Basic Concepts of C++

System Platforms Group
Training Program

Objective

What you should expect…

- an introduction to object programming paradigms
- a selected subset of C++ concepts useful for SystemC and transactional modeling

What you should not expect…

- an exhaustive training on C++
- a training to make you a C++ expert overnight

Prerequisite

- You should have minimal knowledge of C and C++ before tackling this training

Outline

Module 1 - Introduction

- Introduction to C++
- OOPS
- Organization of Data and Functions

Introduction to C++

C++ is a general purpose programming language that

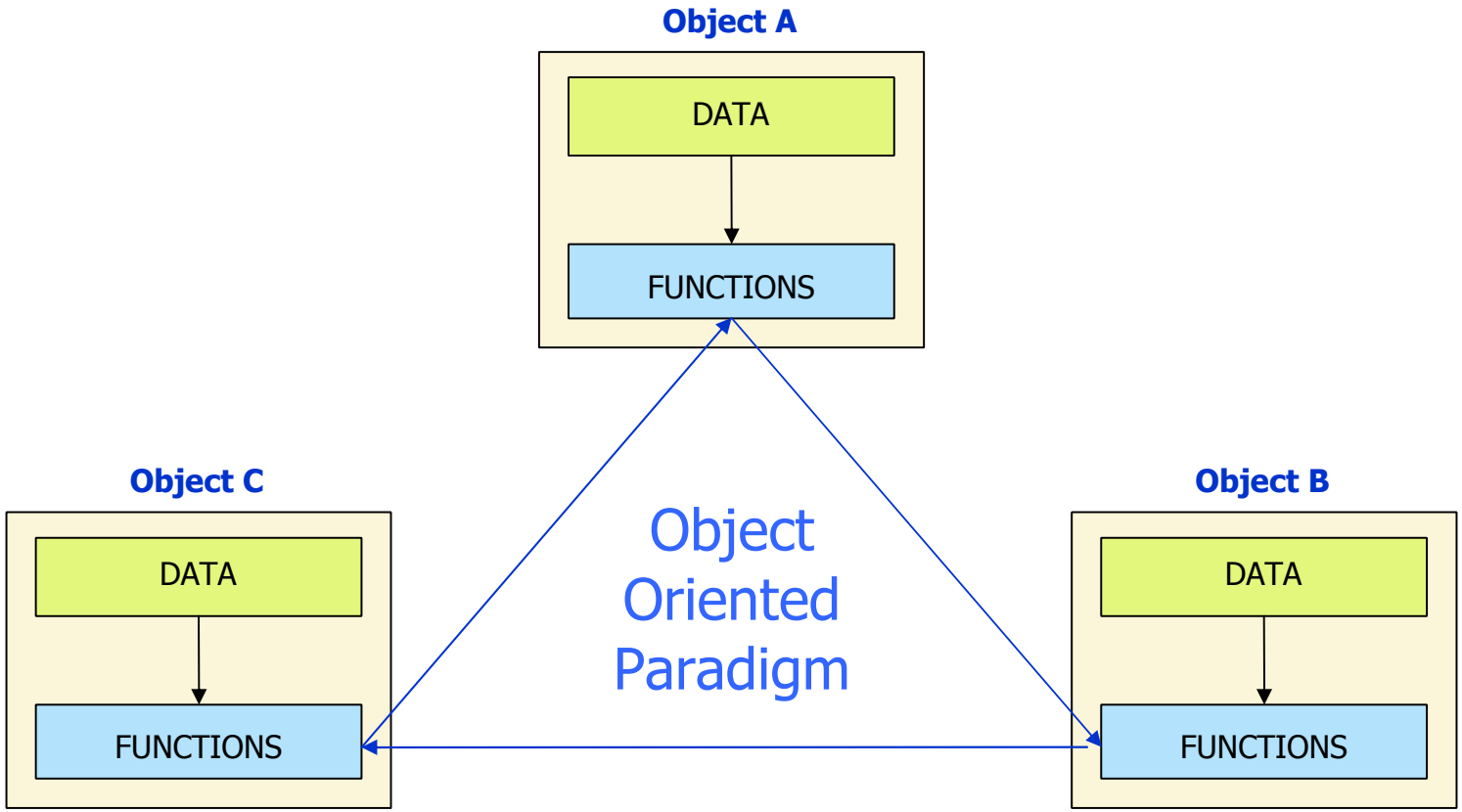
- is a superset of C
- supports data abstraction
- supports object-oriented programming (i.e. classes)
- supports new input/output interface
- supports references
- supports operator overloading
- supports generic types (i.e. templates), file operations, and exceptions

