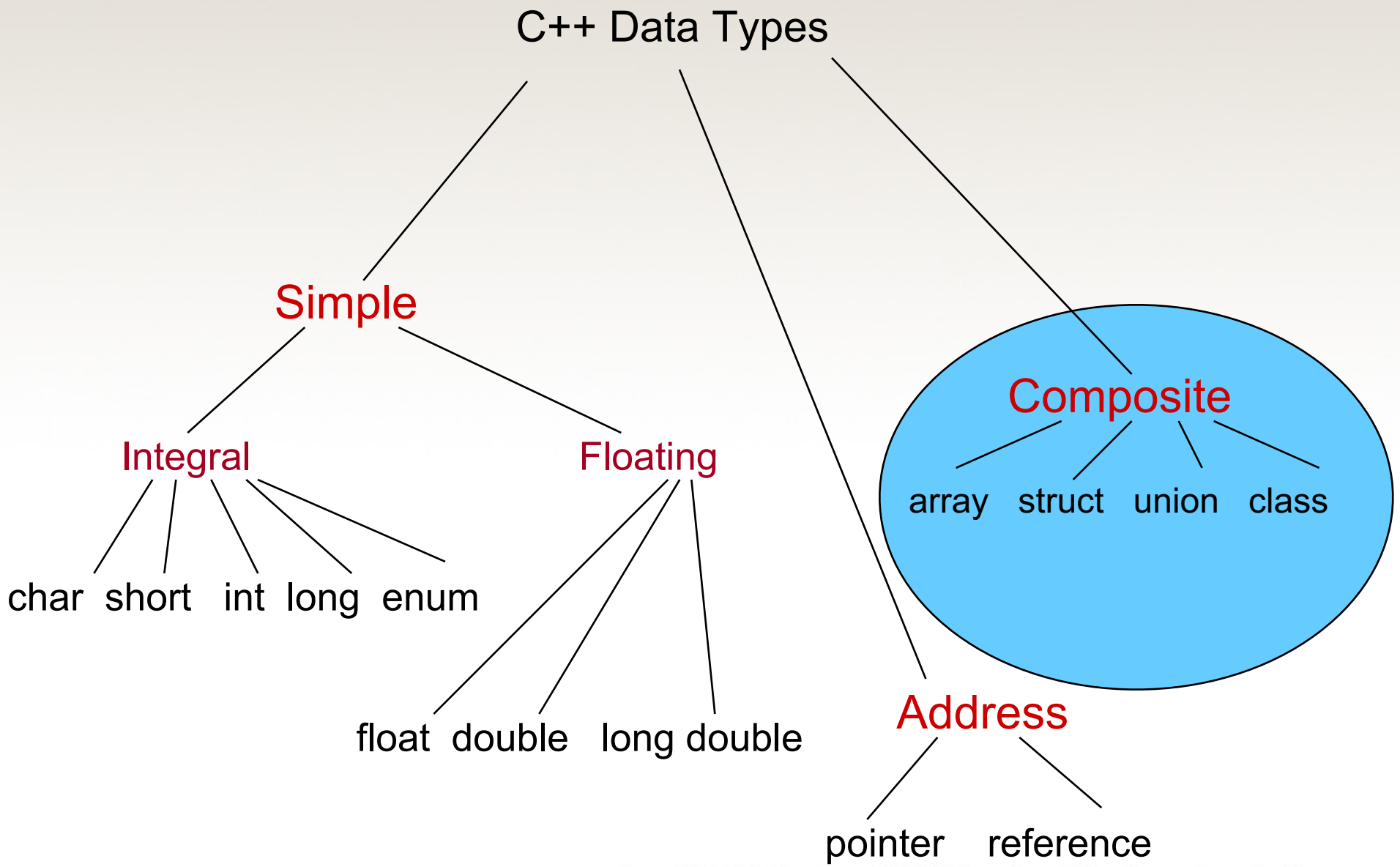
function access actual
argument itself
(not a copy)

Using struct type
Reference Parameter to change a member

```
void  AdjustForInflation(CarType& car, float perCent)
// Increases price by the amount specified in perCent
{

  car.price = car.price * perCent + car.price;

};
```

SAMPLE CALL

```
AdjustForInflation(myCar, 0.03);
```



One-Dimensional Array

- A one-dimensional array is a structured composite data type made up of a finite, fixed size collection of homogeneous (*all of same data type*) elements
- "structured": elements have relative positions (index)
- There is direct access to each element

Array operations (*creation, storing a value, retrieving a value*) are performed using a declaration and indexes.

```
int a[20];
CarType carFleet[100];
a[0]=34;
```

