C++ Programming Essentials

Unit 3: Basic Abstract Data Types

CHAPTER 10: STRUCTURED TYPES AND CLASSES

DR. ERIC CHOU

IEEE SENIOR MEMBER

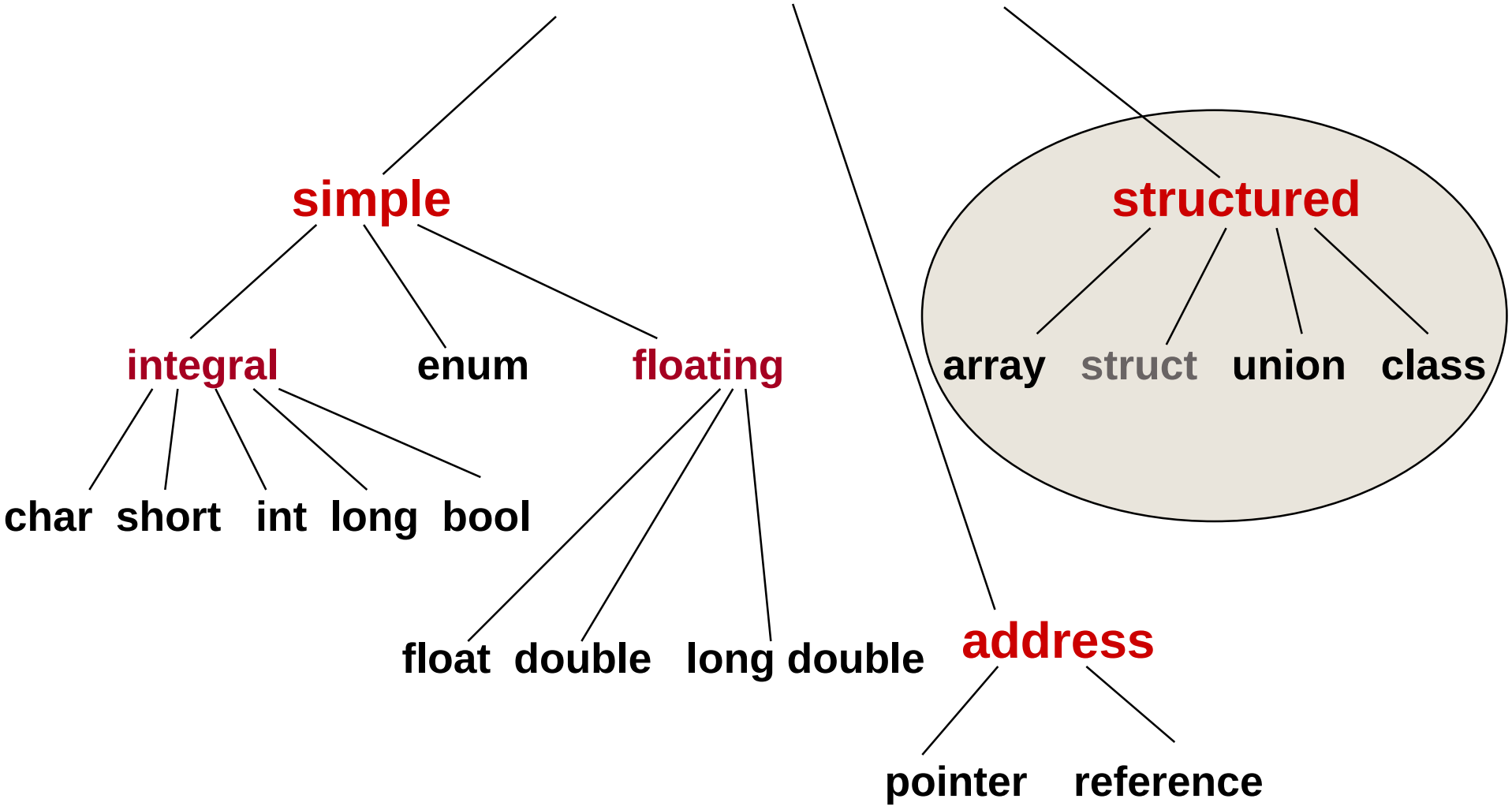
LECTURE 1

Structured Data Types

Chapter 10 Topics

- Meaning of a Structured Data Type
- Declaring and Using a struct Data Type
- C++ union Data Type
- Meaning of an Abstract Data Type
- Declaring and Using a class Data Type
- Using Separate Specification and Implementation Files
- Invoking class Member Functions in Client Code
- C++ class Constructors

C++ Data Types


simple
integral
enum
floating
char
short
int
long
bool
float
double
long double
structured
array
struct
union
class
address
pointer
reference

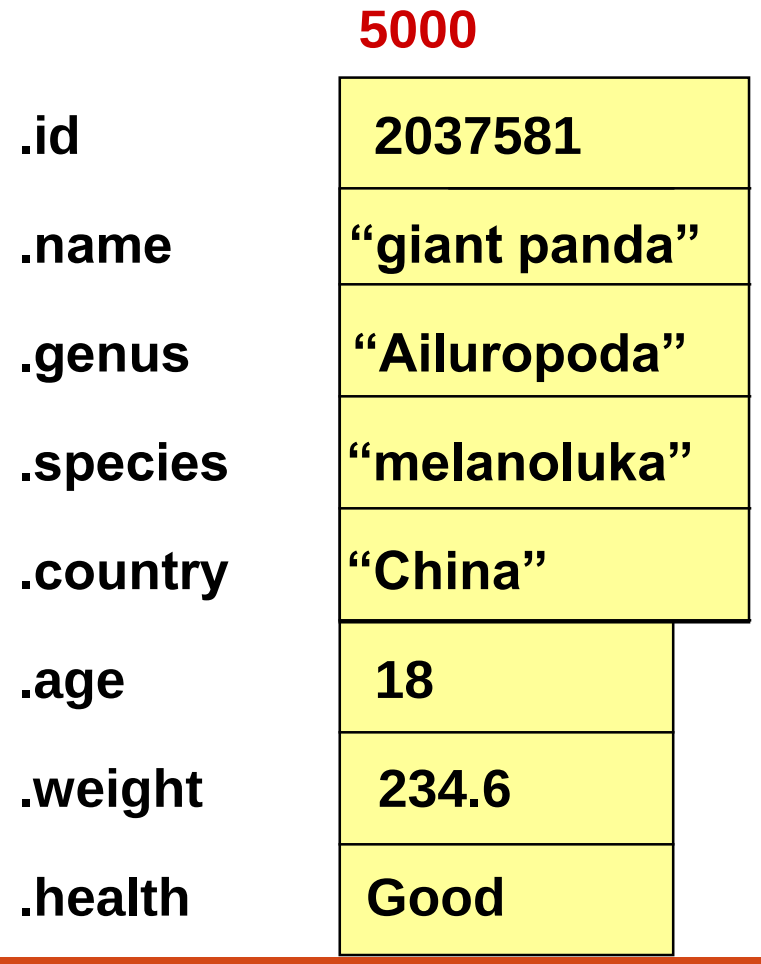
Structured Data Type

- A structured data type is a type in which each value is a collection of component items.
- the entire collection has a single name
- each component can be accessed individually

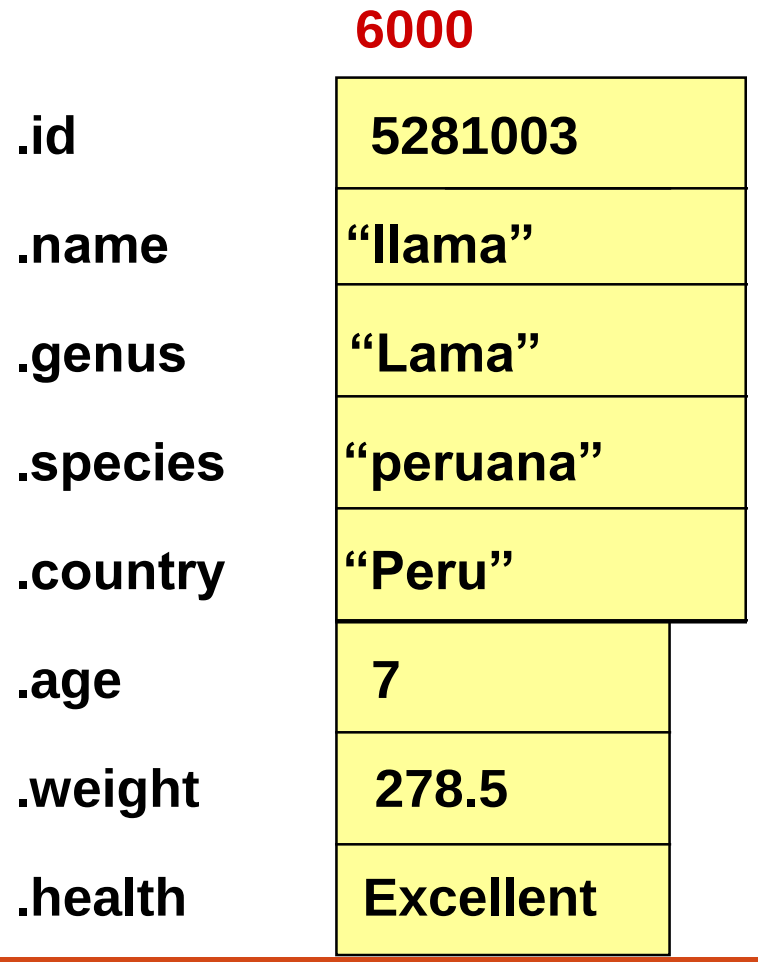
C++ Structured Type

often we have related information of various types that we'd like to store together for convenient access under the same identifier, for example . . .

thisAnimal


5000
.id
2037581
.name
“giant panda”
.genus
“Ailuropoda”
.species
“melanoluka”
.country
“China”
.age
18
.weight
234.6
.health
Good

anotherAnimal



class versus struct

Class (class):

- Template of objects
- Member of a class:
  1. Data fields (variable, struct, union, enum, or class)
  2. Member Functions

Structure (struct):

- Template of records
- Member of Structure:
  1. Data fields (variable, struct, union, enum or class)

array versus struct

Array (Homogeneous Data Collection):

- Not data structure template
- Member of an array:
  1. Data elements of a same data type

Structure (struct):

- Template of records
- Member of Structure:
  1. Data fields (variable, struct, union, enum or class)

LECTURE 2

struct Type Declaration and Usage

C++ struct (Record Data Type)

Syntax:

```
struct type_name {   // type_name is the name of this struct type
 member_type1 member_name1;
 member_type2 member_name2;
 member_type3 member_name3;
 .
 .
} object_names;
```

`struct` type Declaration

SYNTAX

```
struct  TypeName                    // does not  allocate memory
{
  MemberList
};
```

MemberList SYNTAX

```
DataType   MemberName ;
DataType   MemberName ;
    .
    .
    .
```

Difference Between C and C++

C Syntax:

```
struct name {
  // member data fields
} object_name;
```

Record Declaration:

```
struct name object_name;
```

typedef Declaration:

```
typedef struct name type_name;
```

C Syntax:

```
struct type_name {
  // member data fields
} object_name;
```

Record Declaration:

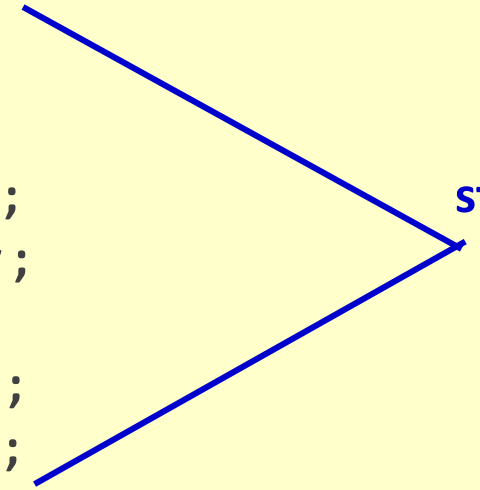
```
type_name object_name;
```

typedef Declaration

creates an alias that can be used anywhere in place of a (possibly complex) type name.