OOPS - Object Oriented Programming

OOPS is characterized by the following features:

- emphasis is more on data rather than procedure
- programs are divided into objects
- data structures (classes) are designed in such a way that they characterize the objects
- data is hidden and cannot be accessed by external functions
- objects communicate with each other through functions
- bottom-up approach is adopted in program design

Organization of Data and Functions



Objects interact with each other using *functions* in object oriented paradigm

Module 2 - Objects and Classes

- Object
- Class
- Class vs. Struct

Object

OO Design and Programming

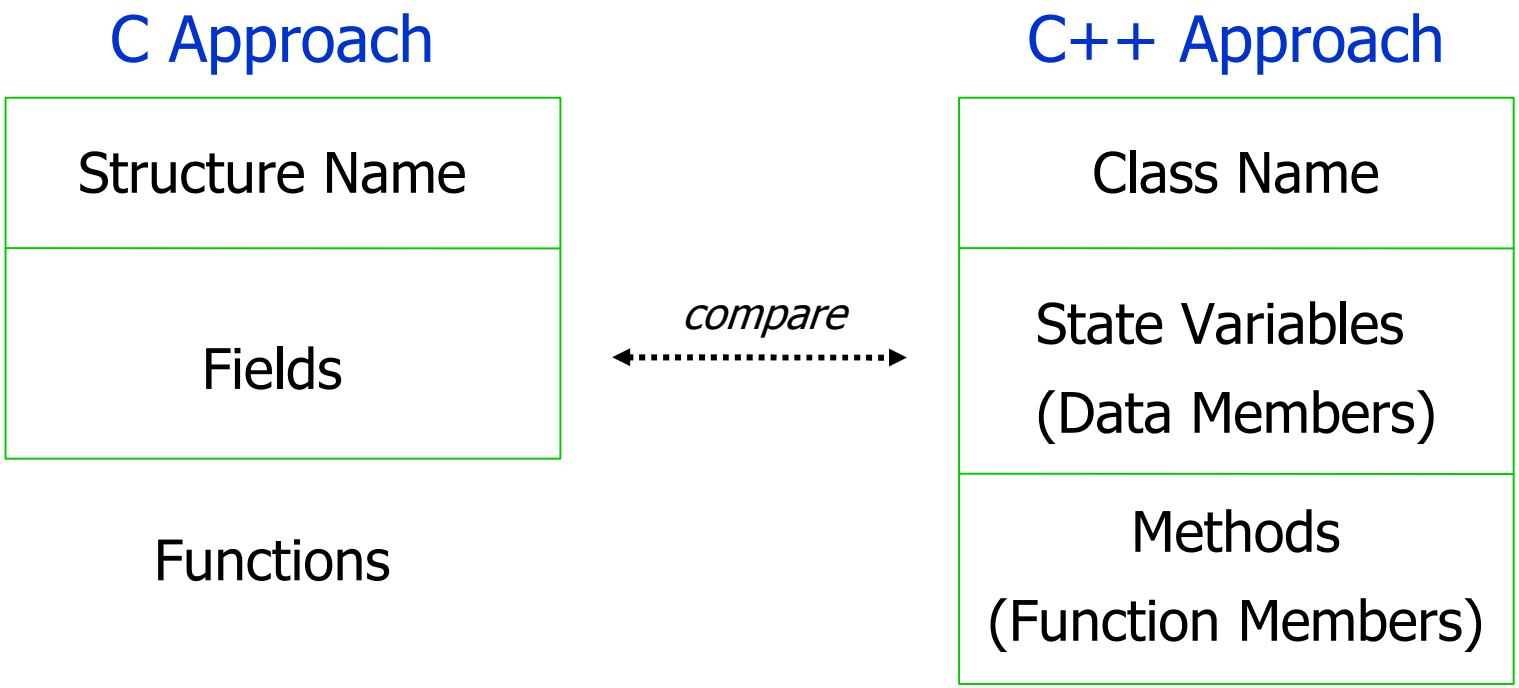
- Object-oriented design and programming methodology supports good software engineering by
  - promoting a thinking manner in which we model the way we think as we interact with the real world