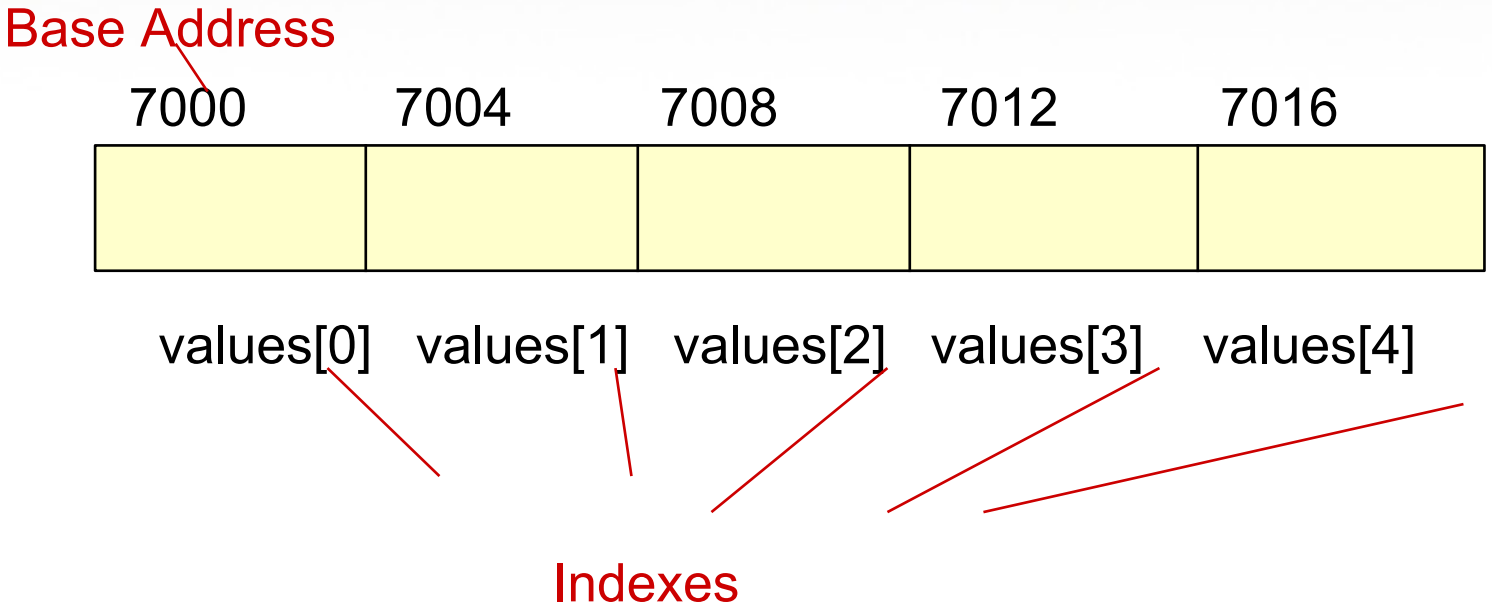
One-Dimensional Arrays in C++

- True or False?
  - Arrays cannot be assigned one to another, and cannot be the return type of a function.

Array Implementation

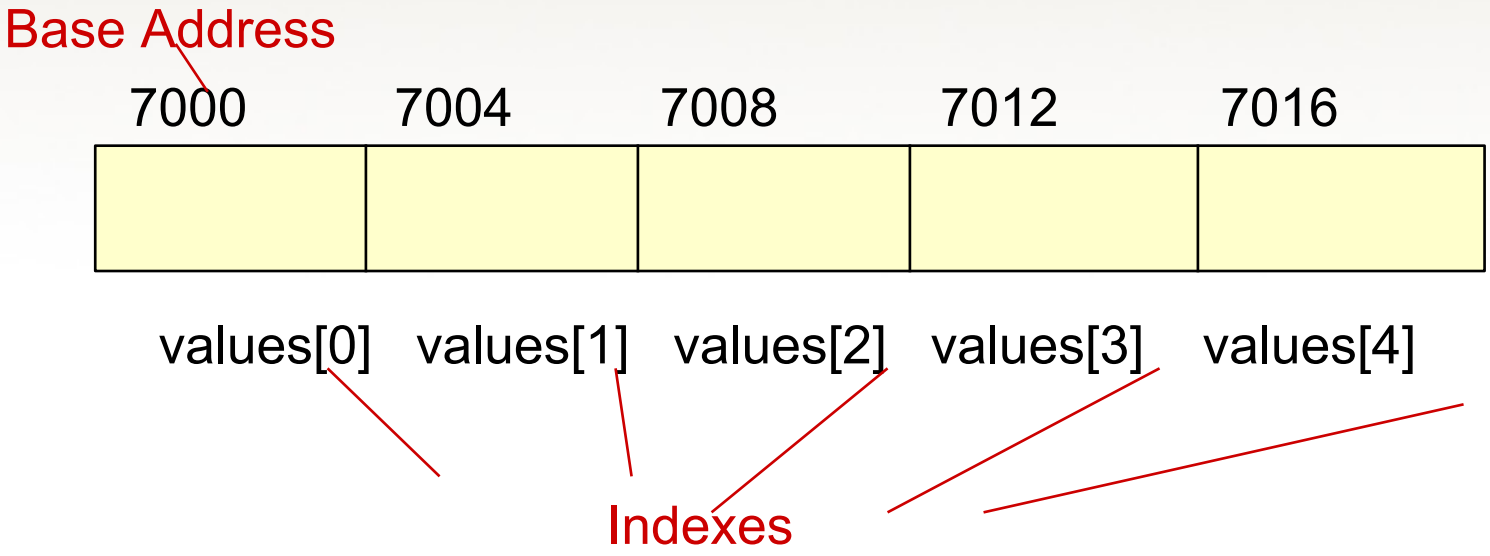
C++ array elements are stored in a contiguous memory block with a base address (lowest address) and size = sizeOfElement*arrayLen;

float values[5]; ***// assume element size is 4 bytes***



Array Implementation: accessing element

float values[5]; ***// assume element size is 4 bytes***



Address of values[Index]?

Address(Index) = BaseAddress + Index * SizeOfElement

What is address of values[0]? values[2]?