struct AnimalType (I)

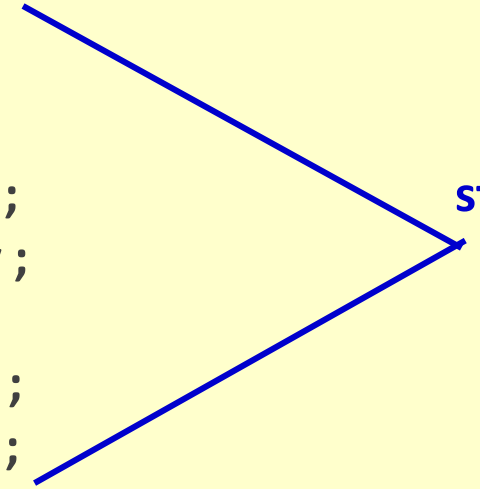
```
enum  HealthType  { Poor,  Fair,  Good,  Excellent } ;

struct  AnimalType                    // declares a  struct data type
{                                     //  does not allocate memory
 long          id ;
 string        name ;
 string        genus ;
 string        species ;
 string        country ;
 int           age ;
 float         weight ;
 HealthType    health ;
} ;

AnimalType    thisAnimal ;        // declare  variables of AnimalType
AnimalType    anotherAnimal ;     // these instantiates struct records
```



struct members

struct AnimalType (II)

```
enum  HealthType  { Poor,  Fair,  Good,  Excellent } ;

struct  AnimalType                    // declares a  struct data type
{                                     //  does not allocate memory
 long          id ;
 string        name ;
 string        genus ;
 string        species ;
 string        country ;
 int           age ;
 float         weight ;
 HealthType  health ;
} thisAnimal, anotherAnimal ;

// declare  variables of AnimalType
 // these instantiates struct records
```



struct members

`struct` type Declaration

- The struct declaration names a type and names the members of the struct.
- It does not allocate memory for any variables of that type!
- You still need to declare your struct variables.

More about `struct` type declarations

- If the struct type declaration precedes all functions it will be visible throughout the rest of the file. If it is placed within a function, only that function can use it.
- It is common to place struct type declarations with TypeNames in a (.h) header file and #include that file.
- It is possible for members of different struct types to have the same identifiers. Also a non-struct variable may have the same identifier as a structure member.

Accessing `struct` Members

- Dot **.** (period) is the member selection operator.
- After the struct type declaration, the various members can be used in your program only when they are preceded by a struct variable name and a dot.

EXAMPLES

```
thisAnimal.weight

anotherAnimal.country
```

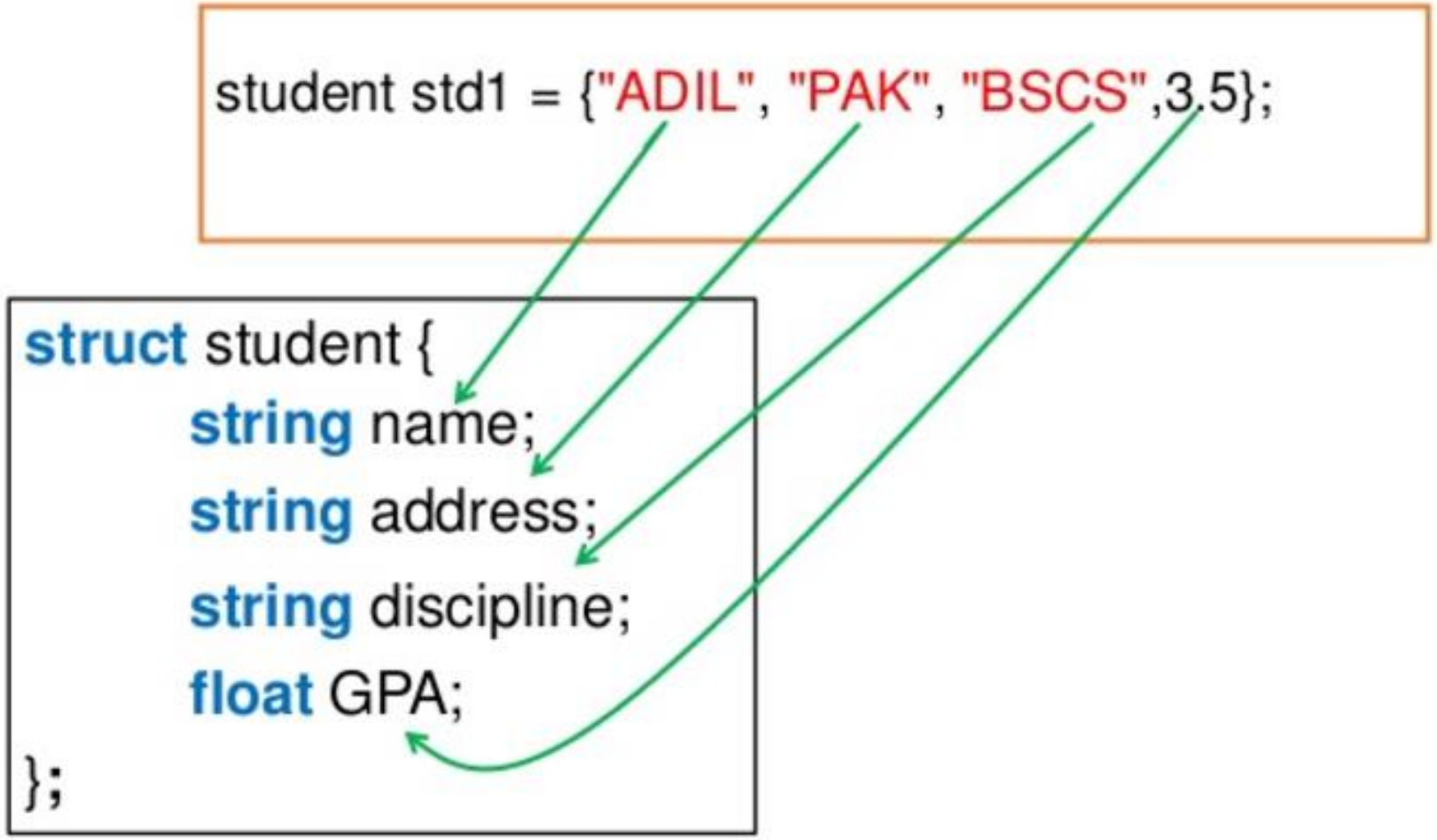
Valid operations on a struct member depend only on its type

```
thisAnimal.age  =  18;

thisAnimal.id    =  2037581;

cin  >>  thisAnimal.weight;

getline ( cin, thisAnimal.species );

thisAnimal.name = “giant panda”;

thisAnimal.genus[ 0 ] = toupper (thisAnimal.genus[ 0 ] ) ;

thisAnimal.age++;
```

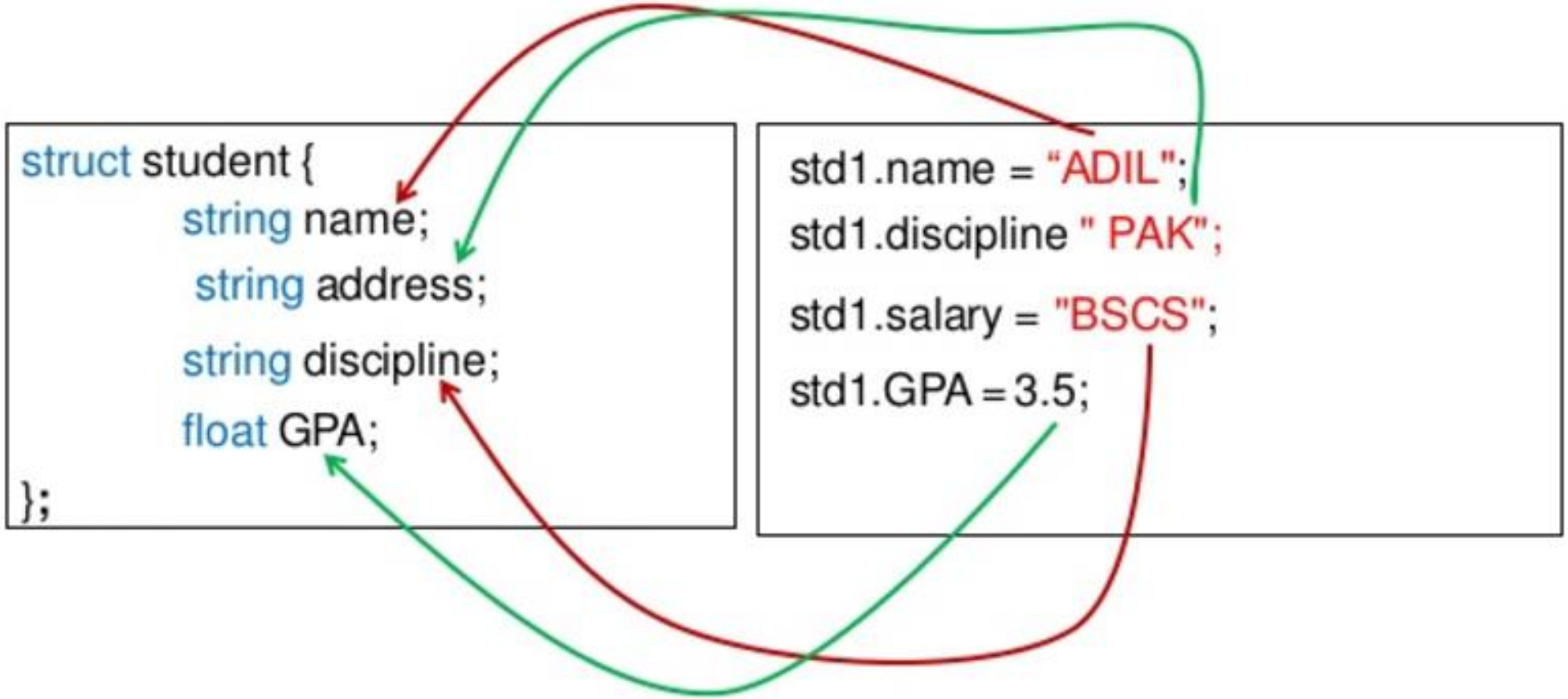
LECTURE 3

Initialization of struct records

Initializing Structures(1st Way)



Initializing Structures(2nd Way)



Initializing Structures (3rd Way: Out of Order Assignments)

```
struct {
    int sec, min, hour, day, mon, year;
 } z = {.day=31, 12, 2014, .sec= 30, 15, 17};

// initializes z to {30, 15, 17, 31, 12, 2014}
```

Initializing Structures (4th Way: Using Constructor())
A struct is a special class without method in C++

By Constructor

```
1  struct Date
2  {
3      int day;
4      int month;
5      int year;
6
7      Date()
8      {
9          day=0;
10         month=0;
11         year=0;
12     }
13 };
```

By default value of a constructor decl.

```
1  struct Date
2  {
3      int day;
4      int month;
5      int year;
6
7      Date():day(0),
8             month(0),
9             year(0){}
10 };
```

By parameters of a constructor

```
1  struct Date
2  {
3      int day;
4      int month;
5      int year;
6
7      Date(int d=0, int m=0, int y=0):day(d),
8             month(m),
9             year(y){}
10 };
```

```
Date d( 4, 2, 42 );
```

Copy the Contents by Assignment

class has similar assignment operation called copy constructor

Structure Variable in Assignment Statement
S1 = S2;
• The statement assigns the value of each member of S2 to the corresponding member of S1. Note that one structure variable can be assigned to another only when they are of the same structure type, otherwise complier will give an error.