- Example: Watching Television
  - The remote control is a physical object with properties as
    - weight, size, ability to send message to the television, etc
  - The television is also a physical object with various properties

Characteristics of Object

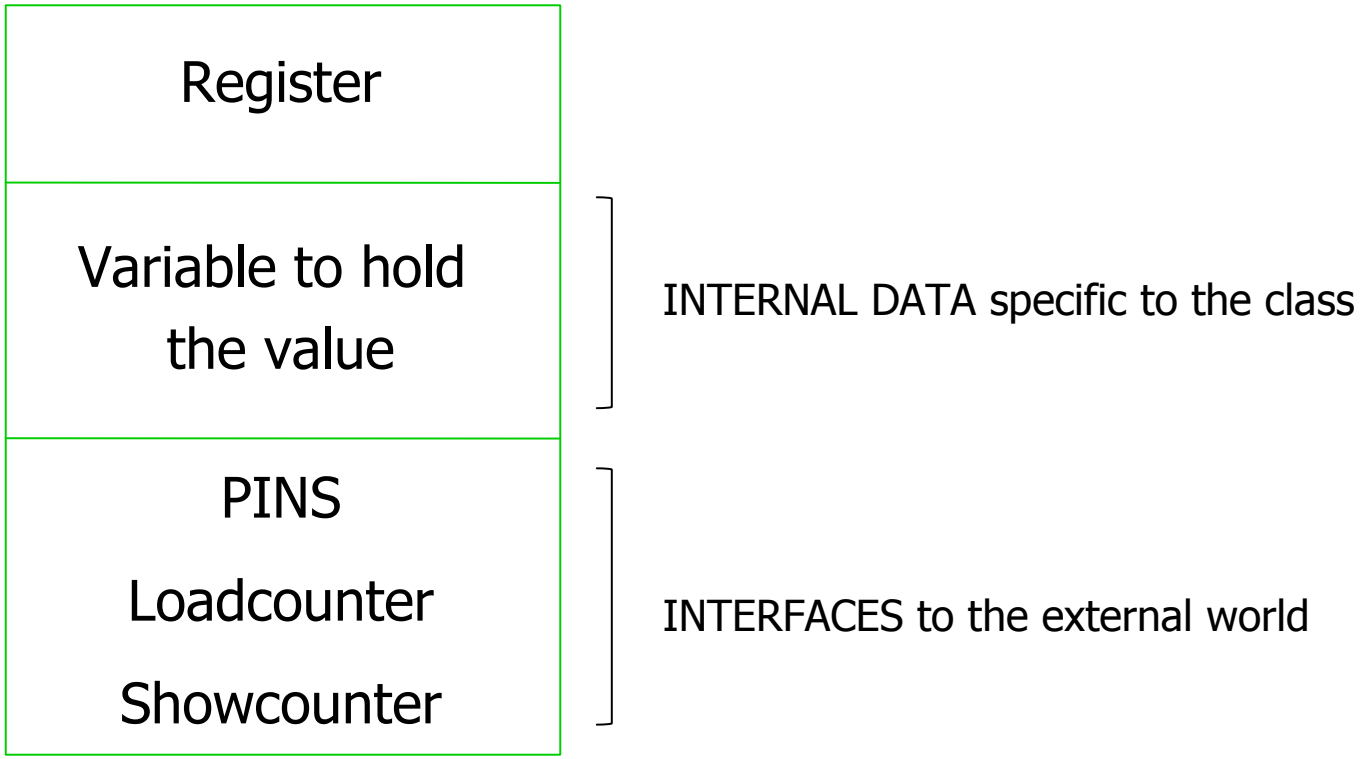
- An object is almost anything with the following characteristics:
  - Name
  - Properties
  - Ability to act upon receiving a message
    - Two basic message types
      - Directive message to perform an action
      - Request message to change one of the object properties