Passing Arrays as Parameters

- In C++, arrays are *always* passed by reference, and & is not used with formal parameter type.
  - Whenever an array is passed as an argument, its base address is sent to the called function.
- This means function can modify array
- you can protect array from unintentional changes by using const in formal parameter list and function prototype.

```
float SumValues (const  float  values[ ],
      numOfValues)
//  Pre: values[ 0] through values[numOfValues-1]
//       have been assigned
//  Returns the sum of values[0] through
//  values[numOfValues-1]
{
  float  sum  = 0;
  for ( int  index = 0; index < numOfValues;
        index++ )
  {
    sum = values [index] + sum;
  }
  return  sum;
}
```

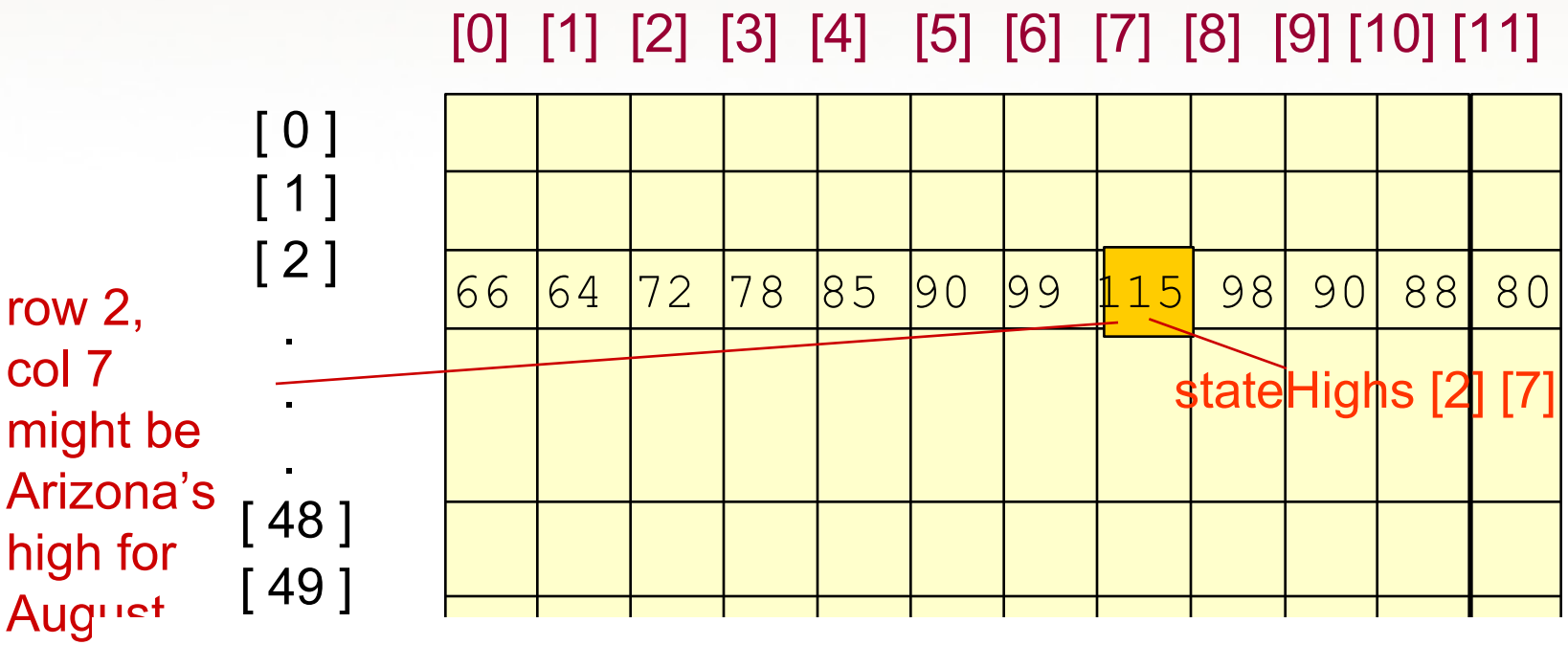
Two-Dimensional Array at the Logical Level

- A two-dimensional array is a structured composite data type made up of a finite, fixed size collection of homogeneous elements having relative positions given by a pair of indexes and to which there is direct access.

- Array operations (*creation, storing a value, retrieving a value*) are performed using a declaration and a pair of indexes (*called row and column*) representing the component's position in each dimension.

EXAMPLE -- To keep monthly high temperatures for 50 states in a two-dimensional array.

```
const  int  NUM_STATES    =  50 ;
const  int  NUM_MONTHS  =  12 ;
int  stateHighs [ NUM_STATES ] [ NUM_MONTHS ] ;
```



How to Find the average high temperature for Arizona?

```
const  int  NUM_STATES    =  50 ;
const  int  NUM_MONTHS  =  12 ;
int  stateHighs [ NUM_STATES ] [ NUM_MONTHS ] ;
```

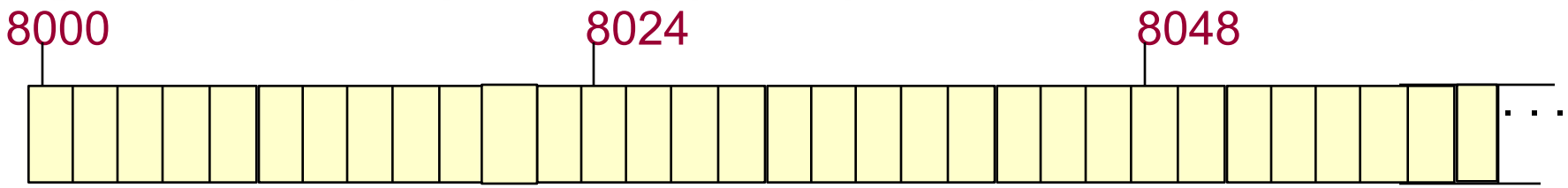
rows

columns

STORAGE

- In memory, C++ stores 2D arrays in row order: the first row is followed by the second row, etc.