LECTURE 4

Aggregate Operations for struct Type

Aggregate Operation

is an operation on a data structure as a whole, as opposed to an operation on an individual component of the data structure

Aggregate `struct` Operations

- I/O, arithmetic, and comparisons of entire struct variables are NOT ALLOWED!
- Operations valid on an entire struct type variable:
  - Assignment to another struct variable of same type,
  - Pass to a function as argument (by value or by reference),
  - Return as value of a function

Array Versus struct

Aggregate Operation	Array	struct
Arithmetic	No	No
Assignment	No	Yes
Input/output	No (except strings)	No
Comparison	No	No
Parameter passing	By reference only	By value or by reference
Function returning a value	No	Yes

Examples of aggregate `struct` operations

```
anotherAnimal   =   thisAnimal ;       // assignment

WriteOut(thisAnimal);                  // value parameter

ChangeWeightAndAge(thisAnimal); // reference parameter

thisAnimal = GetAnimalData( );    // return value of function
```

NOW WE'LL WRITE THE 3 FUNCTIONS USED HERE . . .

```
void  WriteOut( /* in */ AnimalType  thisAnimal)
// Prints out values of all members of thisAnimal
// Precondition:     all members of thisAnimal are assigned
// Postcondition:     all members have been written out
{
 cout  <<  "ID # "  <<  thisAnimal.id  <<  thisAnimal.name  << endl ;
 cout  <<  thisAnimal.genus  <<  thisAnimal.species << endl ;
 cout  <<  thisAnimal.country << endl ;
 cout  <<  thisAnimal.age  << " years "  << endl ;
 cout  <<  thisAnimal.weight << " lbs. "  << endl ;
 cout  << "General health : " ;
    WriteWord ( thisAnimal.health ) ;
}
```

LECTURE 5

Passing struct Data Type by Reference (alias)

Passing a `struct` Type by Reference (Alias)

```
void   ChangeAge ( /* inout */   AnimalType&   thisAnimal )   // pass by alias

// Adds 1 to age
// Precondition:     thisAnimal.age is assigned
// Postcondition:    thisAnimal.age == thisAnimal.age@entry + 1

{

 thisAnimal.age++ ;

}
```

```
AnimalType   GetAnimalData ( void )
// Obtains all information about an animal from keyboard
// Postcondition:
//   Function value == AnimalType members entered at  kbd
{
    AnimalType  thisAnimal ;
    char            response ;
    do  {            //  have user enter all members until they are correct
                     .
                     .
                     .
    }  while (response != 'Y' ) ;
    return  thisAnimal ;
}
```

Demo Program:

animal.cpp

Go Dev C++!!!

LECTURE 6

Hierarchical struct (Compound struct)

Hierarchical Structures

- The type of a struct member can be another struct type. This is called nested or hierarchical structures.
- Hierarchical structures are very useful when there is much detailed information in each record.

FOR EXAMPLE . . .

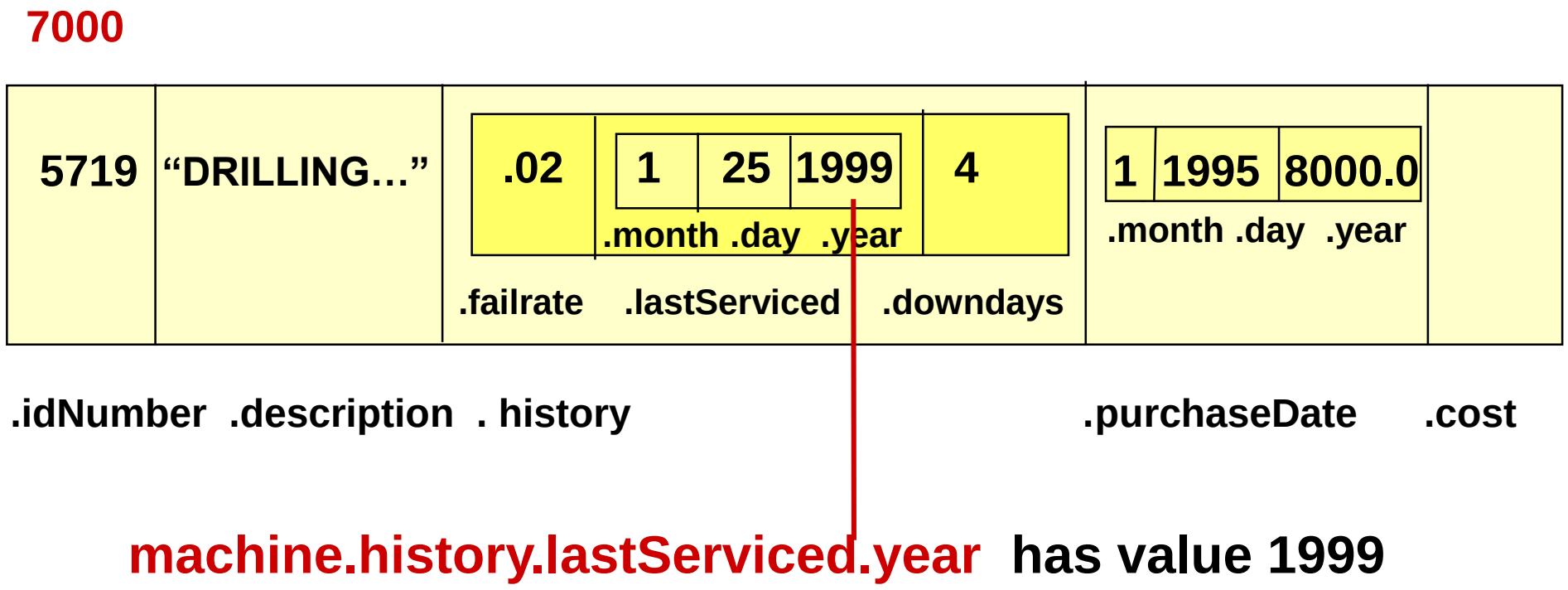
struct MachineRec

Information about each machine in a shop contains:

- an idNumber,
- a written description,
- the purchase date,
- the cost,
- and a history (including failure rate, number of days down, and date of last service).

```
struct  DateType{
 int    month ;                 // Assume 1 . . 12
 int    day ;                   // Assume  1 . . 31
 int    year ;                  // Assume 1900 . . 2050
};
struct  StatisticsType{
 float                failRate ;
 DateType             lastServiced ; // DateType is a struct type
 int                  downDays ;
} ;
struct MachineRec{
 int                      idNumber ;
 string       description ;
 StatisticsType   history ;        // StatisticsType is a struct type
 DateType                 purchaseDate ;
 float                    cost ;
} ;
MachineRec     machine ;
```

struct type variable `machine`



LECTURE 7

Struct Member Function (closure)

Member Functions

C++ also allows function declarations (called member functions) inside structures.

```
struct struct_name {
 Type member_function1(signature);
 Type member_function2(signature);
};
```

- Typically, member function names are distinct, but the name of a member function may be the same as another if their signatures differ.
- There is no size overhead for member functions inside structures (try sizeof() to convince yourself).

Member Functions

- Member functions are essential in object-oriented programming because new data types combine functionality (member functions) with data (data members), all in a single unit.
- This concept lets you design objects with an implementation (how objects work) and an interface (how objects behave). We explore these important concepts in Chapter 4 with class definitions.
- The dot (.) operator provides access to structure members.

```
struct X {
 int num;          // data member
 int f();          // member function
};

X a = { 12 };         // initialize data member
cout << a.num << endl;      // data member, displays 12
cout << a.f() << endl;      // calls member function
```

Structure Pointers

Pointers may address structures. The formats are

```
struct struct_name {
 Type data_member;
 Type member_function(signature);
} *pname = { init_list };
```

```
struct struct_name *pname = { init_list };
```

- The brace-enclosed init_list is optional and must contain a pointer expression whose type matches or converts to struct_name. If you initialize structure pointers, the braces surrounding init_list are not necessary.
- The word struct_name is optional in the first format, and the keyword struct is optional in the second format when you define struct_name elsewhere.

Structure Pointers

- The brace-enclosed init_list is optional and must contain a pointer expression whose type matches or converts to struct_name. If you initialize structure pointers, the braces surrounding init_list are not necessary.
- The word struct_name is optional in the first format, and the keyword struct is optional in the second format when you define struct_name elsewhere.

Here are the two formats to access structure members.

```
struct_name_variable.member_name   // structure

struct_name_pointer->member_name   // structure pointer
```

Demo Program:

stPointer.cpp

Go Dev C++!!!

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

 struct block {            // structure type
  int buf[80];           // data member
  char *pheap;           // data member
  void header(const char *);    // member function
 };

 block data = { {1,2,3}, "basic" };  // structure variable
 block *ps = &data;          // structure pointer

void block::header(const char *st){ // definition of member function for struct block
    cout << st << endl;
}

int main(int argc, char** argv) {

     data.pheap++;            // increment data member
     cout << data.pheap << endl;
     data.header("magic");        // call member function
     ps->pheap++;             // increment data member
     cout << data.pheap << endl;
     ps->header("magic");         // call member function
     //ps.pheap++;             // illegal, ps is a pointer
    //data->header("magic");        // illegal, data is a structure
     return 0;
}


```

```
C:\Eric_Chou\Cpp Course\C++ Programming Essentials\CppDev\ch10\struct pointer\stPointer.exe
asic
magic
sic
magic

--------------------------------
Process exited after 0.007653 seconds with return value 0
Press any key to continue . . .
```

LECTURE 8

Union Data Type

Unions in C++

DEFINITION

A union is a struct that holds only one of its members at a time during program execution.

EXAMPLE

```
union   WeightType{
 long    wtInOunces ;
 int     wtInPounds;
 float   wtInTons;
} ;
```

only one at a time

Using Unions

```
union   WeightType{                    // declares a union type
 long    wtInOunces ;
 int      wtInPounds;
 float wtInTons;
 } ;

WeightType   weight;                   // declares a union variable
weight.wtInTons = 4.83 ;

//  Weight in tons is no longer needed. Reuse the memory space.
weight.wtInPounds = 35;
```

Syntax for Union

Data has Different Interpretations

Syntax

```
union union-name
{
    public-members-list;
private:
    private-members-list;
} object-list;
```

- Union is similar to struct (more that class), unions differ in the aspect that the fields of a union share the same position in memory and are by default public rather than private.
- The size of the union is the size of its largest field (or larger if alignment so requires, for example on a SPARC machine a union contains a double and a char so its size is likely to be 24 because it needs 64-bit alignment).
- Unions cannot have a destructor.

Demo Program:

employee.cpp

Go Dev C++!!!

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

union Employee{
    int Id;
    char Name[25];
    int Age;
    long Salary;
};

int main(int argc, char *argv[]){
    Employee E;

    cout << "\nEnter Employee Id : ";
    cin >> E.Id;

    cout << "\nEnter Employee Name : ";
    scanf("%s", &E.Name);
    string empty;  // consume the /n mark.
    getline(cin, empty);

    cout << "\nEnter Employee Age : ";
    cin >> E.Age;

    cout << "\nEnter Employee Salary : ";
    cin >> E.Salary;

    cout << "\n\nEmployee Id : " << E.Id;
    cout << "\nEmployee Name : " << E.Name;
    cout << "\nEmployee Age : " << E.Age;
    cout << "\nEmployee Salary : " << E.Salary;
    return 0;
}
```