Base Address

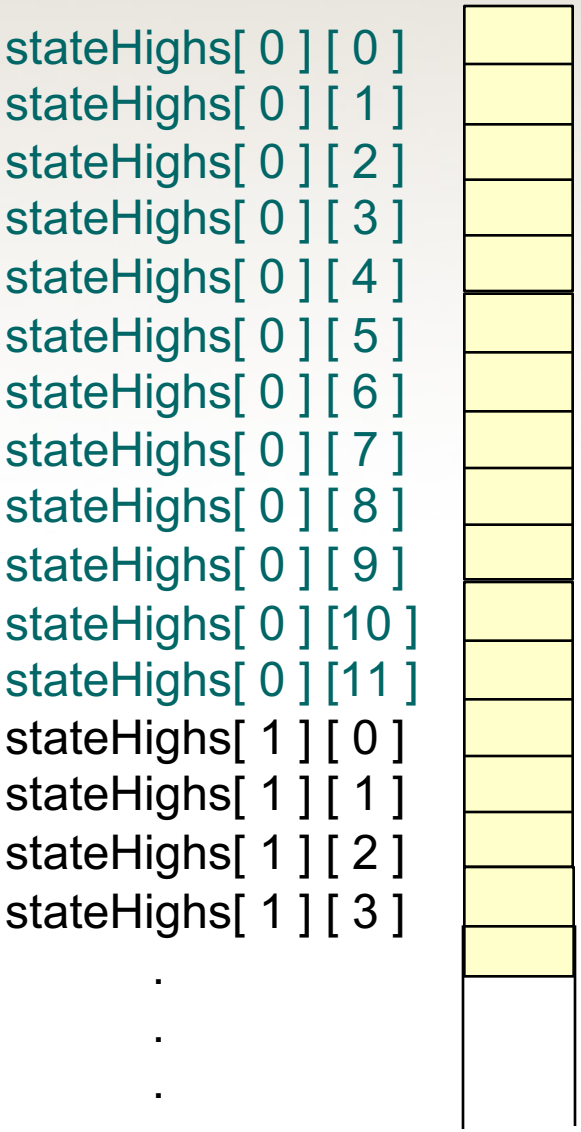


12 highs for state 0
Alabama
first row

12 highs for state 1
Alaska
second row

etc.

Implementation Level View



Base Address 8000

To locate an element such as
stateHighs [2] [7]
the compiler needs to know
that there are 12 columns
in this two-dimensional array.

At what address will
stateHighs [2] [7] be found?

baseAddress+(2*12+7)*2

There are 2*12+7 elements before it
Assume 2 bytes for type int.

Two-Dimensional Array Parameters

- Just as with a one-dimensional array, when a two- (or higher) dimensional array is passed as a parameter, base address of array is sent to the function.
- size of all dimensions except the first must be included in function heading and prototype.
- sizes of those dimensions for the formal parameter must be exactly the same as in the actual array.

```
void  findAverages (const int  stateHighs [  ] [ NUM_MONTHS],
                    int   stateAverages [  ])
```

```
//  Pre:   stateHighs[ 0..NUM_STATES-1] [ 0..NUM_MONTHS-1]
//  assigned
//  Post:  stateAverages[ 0..NUM_STATES-1 ] contains rounded
//  high temperature for each state
{
  int  state;
  int  month;
  float  total;
  for  ( state = 0 ;  state  <  NUM_STATES;  state++ )
  {
    total = 0.0;
    for ( month = 0 ;  month  <  NUM_MONTHS ; month++ )
       total = stateHighs [ state ][ month ] + total;
    stateAverages [ state ] = total / 12.0 + 0.5;
  }
}
```

- Use the two-dimensional stateHighs array to fill a one-dimensional stateAverages array

Using **typedef** with arrays

helps eliminate chances of size mismatches between formal and actual parameters.

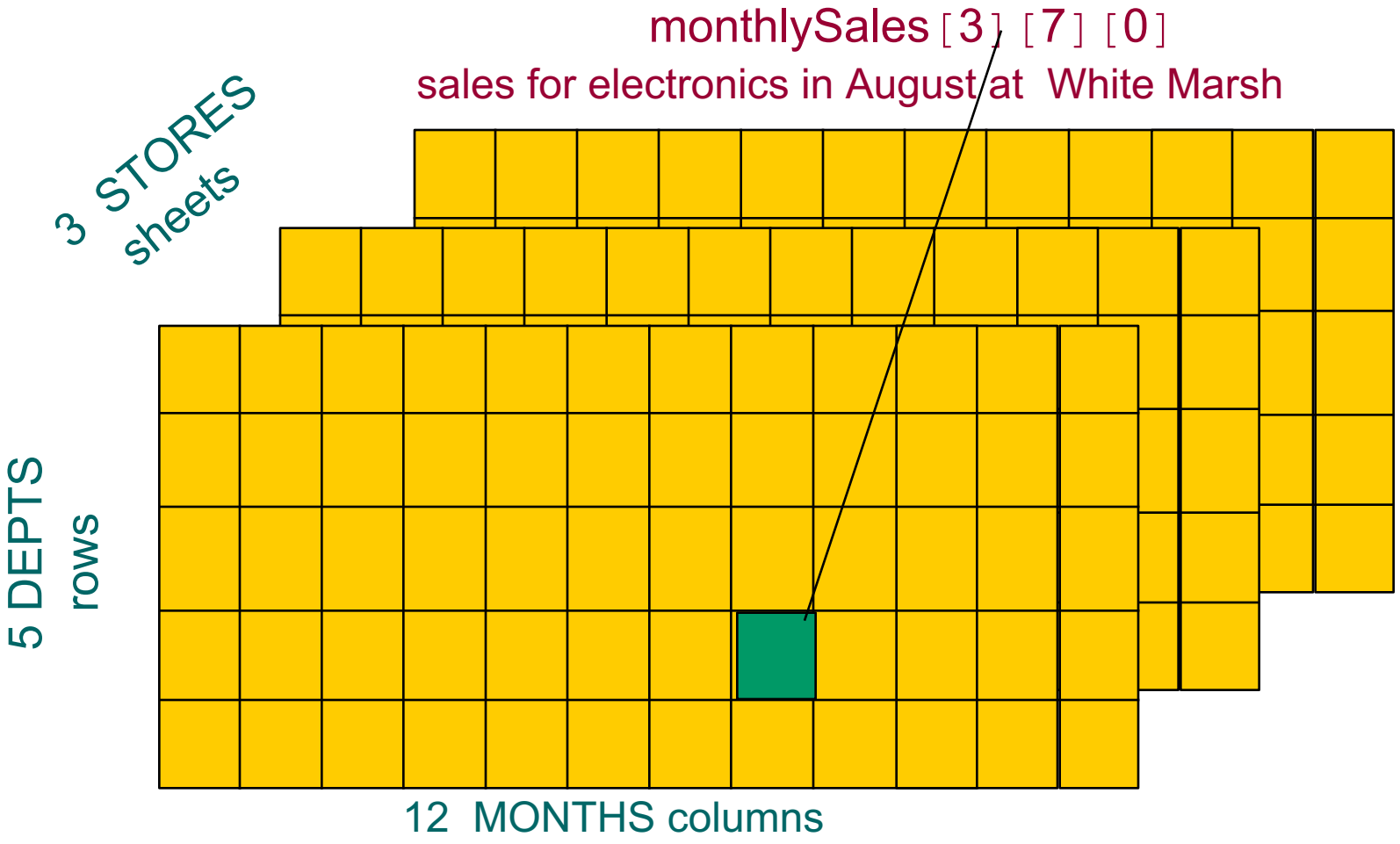
```
typedef int  StateHighsType [ NUM_STATES ][ NUM_MONTHS ];

typedef float  StateAveragesType [ NUM_STATES ];

void  findAverages( const  StateHighsType  stateHighs,
                    StateAveragesType  stateAverages  )
{
            .
            .
            .
}
```