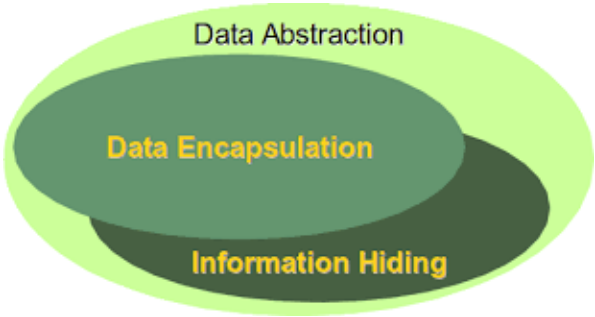
```
C:\Eric_Chou\Cpp Course\C++ Programming Essentials\CppDev\ch10\employe

Enter Employee Id : 90210

Enter Employee Name : Eric Chou

Enter Employee Age : 50

Enter Employee Salary : 1000000


Employee Id : 1000000
Employee Name : @B
Employee Age : 1000000
Employee Salary : 1000000
--------------------------------
Process exited after 21.37 seconds with return value 0
Press any key to continue . . .
```

LECTURE 9

Abstraction

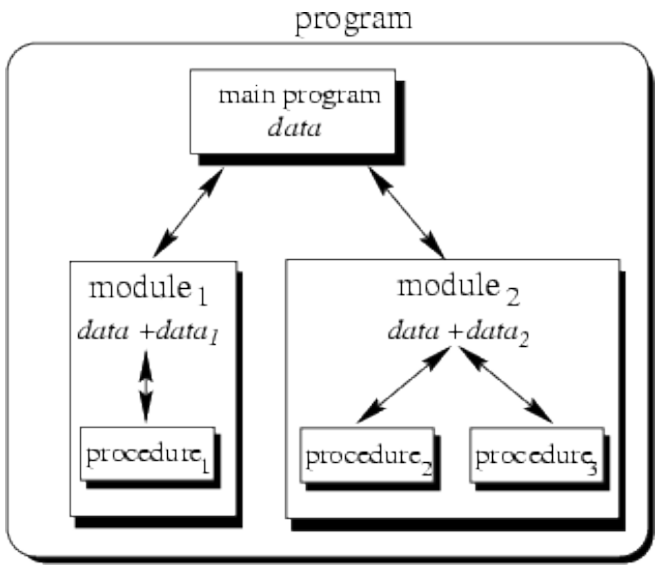
Abstraction

- 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
- is necessary for managing large, complex software projects

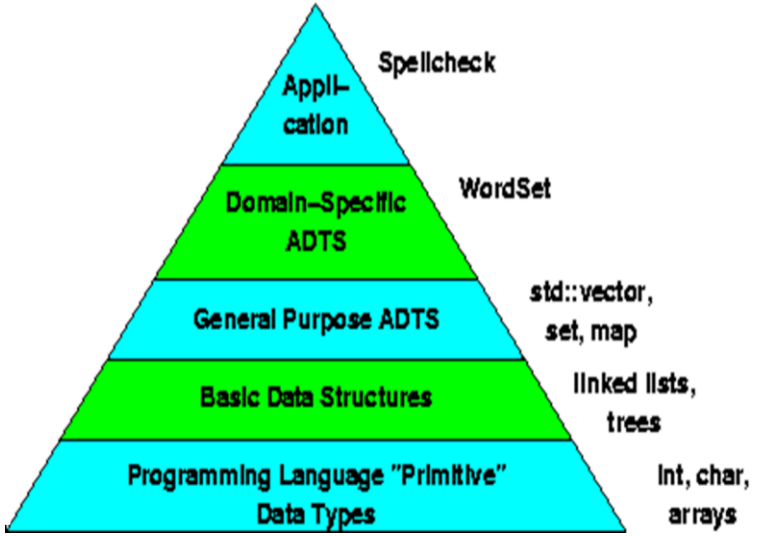


Abstraction

Abstraction (from the Latin ***abs***, meaning ***away*** from and ***trahere***, meaning to ***draw***) is the process of taking away or removing characteristics from something in order to reduce it to a set of essential characteristics.



Data Abstraction in C Language



Control Abstraction

- separates the logical properties of an action from its implementation

```
.
.
.
Search (list, item, length, where, found);
.
.
.
```

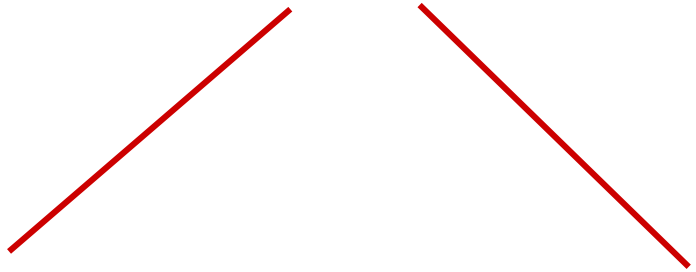
- the function call depends on the function's specification (description), not its implementation (algorithm)

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 Type



set of values (domain)

allowable operations on those values

FOR EXAMPLE, data type `int` has

domain	operations
-32768 . . . 32767	+, -, *, /, %, >>, <<

Abstract Data Type (ADT)

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

FOR EXAMPLE . . .

ADT Specification Example

TYPE

TimeType

DOMAIN

Each TimeType 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

ComplexNumberType

DOMAIN

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

OPERATIONS

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

ADT Implementation means

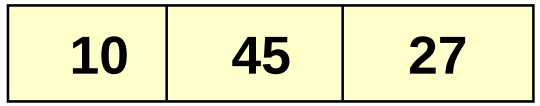
- choosing a specific data representation for the abstract data using data types that already exist (built-in or programmer-defined)
- writing functions for each allowable operation

Several Possible Representations of `TimeType`

3 int variables

3 strings

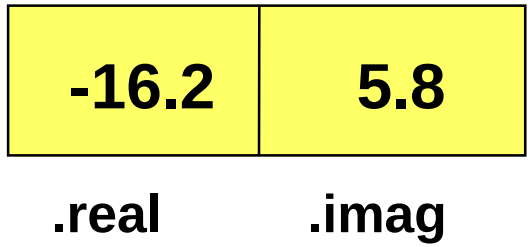
3-element int array



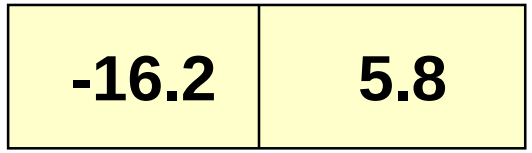
- actual choice of representation depends on time, space, and algorithms needed to implement operations

Some Possible Representations of `ComplexNumberType`

struct with 2 float members



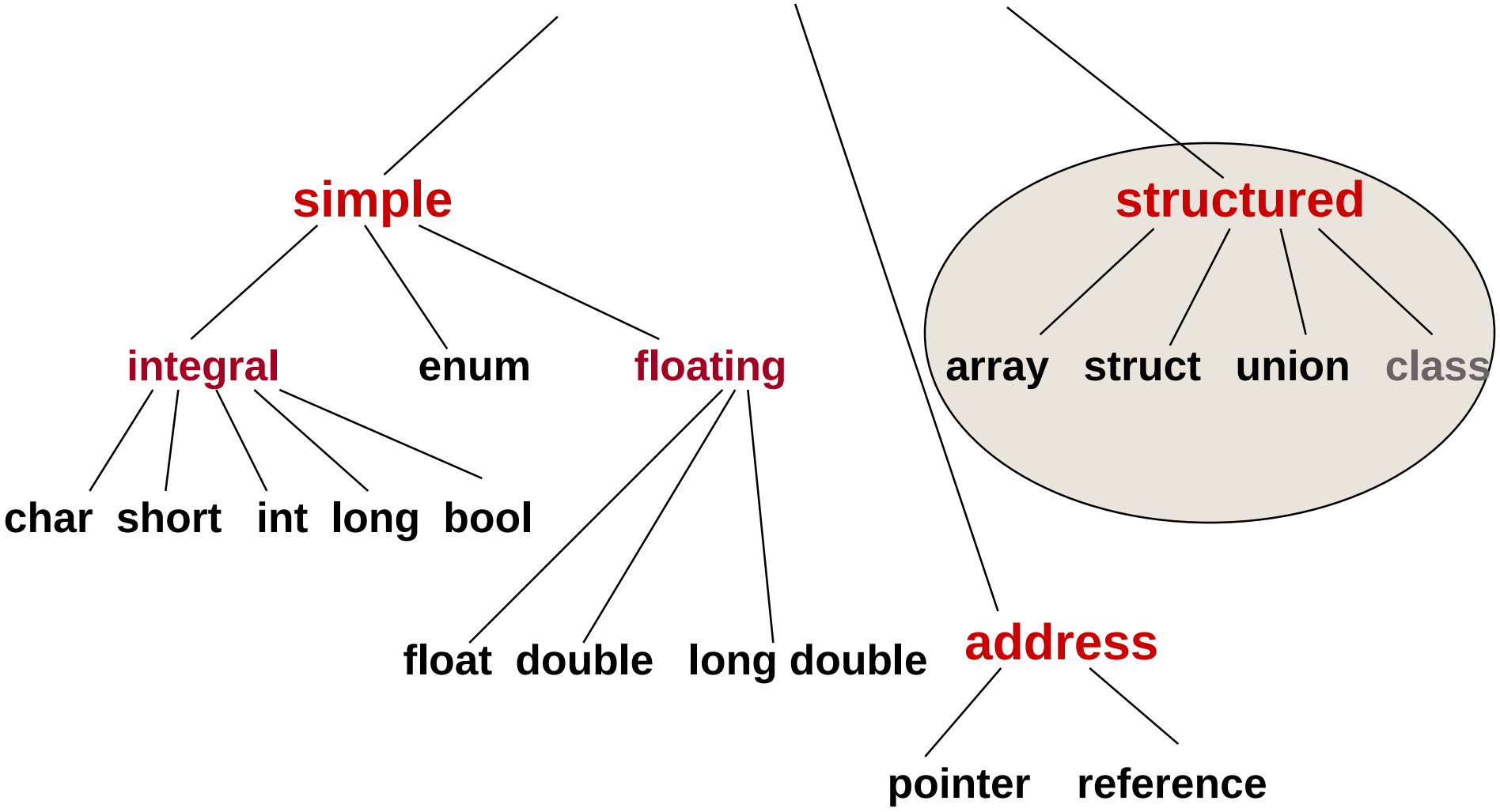
2-element float array



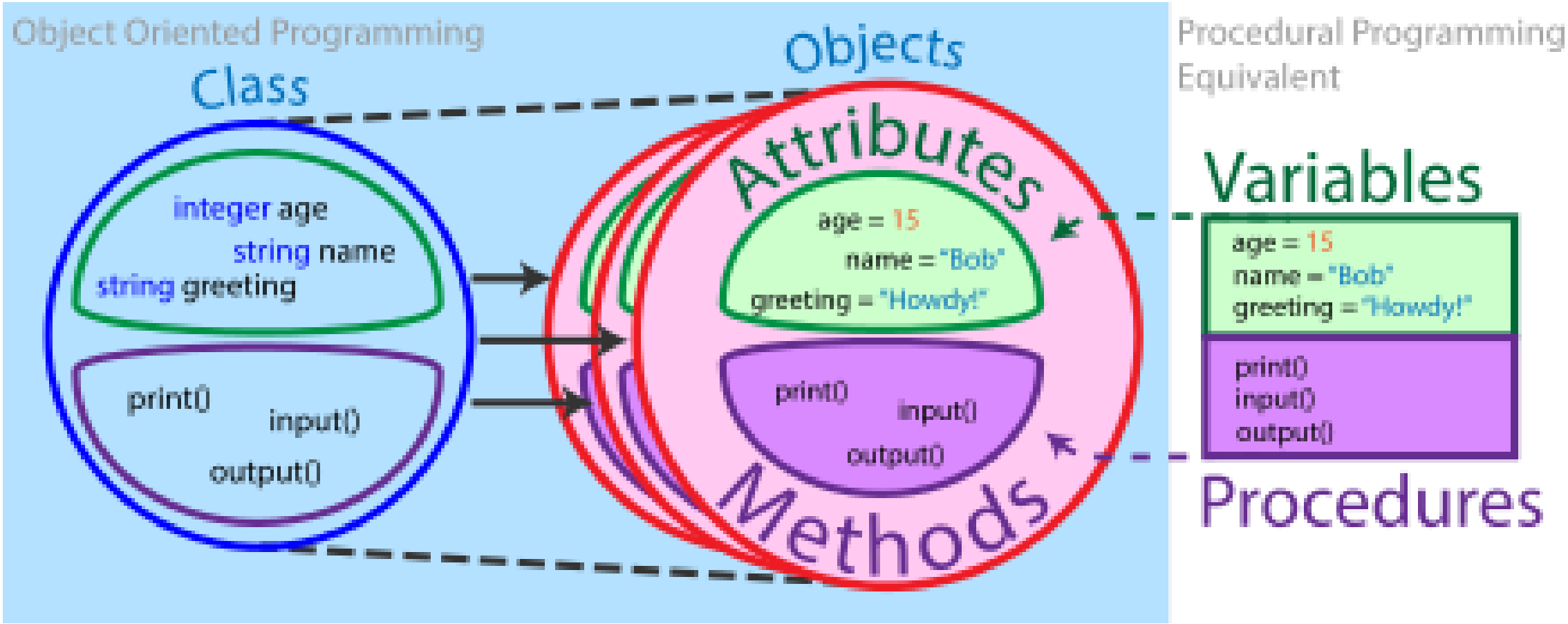
LECTURE 10

Class Data Type

C++ Data Types


simple
integral
enum
floating
char
short
int
long
bool
float
double
long double
structured
array
struct
union
class
address
pointer
reference