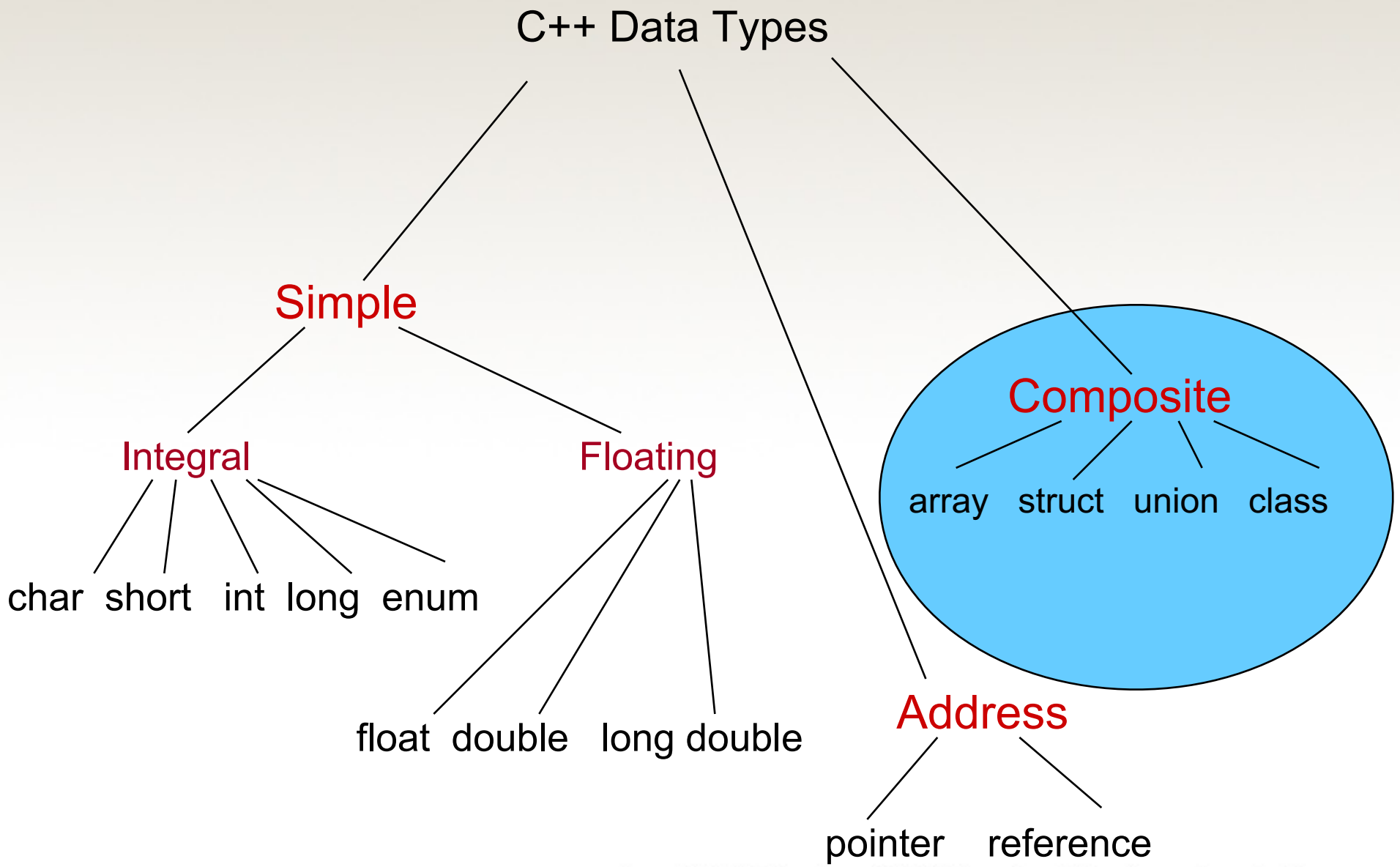
Declaring Multidimensional Arrays

EXAMPLE USING TYPEDEF

```
const int NUM_DEPTS = 5;
// mens, womens, childrens, electronics, linens
const int NUM_MONTHS =  12 ;
const int NUM_STORES =  3 ;
// White Marsh,  Owings Mills, Towson

typedef  long MonthlySalesType [NUM_DEPTS] [NUM_MONTHS]
                               [NUM_STORES];

MonthlySalesType   monthlySales;
```

```
const int NUM_DEPTS = 5; // mens, womens, childrens, electronics, linens
const int NUM_MONTHS  =  12 ;
const int NUM_STORES  =   3 ;    // White Marsh,  Owings Mills, Towson

typedef long MonthlySalesType [NUM_DEPTS] [NUM_MONTHS] [NUM_STORES] ;
MonthlySalesType   monthlySales;
```





C++ **class** data type

- A class is an unstructured type that encapsulates a fixed number of data components (data members) with the functions (called member functions) that manipulate them.
- predefined operations on an instance of a class are whole assignment and component access.

class DateType Specification

```
//  SPECIFICATION FILE      ( datetype.h )

class  DateType                 // declares a  class data type
{

public :
   //  4 public member functions
   void Initialize (int newMonth, int newDay, int newYear ) ;
   int  GetYear( )  const ;       // returns year
   int  GetMonth( ) const ;       // returns month
   int  GetDay( )   const ;       // returns day

private :
   //  3 private data members
   int  year ;
   int  month ;
   int  day ;
};
```

; must be there!!!

Use of C++ data type **class**

- Variables of a class type are called objects (or instances) of that particular class.
- Software that declares and uses objects of the class is called a client.
- Client code uses public member functions (called methods in OOP) to handle its class objects.
- **-** means calling a public member function.

Client Code Using **DateType**

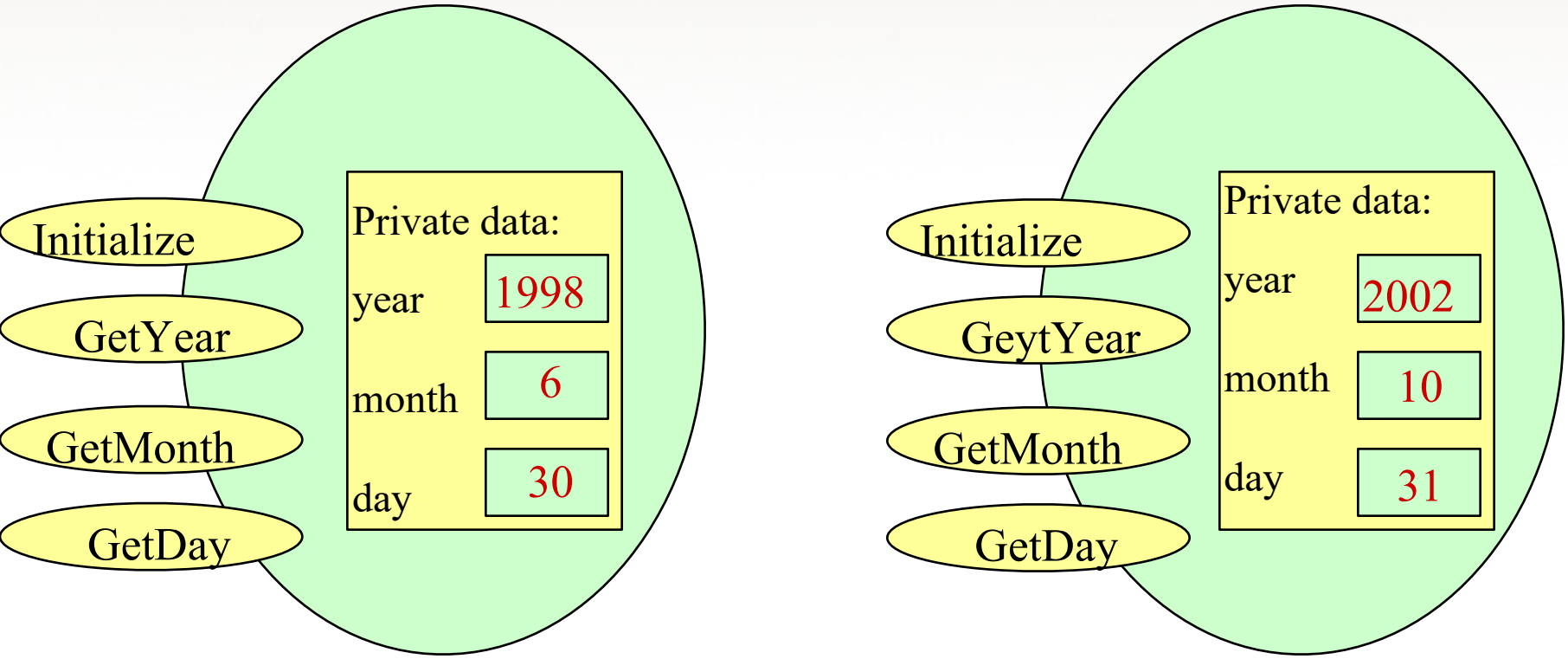
```
#include "datetype.h"  // includes specification of the class
using namespace std;
int  main ( void )
{
  DateType  startDate; // declares 2 objects of DateType
  DateType  endDate;
  bool  retired  =  false;
  startDate.Initialize ( 6, 30, 1998 );
  endDate.Initialize ( 10, 31, 2002 );
  cout <<  startDate.GetMonth( ) << "/" << startDate.GetDay( )
       << "/" <<  startDate.GetYear( ) << endl;
  while ( ! retired )
  {
    // finishSomeTask
  }
}
```

2 separate files generally used for **class** type

```
//   SPECIFICATION FILE        ( datetype .h )
//   Specifies the data and function members.
 class DateType
 {
 public:
          . . .
 private:
          .  .  .
 } ;
```

```
//   IMPLEMENTATION FILE      ( datetype.cpp )

//   Implements the DateType member functions.

     . . .
```

DateType Class Instance Diagrams



Implementation of **DateType** member functions

```
// IMPLEMENTATION FILE                          (datetype.cpp)
#include "datetype.h"    // also must appear in client code

void DateType :: Initialize ( int  newMonth, int  newDay,
                              int  newYear )
//  Post:  year is set to newYear.
//         month is set to newMonth.
//         day is set to newDay.
{
   year =  newYear;
   month =  newMonth;
   day  =  newDay;
}
```

```
int DateType :: GetMonth ( ) const
//  Accessor function for data member month
{
  return  month;
}

int DateType :: GetYear ( ) const
//  Accessor function for data member year
{
  return  year;
}

int DateType :: GetDay ( ) const
//  Accessor function for data member day
{
  return  day;
}
```

Familiar Class Instances and Member Functions

- member selection operator (.) selects either data members or member functions.
- Header files iostream and fstream declare istream, ostream,and ifstream, ofstream I/O classes.
- Both cin and cout are class objects and get and ignore are member functions.

```
cin.get (someChar);
cin.ignore (100, '\n');
```

- These statements declare myInfile as an instance of class ifstream and invoke member function open.

```
ifstream  myInfile ;
myInfile.open ( "mydata.dat" );
```

Scope Resolution Operator (::)

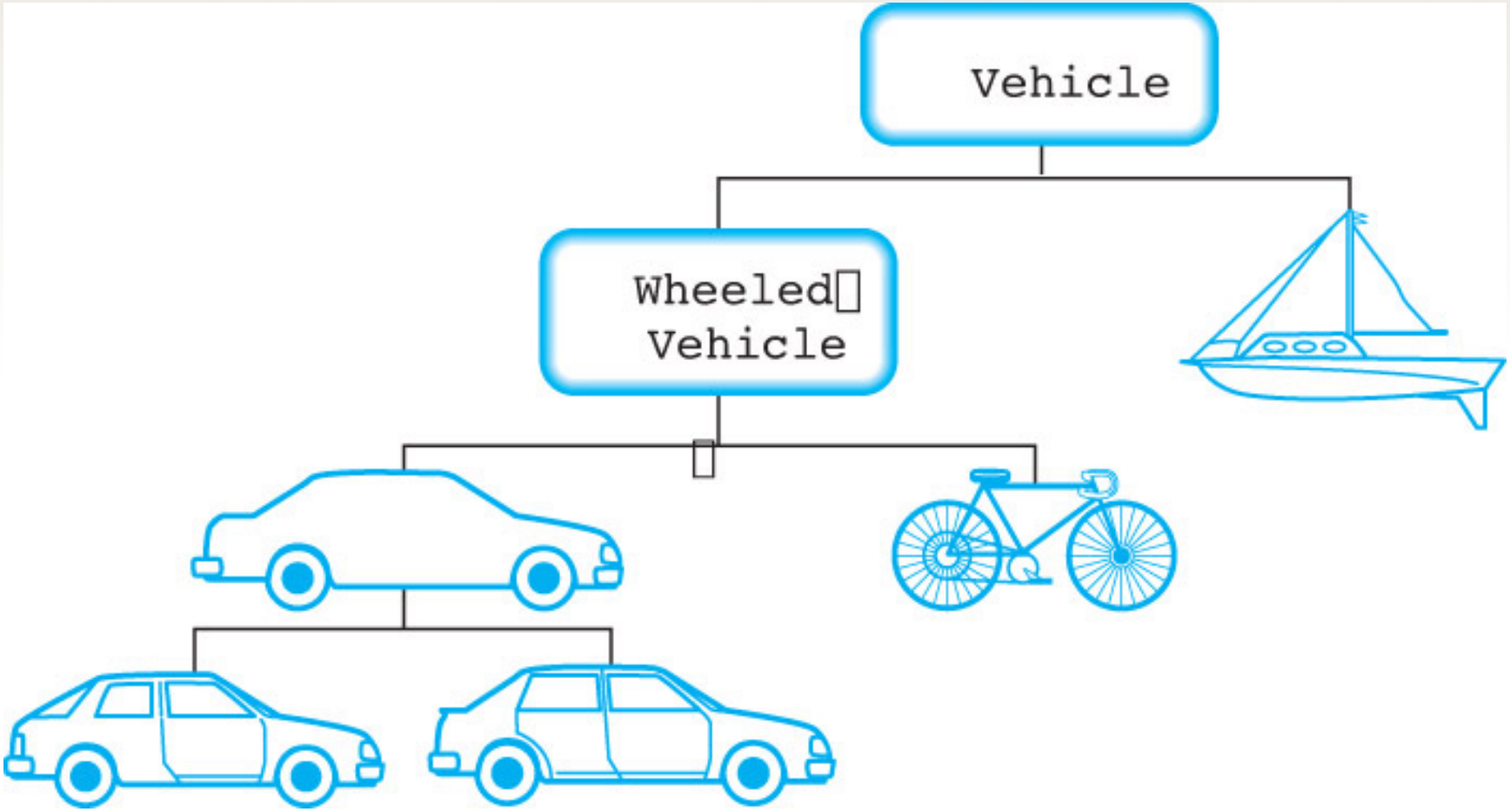
- C++ programs typically use several class types.
- Different classes can have member functions with the same identifier, like Write( ).
- Member selection operator is used to determine the class whose member function Write( ) is invoked.

```
currentDate.Write( ) ;  // class DateType
numberZ.Write( ) ; // class ComplexNumberType
```