Object-Oriented Programming



Class is a Template to Build Objects

- Class has more mechanism to manage member data field and member functions than struct.
- Class has mechanism to handle each object and class as a whole.
- Class has mechanism to handle relationships among classes.
- Class is used to defined the three major features of Object-Oriented Programming: Encapsulation, Inheritance and Polymorphism.

- class Name
- {
- public:
  - » constructor(s)
  - » destructor
  - » function members
  - » data members
- protected:
  - » function members
  - » data members
- private:
  - » function members
  - » data members
- };

```
class MyClass
{
public:
        MyClass() { myValue = 10; }
        int getMyValue () { return myValue;}
        int setMyValue (int aValue) { myValue = aValue;}
private:
        int myValue;
};

int main(void)
{
        MyClass inst0, inst1;
        inst1.setMyValue(20);
        cout << inst0.getMyValue() << endl
             << inst1.getMyValue() << endl;
        return 0;
}
```

// In C++, main() is not a member function of any class.
// In Java, any class can have a main() function

class TimeType Specification

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

class  TimeType                           // declares a  class data type
{                                                  //  does not allocate memory

public :                                  //  5 public function members

 void       Set ( int  hours , int  mins , int  secs ) ;
 void       Increment ( ) ;
 void       Write ( )  const ;
 bool       Equal ( TimeType   otherTime )  const ;
 bool       LessThan ( TimeType   otherTime )  const ;

private :                                 //  3 private data members

 int        hrs ;
 int        mins ;
 int        secs ;
} ;
```

Use of C++ data Type **class**

- 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 public member functions to handle its class objects

Client Code Using `TimeType`

```
#include  "timetype.h"      // includes specification of the class
using  namespace  std ;

int  main ( )
{
  TimeType   currentTime ;          // declares 2 objects of TimeType
  TimeType   endTime ;
  bool         done  =  false ;

  currentTime.Set ( 5, 30, 0 ) ;
  endTime.Set ( 18, 30, 0 ) ;
  while  ( ! done )
    {      . . .

           currentTime.Increment ( ) ;
           if  ( currentTime.Equal ( endTime ) )
                done  =  true ;
     } ;
}
```

LECTURE 11

Class Declaration

class type Declaration

- The class declaration creates a data type and names the members of the class.
- It does not allocate memory for any variables of that type!
- Client code still needs to declare class variables.

C++ Data Type **class** represents an ADT

- 2 kinds of class members:
  data members and function members
- class members are **private** by default
- 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.

Public and Private Member

Visibility [.h file]

```
class list_node {
    list_node* prev;
    list_node* next;
    list_node* head_node;
public:
    int val;
    list_node();
    list_node* predecessor();
    list_node* successor();
    bool singleton();
    void insert_before(list_node* new_node);
    void remove();
    ~list_node();
};
```

Private Members

Public Members (Header for Member Methods)

Aggregate **class** Operations

1. built-in operations valid on class objects are:
   - member selection using dot (.) operator ,
   - assignment to another class variable using (=),
2. pass to a function as argument (by value or by reference),
3. return as value of a function
4. other operations can be defined as class member functions

2 separate files Generally Used for **class** Type

```
//   SPECIFICATION FILE                ( timetype .h )
//   Specifies the data and function members.
class TimeType{
   public:

         . . .

   private:
       . . .
  } ;



//   IMPLEMENTATION FILE                ( timetype.cpp )
//   Implements the TimeType member functions.

      . . .
```

Implementation File for `TimeType`

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

//   Implements the TimeType member functions.

#include  " timetype.h"   // also must appear in client code
#include  <iostream>
 . . .

bool  TimeType :: Equal ( /* in */  TimeType otherTime ) const
//  Postcondition:
//        Function value == true,    if this time equals otherTime
//                                   == false , otherwise
{
    return ( (hrs == otherTime.hrs)  && (mins == otherTime.mins)
                                          && (secs  == otherTime.secs) ) ;
 }

     . . .
```

Separate Method Definition

using a :: **scope** resolution operator [.cc file .cpp file]

```
void list_node::insert_before(list_node* new_node) {
    if (!new_node->singleton())
        throw new list_err("attempt to insert node already on list");
    prev->next = new_node;
    new_node->prev = prev;
    new_node->next = this;
    prev = new_node;
    new_node->head_node = head_node;
}
```

Familiar Class Instances and Function Members

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

```
cin.get (someChar) ;

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

- these statements declare myInfile as an instance of class **ifstream** and invoke function member open

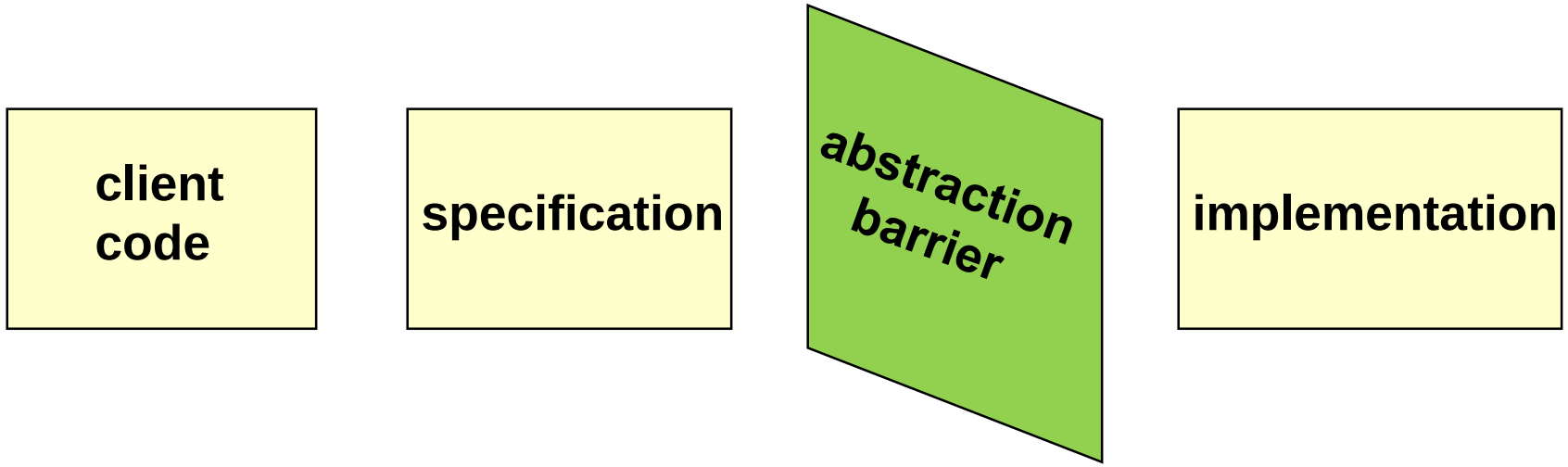
```
ifstream  myInfile ;

myInfile.open ( "A:\\mydata.dat" ) ;
```

LECTURE 12

Data Encapsulation (Information Hiding)

Information Hiding

- Class implementation details are hidden from the client's view. This is called information hiding.
- Public functions of a class provide the interface between the client code and the class objects.



Scope Resolution Operator (::)

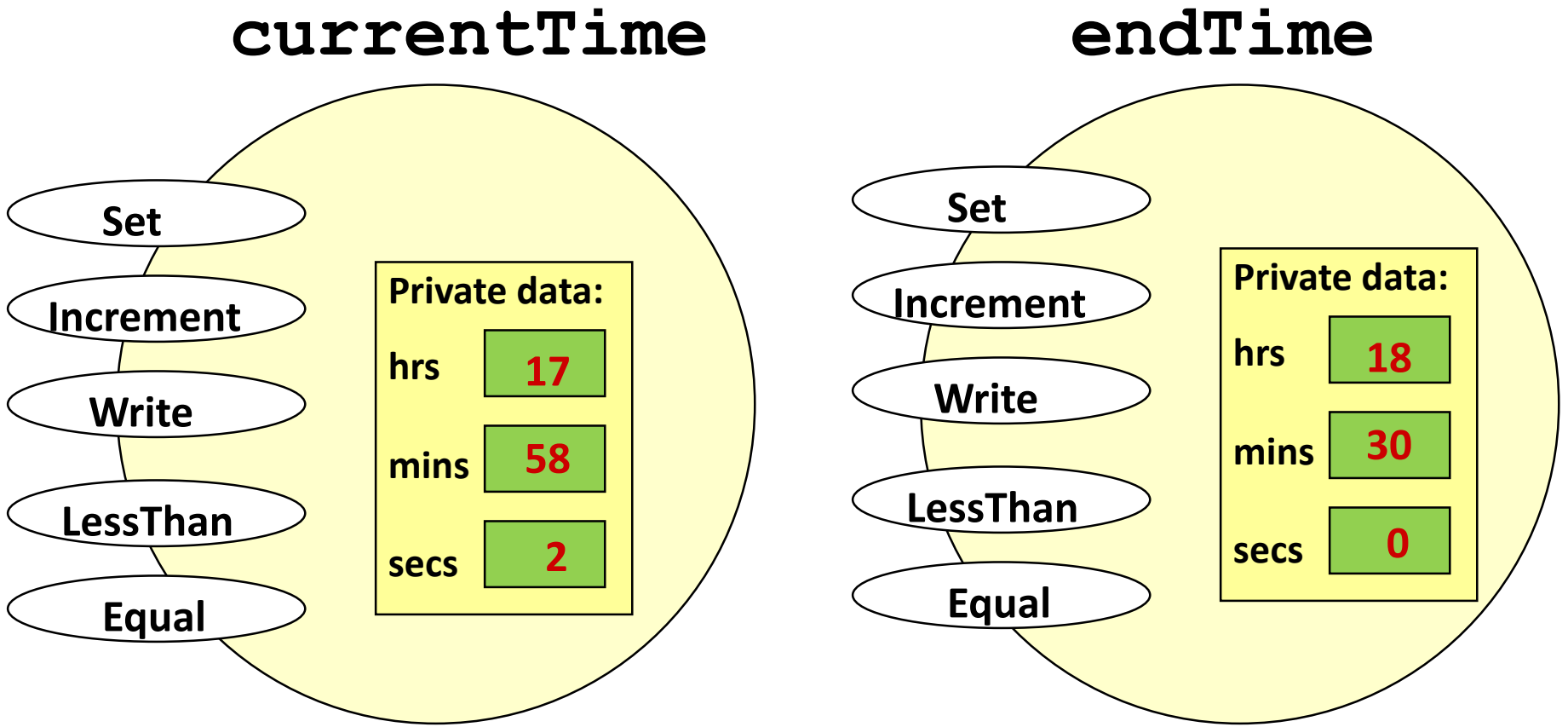
- C++ programs typically use several class types
- different classes can have member functions with the same identifier, like Write( )
- member selection operator is used to determine the class whose member function Write( ) is invoked