- In implementation file, scope resolution operator is used in heading before function's name to specify its class.

```
void  DateType :: Write ( )   const
{          . . .

}
```

Inheritance



Object-Oriented Programming

- Three inter-related constructs: classes, objects, and inheritance
- Objects are basic run-time entities in an object-oriented system.
- A class defines the structure of its objects.
- Classes are organized in an “is-a” hierarchy defined by inheritance.

Inheritance

- Allows programmers to create a new class that is a specialization of an existing class.
  - new class is called a derived class of the existing class
  - the existing class is the base class of the new class.

Inheritance

- Inheritance fosters reuse by allowing an application to take an already-tested class and derive a class from it that inherits the properties the application needs

- Polymorphism: the ability of a language to have duplicate method names in an inheritance hierarchy and to apply the method that is appropriate for the object to which the method is applied

```
# include <string>
class MoneyType
{
public:
  void Initialize(long, long);
  long DollarsAre( ) const;
  long CentsAre( ) const;
private:
  long dollars;
  long cents;
};
```

```
class ExtMoneyType:public MoneyType
{
public:
  string CurrencyIs( );
  void Initialize(long, long, const string);
private:
  string currency;
};
ExtMoneyType extMoney;
void ExtMoneyType::Initialize
  (long newDollars, long newCents, string newCurrency)
{
  currency = newCurrency;
  MoneyType::Initialize(newDollars, newCents);
}
String ExtMoneyType::CurrencyIs() const
{
  return currency;

}
```

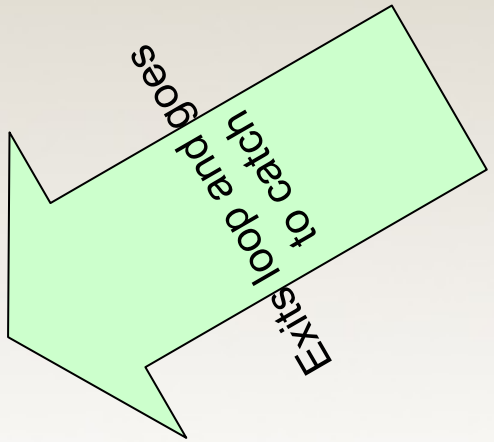
Exceptions

- An exception is an unusual situation that occurs when the program is running.
- Exception Management
  - Define the error condition
  - Enclose code containing possible error (try).
  - Alert the system if error occurs (throw).
  - Handle error if it is thrown (catch).

try, catch, and throw

```
Try
{
   // code that contains a possible error
   … throw string("An error has occurred in function
   …");


}
Catch (string message)
{
   std::cout << message << std::endl;
   return 1;
}
```

```
try
{
  infile >> value;
  do
  {
    if (value < 0)
      throw string("Negative value");
    sum = sum + value;
  } while (infile);
}
catch (string message)
// Parameter of the catch is type string
{
  // Code that handles the exception
  cout << message << " found in file. Program aborted."
  return 1;
}
// Code to continue processing if exception not thrown
cout << "Sum of values on the file: " << sum;
```



Namespace

```
namespace mySpace
{
  // All variables and
 // functions within this
  // block must be accessed
 // using scope
  // resolution operator (::).
}
```

Purpose: Avoid namespace pollution.

Three Ways to Access Members within a Namespace

- Qualify each reference:

  `mySpace::name` with every reference.

- Using declaration:

  ```
  using mySpace::name;
  ```

  All future references to name refer to mySpace::name.

- Using directive:

  ```
  using namespace mySpace;
  ```

  All members of mySpace can be referenced without qualification.

Rules for Use of Namespace std
(within text)

- Qualify names in prototypes and/or function definitions.
- If name used more than once in a function block, use a using declaration.
- If more than one name is used from a namespace, use a using directive.

Enum Variable

```
#include <iostream>
using namespace std;
enum direction {East, West, North, South};
int main(){
    direction dir;
    dir = West;
    cout<<dir;
    return 0;
}
```

Enum is a user defined data type where we specify a set of values for a variable and the variable can only take one out of a small set of possible values. We use enum keyword to define a Enumeration.

Enum with switch

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

int main(){
enum Color { red, green, blue };
Color r = red;
switch(r)
{
   case red  : std::cout << "red\n";   break;
   case green: std::cout << "green\n"; break;
   case blue : std::cout << "blue\n";  break;
}
cout << "value of r is: " << r;
return 1;
}
```