```
currentTime .Write( ) ;          // class TimeType
numberZ .Write( ) ;              // class ComplexNumberType
```

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

```
void  TimeType :: Write ( )  const  {      . . .      }
```

TimeType Class Instance Diagrams



Use of `const` with Member Functions

- 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)
- Then, no data should be updated.

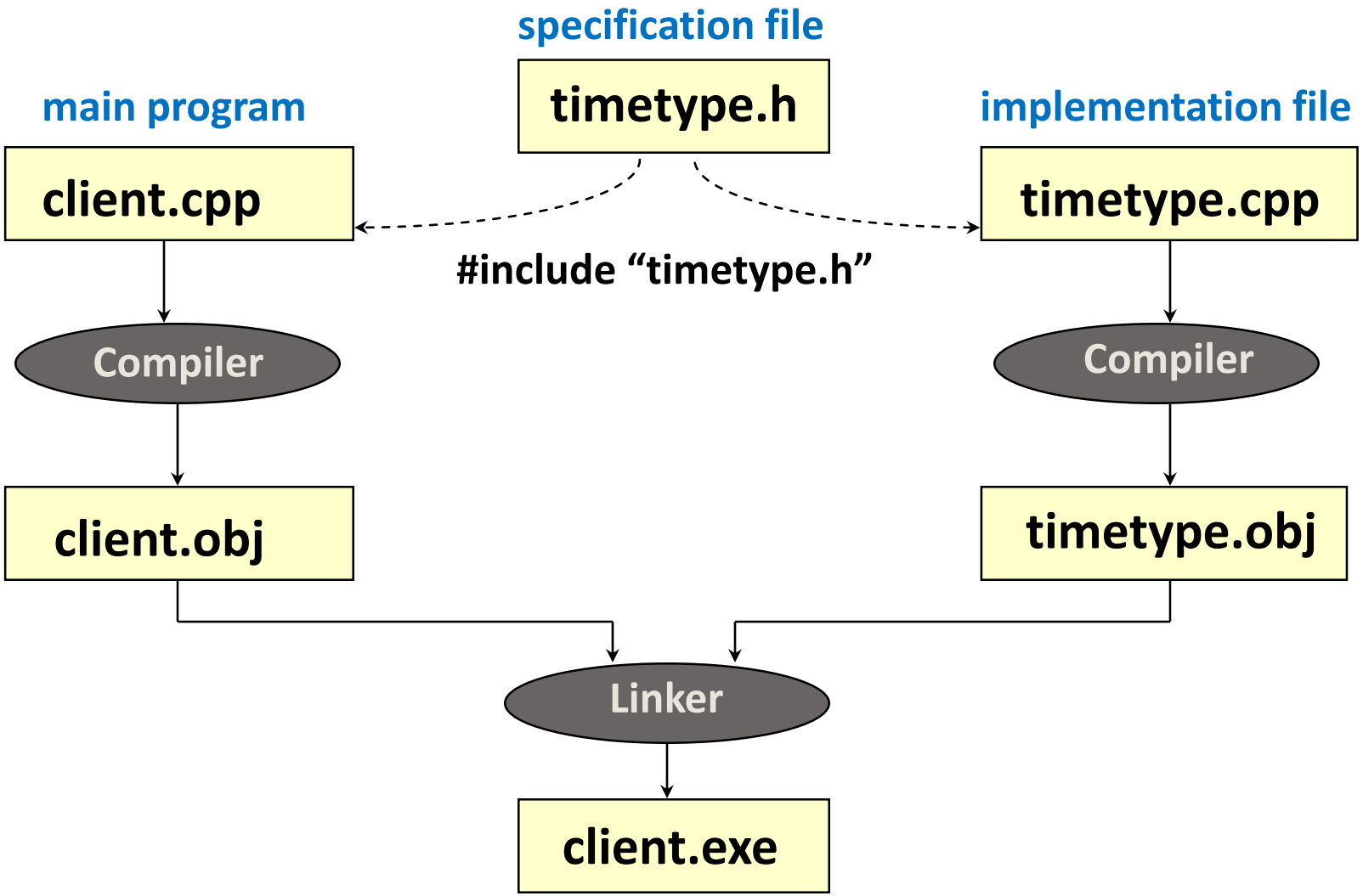
Example Using `const` with a Member Function

```
void   TimeType :: Write ( )   const
  //   Postcondition:   Time has been output in form HH:MM:SS
{          if  ( hrs < 10 )
                cout << '0' ;
           cout  << hrs  <<  ':' ;
           if  ( mins < 10 )
                cout << '0' ;
           cout  << mins  <<  ':' ;
           if  ( secs < 10 )
                cout << '0' ;
           cout  << secs ;
}
```

LECTURE 13

Separate Compilation

Separate Compilation and Linking of Files



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
- 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`

```
// timetype .h
// SPECIFICATION FILE
#ifndef   TIME_H
#define  TIME_H

class TimeType
{
 public:
  . . .

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

FOR COMPILATION THE CLASS DECLARATION IN

FILE timetype.h WILL BE INCLUDED ONLY ONCE

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

```
// client.cpp
// Appointment program
#include "timetype.h"
int   main ( void )
{
       . . .
}
```

LECTURE 14

class Constructor

Class Constructors

- a class constructor is a function, for an object, whose purpose is to initialize the 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
- 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 `TimeType` Class Constructors

```
class  TimeType                                    // timetype.h
{
public :                                           //  7 function members
 void          Set ( int  hours ,  int  minutes ,  int   seconds ) ;
 void          Increment ( ) ;
 void          Write ( )  const ;
 bool       Equal ( TimeType   otherTime )  const ;
 bool       LessThan ( TimeType  otherTime )  const ;

   TimeType ( int  initHrs ,  int  initMins ,  int  initSecs ) ;  // constructor

   TimeType ( ) ;                                    // default constructor

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

Implementation of `TimeType` Default Constructor

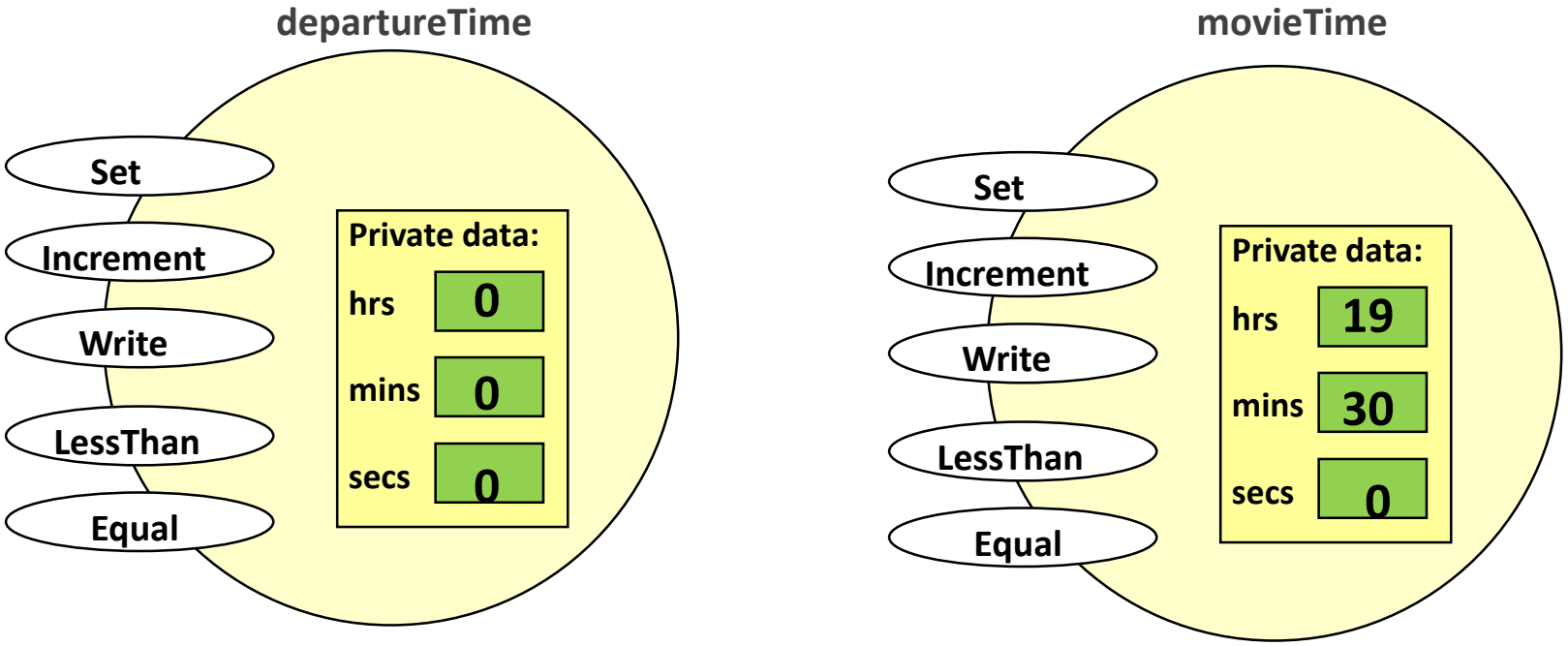
```
TimeType :: TimeType ( ) // Constructor has the same name as class
// Default Constructor
// Postcondition:
//                          hrs == 0  &&  mins == 0  &&  secs == 0
{
   hrs  =  0 ;
   mins = 0 ;
   secs = 0 ;
}
```

Implementation of Another `TimeType` Class Constructor

```
TimeType :: TimeType ( /* in */   int   initHrs,
                       /* in */   int   initMins,
                       /* in */   int   initSecs )
// Constructor
// Precondition:  0 <= initHrs <= 23    &&    0 <= initMins <= 59
//                                          0 <= initSecs <= 59
// Postcondition:
//                        hrs == initHrs  &&  mins == initMins  && secs == initSecs
{
   hrs  =  initHrs ;
     mins =  initMins ;
     secs =  initSecs ;
}
```

Automatic invocation of constructors occurs

```
TimeType   departureTime ;          // default constructor invoked
TimeType   movieTime (19, 30, 0 ) ;  // parameterized constructor
```



Overloaded Constructors

Constructors Taking Different Parameters

We have overloaded the constructor in **int_list_node**, providing two alternative implementations. One takes an argument, the other does not. Now the programmer can create **int_list_node**s with or without specifying an initial value:

```
int_list_node element1;                    // val = 0
int_list_node *e_ptr = new int_list_node(13);    // val = 13
```

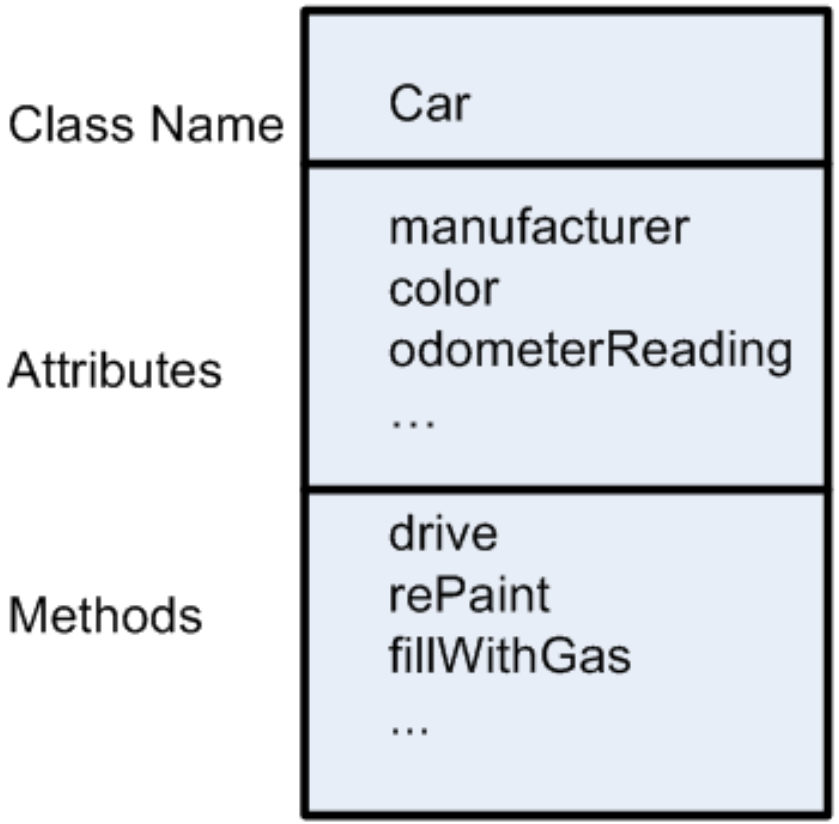
In C++, the compiler ensures that constructors for base classes are executed before those of derived classes. In our example, the constructor for **gp_list_node** will be executed first, followed by the constructor for **int_list_node**.

Note: default constructors

LECTURE 15

Instantiation (Constructor)

Class



Members (Inherited)

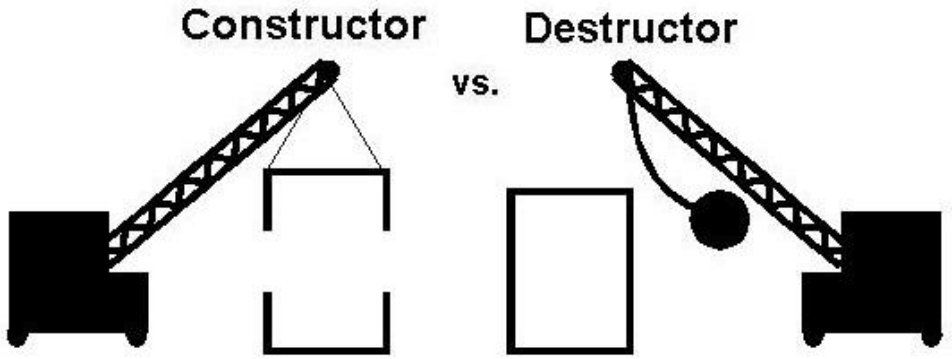
Data Members
Data Fields

Member Methods
Methods
Subroutines

Non-Members (Not Inherited)

Constructors
A::A(){}

Destructors
A::~A(){}



Declarations and Constructors in C++

Declarations and Constructors in C++:

```
foo   b;       // calls foo::foo()
```

If a C++ variable of class type foo is declared with no initial value, then the compiler will call foo's zero-argument constructor.

Declarations and Constructors with parameters in C++:

```
foo   b(10, 'x');       // calls foo::foo(int, char)
foo   c{10, 'x'};       // alternative syntax in C++11
```

Declarations and Constructors with Reference in C++:

```
foo   a;
…
foo   b(a);    // calls  foo::foo(foo&)
foo  c{a} ;    // alternative syntax
```

Copying of Objects:

```
foo   a;       // calls foo::foo()
…
foo   b = a;  // calls  foo::foo(foo&)
```

In recognition of this intent, a single-argument constructor in C++ is called a **copy constructor**. It is important to realize here that the equals sign (=) in these declarations indicates initialization, not assignment. The effect is not the same as that of the similar code fragment.

Assignment:

```
foo   a, b;       // calls foo::foo() twice
…
b = a;       // calls  foo::operator=(foo&)
```

This is assignment not initialization.

C++ 98 Constructors

```
int i ;                          // Uninitialised built-in type
int j = 10;                      // Initialised built-in type
int k(10);                       // Initialised built-in type

int array[] = {1, 2, 3 };        // Aggregate initialisation
char str[] = "Hello";            // String literal initialisation

X x1;                            // Default constructor
X x2(10.7);                      // Non-default constructor
X x3 = x2;                       // Copy constructor
X x4 = 10.7;                     // Copy-constructor elision
```

C++98 has a frustratingly large number of ways of initialising an object.
(Note: not all these initialisations may be valid at the same time, or at all.
We're interested in the syntax here, not the semantics of the class X)

C++ 11 Constructors

```
int i { };                  // Default initialised built-in type
int j { 10 };               // Initialised built-in type

X x1 { };                   // Default constructor
X x2 { 10.7 };              // Non-default constructor
X x3 { x2 };                // Copy constructor
X x4 = { 1, 3, 5, 7 };      // Construct as aggregrate type
```

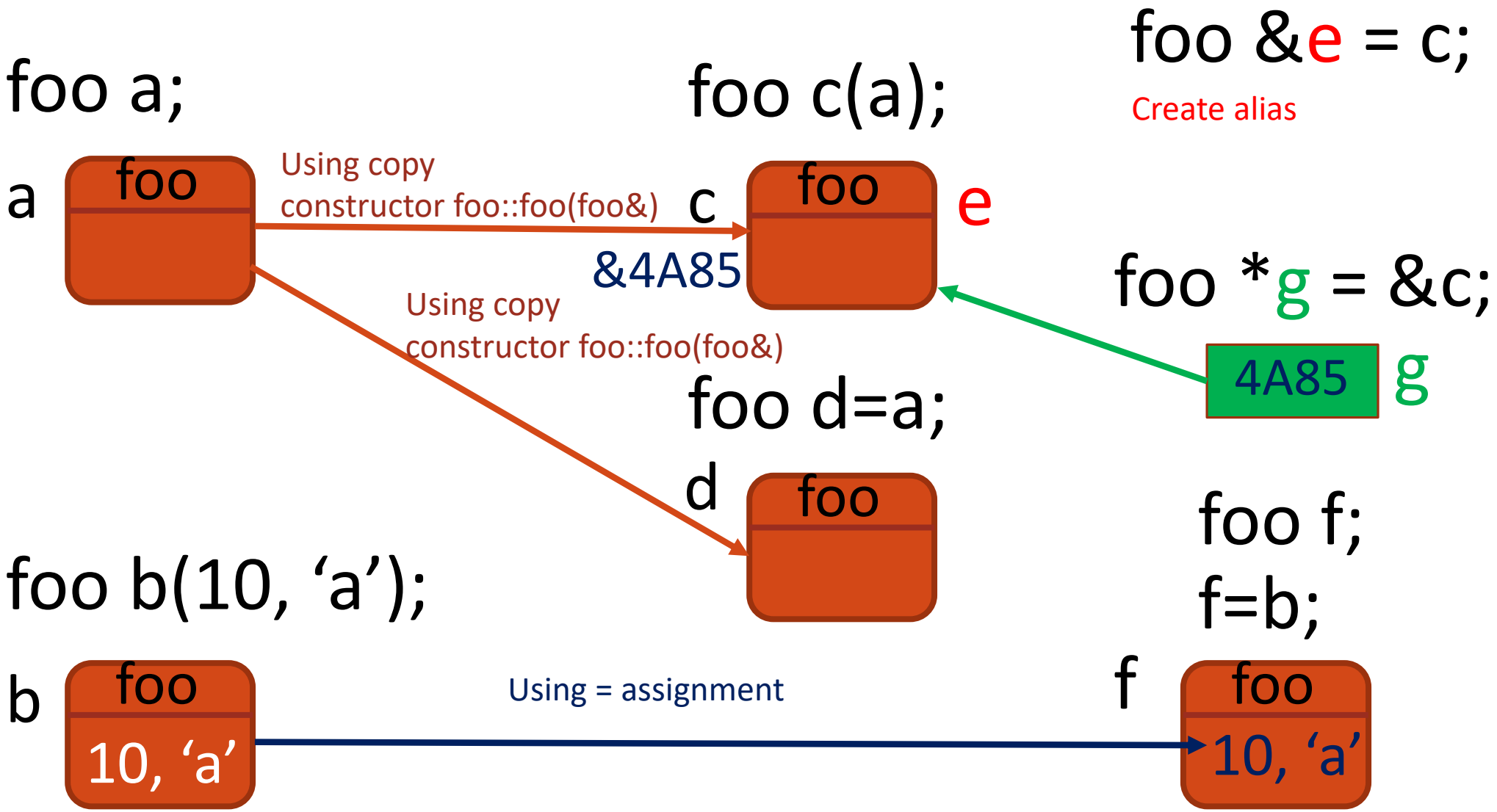
Note:

1. One of the design goals in C++11 was uniform initialisation syntax. That is, wherever possible, to use a consistent syntax for initialising any object. The aim was to make the language more consistent, therefore easier to learn (for beginners), and leading to less time spent debugging.
2. To that end they added brace-initialisation to the language.
3. As the name would suggest, brace-initialisation uses braces ({}) to enclose initialiser values.

Types of C++ Constructors

Default Constructor:	foo a;	// call foo::foo();
Constructor with Parameter:	foo b(10, ‘a’);	// call foo::foo(int, char);
	foo c{10, ‘c’};	// call foo::foo(int, char); C++ 11
Constructor with Reference:	foo b(a);	// foo a; was defined, call foo::foo(foo&)
	foo c{a};	// C++ 11 version
Copy Constructor:	foo b=a;	// foo a; was defined, call foo::foo(foo&)

Assignment: foo a, b; b = a; // This is assignment. It is not using a constructor.


foo a;
a
foo
Using copy
constructor foo::foo(foo&)
foo c(a);
c
&4A85
foo
e
foo &e = c;
Create alias
foo *g = &c;
4A85
g
Using copy
constructor foo::foo(foo&)
foo d=a;
d
foo
foo b(10, 'a');
b
foo
10, 'a'
Using = assignment
foo f;
f=b;
f
foo
10, 'a'

```
class Circle //specify a class
{
    private :
        double radius; //class data members
    public:
        Circle()          //default constructor
        {
            radius = 0;
        }
        Circle(double r)  //parameterized constructor
        {
            radius = r;
        }
        Circle(Circle &t) //copy constructor
        {
            radius = t.radius;
        }
        void setRadius(double r) //function to set data
        {
            radius = r;
        }
        double getArea()
        {
            return 3.14 * radius * radius;
        }
        ~Circle() //destructor
        {}
};
```

Declarations and Constructors in C++

Temporary Objects

In **C++**, the requirement that every object be constructed (and likewise destructed) applies not only to objects with names but also to temporary objects. The following, for example, entails a call to both the **string(const char*)** constructor ad the **~string()** destructor:

```
cout<< string("Hi, Mom").length();
```

The destructor called at the end of the output statement: the temporary objects behaves as if its sop were just the line shown here.

The following entails not only two calls to the default string constructor and a call to string::operator(), but also constructor call to initialize the temporary object returned by operator() – the object whose length is then queried by the caller:

```
string a, b;
…
(a+b).length();
```

Return Value Optimization

- **f** is a function returning a value of class type **foo**.
- If instance of **foo** are too big to fit in a register, the compiler will arrange for **f**'s caller to pass an extra, hidden parameter that specifies the locations into which **f** should construct the return value. **(a mail box for return value)**
- If the return statement itself creates a temporary object -

```
return foo(args)
```

- **f**'s source looks more like this:

```
foo rtn;
…
return rtn;
```

Note: This option is known as Return value optimization.

Return Value Optimization

- In other programs the compiler may need to invoke a **copy constructor** after a function returns:

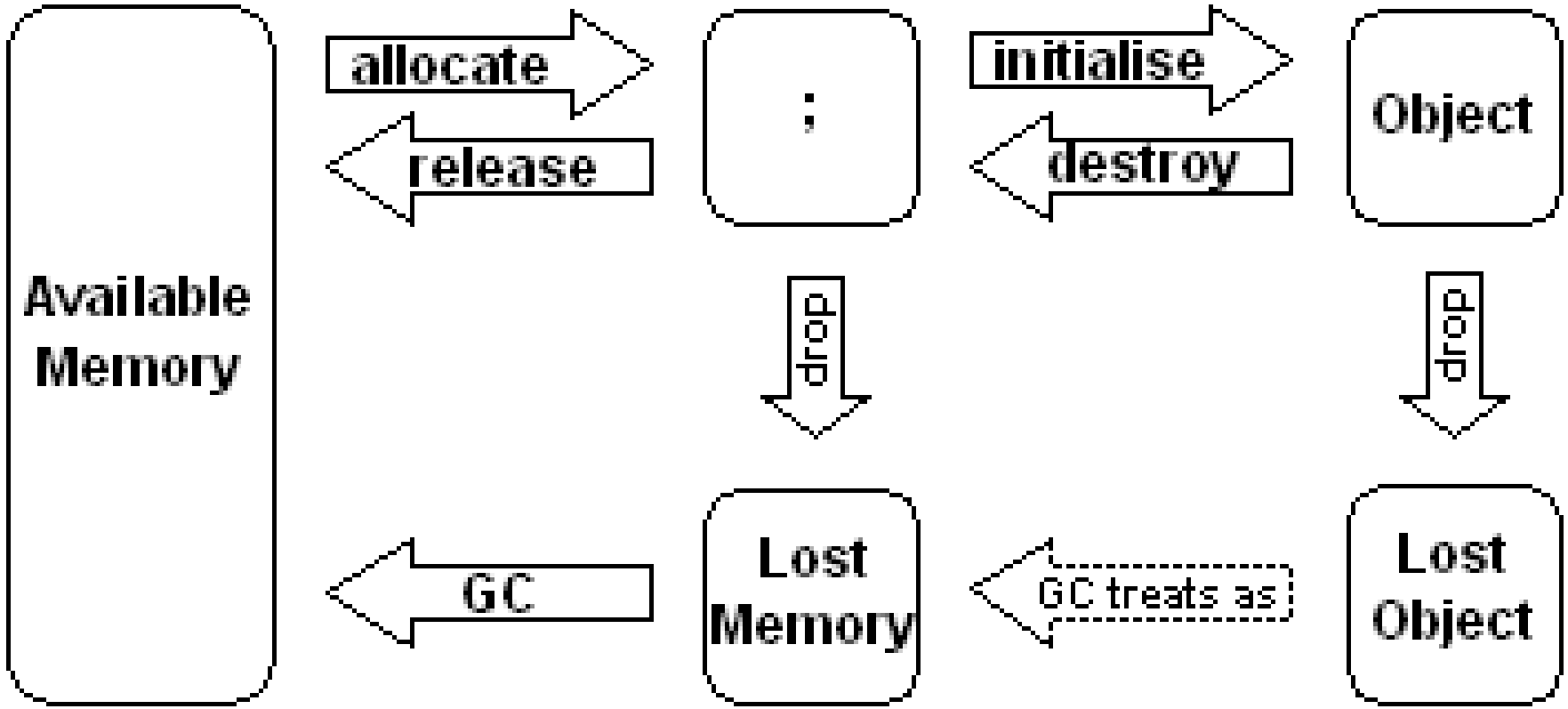
```
foo c;
…
c = foo(args);
```

- The location of **c** cannot be passed to the hidden parameter to unless the compiler is able to prove that **c**'s value will not be used during the call.
- **The bottom line:** returning an object from a function in C++ may entail zero, one or two invocations of return type's copy. Constructor, depending on whether the compiler is able to optimize either or both of the return statement and the subsequent use in the caller.

Three Types of Object Life-Cycle

- C
  - alloc() – use – free()
- C++
  - new() – constructor() – use – destructor()
- Java
  - new() – constructor() – use – [ignore / garbage collection]

Java Object Life Cycle



Initialization and Finalization

1. Choosing a constructor (of the right signature)
2. References and values
   - If variables are references, then every object must be created explicitly - appropriate constructor is called
   - If variables are values, then object creation can happen implicitly as a result of elaboration
3. Execution order
   - When an object of a derived class is created in C++, the constructors for any base classes will be executed before the constructor for the derived class
4. Garbage collection

LECTURE 16

Execution Order and Finalization(Destructor)

Execution Order

Super Class Before this class

- C++ insists that every object be initialized before it can be used.
- If the object's class (call it B) is derived from some other class (call it A), C++ insists on calling an A constructor before calling a B constructor, so that the derived class is guaranteed never to see its inherited fields in an inconsistent state. [Super Class Constructor First]

```
foo::foo( foo_params ) : bar( bar_args ) {
    ...
```

Super class

- **bar(bar_args)** is run first with its own parameter. Then the foo(foo_params) constructor.

Execution Order

Member Constructors

C++ Allow Parameterized Member Objects: (Example for Passing Simple Value)

```
list_node() : prev(this), next(this), head_node(this), val(0) {
    // empty body -- nothing else to do
}
```

C++ Allow Parameterized Member Objects: (Example for Parameterized Constructors)

```
class foo : bar {
    mem1_t member1;     // mem1_t and
    mem2_t member2;     // mem2_t are classes
    ...
}

foo::foo( foo_params ) : bar( bar_args ), member1( mem1_args ),
        member2( mem2_args ) {
    ...
```

Note: The compiler call the copy constructors for member objects, rather than calling the default constructors, followed by operator= within the body of the constructor. Both semantics and performance may be different as a result.

Execution Order

Constructor Forwarding (call a() constructor is calling a(1) constructor)

Constructor Forwarding:

```
class list_node{
    …
    list_node() : list_node(0) {}
```

- In Java, if A extends B

```
class B { int i; } // int = 0, float = 0.0, boolan = false, ref = null
class A { }
```

- Without explicit declaration, both A, B class have their own A() and B() constructor as default constructor.
- Without explicit declaration, A() will call super() which is B().
- If you want to explicitly define the A() constructor, you must write:

```
A(){
    super(args);  // B(args), this statement must be run before
                  // any other statements.
}
```

Garbage Collection

Reclaiming Space with Destructors

C++: (Using destructor)

- destructor for the derived class is called first, followed by those of the base class(es), in reverse order of derivation.

```
~queue(){
   while (!empty()){
      list_nod* p = contents.head();
      p-> remove();
      delete p;
   }
}
```

- Since **dequeue()** has already been designed to delete the node that contained the dequeued element:

```
~queue(){
   while( !empty()){
         int v = dequeue();
    }
}
```

Java: (parking to null)

Return memory by explicit assignment:

```
A a = new A();
….
a = null;  // a's original object will be dangling
           // Then, the object body will be re-claimed
```

C: (no garbage collection, C uses a volunteering recycling)

- Free()

The C library function **void free(void *ptr)** deallocates the memory previously allocated by a call to calloc, malloc, or realloc.

Demo Program: objects.cpp

Go Dev C++!!!

```
int main()
{
    Circle c1; //defalut constructor invoked
    Circle c2(2.5); //parmeterized constructor invoked
    Circle c3(c2); //copy constructor invoked
    cout << c1.getArea()<<endl;
    cout << c2.getArea()<<endl;
    cout << c3.getArea()<<endl;
    return 0;
}
```

LECTURE 17

Visibility of C++

Visibility in C++

```
class Base {
    public: // public members go here
    protected: // protected members go here
    private: // private members go here
};
```

- A **public** member is accessible from anywhere outside the class but within a program.
- A **protected** member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.
- A default (no specifier) member is accessible from anywhere in the same package.
- A **private** member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.

C++

- C++ distinguishes among
  - public class members
    - accessible to anybody
  - protected class members
    - accessible to members of this or derived classes
  - private
    - accessible just to members of this class
- A C++ structure (*struct*) is simply a class whose members are **public** by default
- C++ base classes can also be public, private, or protected

C++Inheritance and Visibility

- Example:

```
class circle : public shape { ...
```

anybody can convert (assign) a circle* into a shape*

```
class circle : protected shape { ...
```

only members and friends of circle or its derived classes can convert (assign) a circle* into a shape*

```
class circle : private shape { ...
```

only members and friends of circle can convert (assign) a circle* into a shape*

Class

Visibility

- With the introduction of inheritance, object-oriented languages must supplement the scope rules of module-based languages to cover additional issues.
- For example,
  - should private members of a base class be visible to methods of a derived class?
  - Should public members of a base class always be public members of a derived class (i.e., be visible to users of the derived class)?
  - How much control should a base class exercise over the visibility of its members in derived classes?

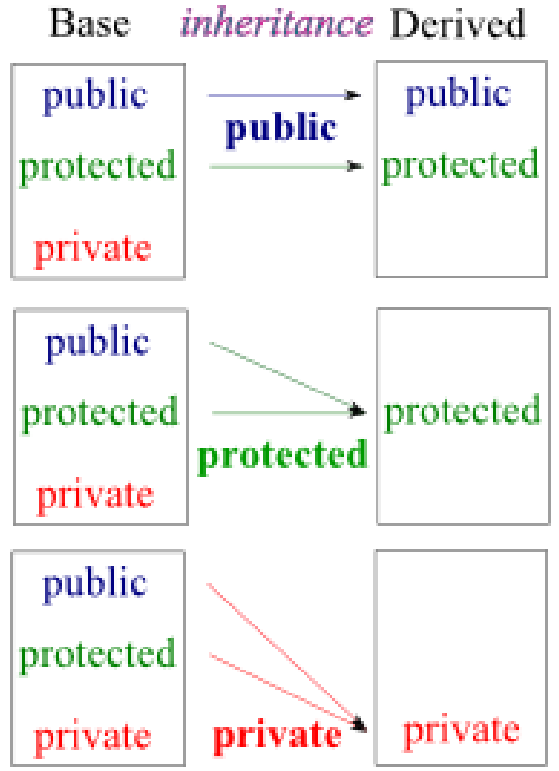
Default Hiding:

In C++, the definition of class queue can specify that its base (list) class is to be private:

```
class queue : private list {
public:
    using list::empty;
    using list::head;
    // but NOT using list::append
    void enqueue(gp_list_node* new_node);
    gp_list_node* dequeue();
};
```

Note:
Sharing is an exception.

Base Class Visibility	Derived class visibility		
	Public derivation	Private derivation	Protected derivation
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected



LECTURE 18

static fields

Static Fields and Methods

Static method can only access static data. Instance method can access both static/instance data

- C++ classes may also contain, static instance fields -- a single field shared by all members of a class
- Often used when declaring class constants (since you generally only need one copy of a constant)
- To make a field static, add the `static` keyword in front of the field
  - can refer to the field like any other field (but remember, everybody shares one copy)
  - static variables are also considered to be global, you can refer to them without an instance
  - static fields can be initialized (unlike other fields)

Memory Management
Class Initializers

C++	`class MyClass { static int x; };` `MyClass::x = 0;`

C#	`static class MyClass {`
	`  static int x;`
	`  static MyClass() { x = 10; } }`

Java	`class MyClass {`
	`  static int x;`
	`  static { x = 10; } }`

Scala	`object MyClass {`
	`  var x: Int;`
	`  x = 10 }`

Ruby	`class MyClass`
	`  @@x = 10`
	`end`

JS	`function MyClass() {}` `MyClass.x = 10;`