Modeling Approach
C vs. C++



Idea of Class Modeling in C++

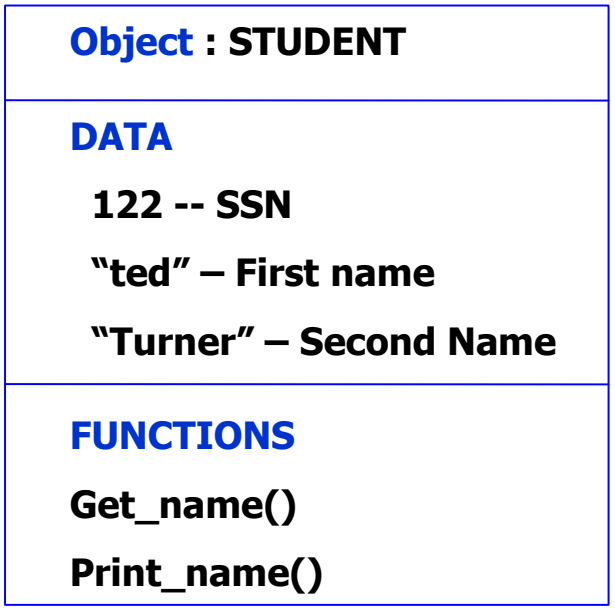
C++ Modeling Approach



Definition of Object

- Objects model real world entities that may represent a person, a place, a vector, time, a list, a hardware block, etc
- Each object can be defined to contain data and functions for manipulating the data

Defining an object



Class

Classes

- Classes are means to model objects with the followings:
  - attributes / data members
  - operations (i.e. member functions or methods)

- A look at the StudentRecord as a class:

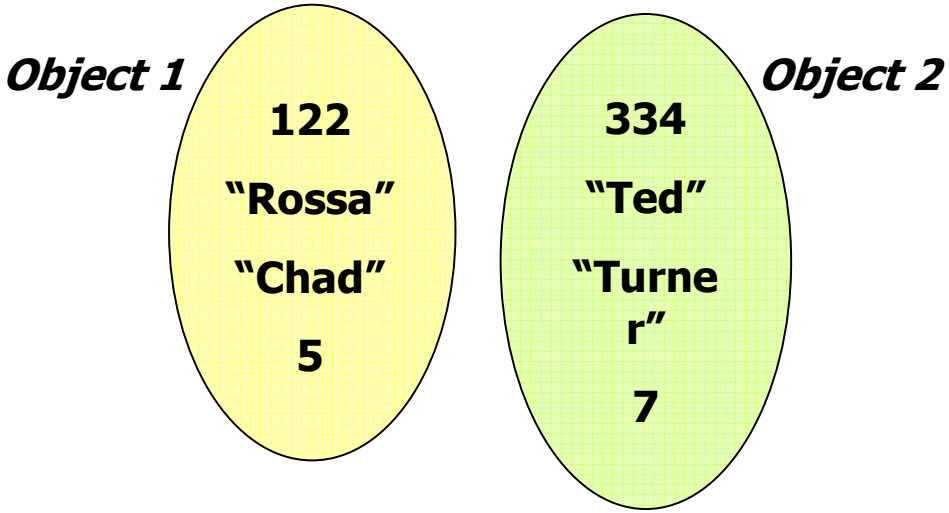
```
class StudentRecord
{
    private:
            int SSN; // Attributes
            char first_name[20]; char last_name[20], int grade;

    public:
            void get_name(); // Methods: operations on attributes
            void print_name();
};
```

Class Instances

Class

STUDENTRECORD
`int SSN;//ATTRIBUTES`
`char first_name[20];`
`char last_name[20],`
`int grade;`
`void get_name();` `//Methods`
`void print_name();`

Objects (Instances of class)



Modeling Advantages of C++ vs. C

- Modular/User Defined Data Type – Abstract hardware design concepts such as concurrency and bit vectors can be added using the class mechanism
- Data Hiding – Data hiding prevents users from changing data item and hides the implementation details from users
- Code Reuse - By encapsulating the details of data and methods within an object, C++ provides high degree of reusability
- Hierarchical - Users do not have to start their modeling effort from scratch but directly build upon the already-developed classes

Class vs. Struct

Difference between Classes and Structs in C++

- The difference between C++ classes and C++ structures is just the default accessibility associated with the members of each (warning: *struct* in C++ != *struct* in pure C)

- In a class, all members are private by default unless otherwise stated

```
class a
{
    int x;
};
```

note the difference

```
class b
{
public:
    int y;
};
```

- In a struct, all members are public by default

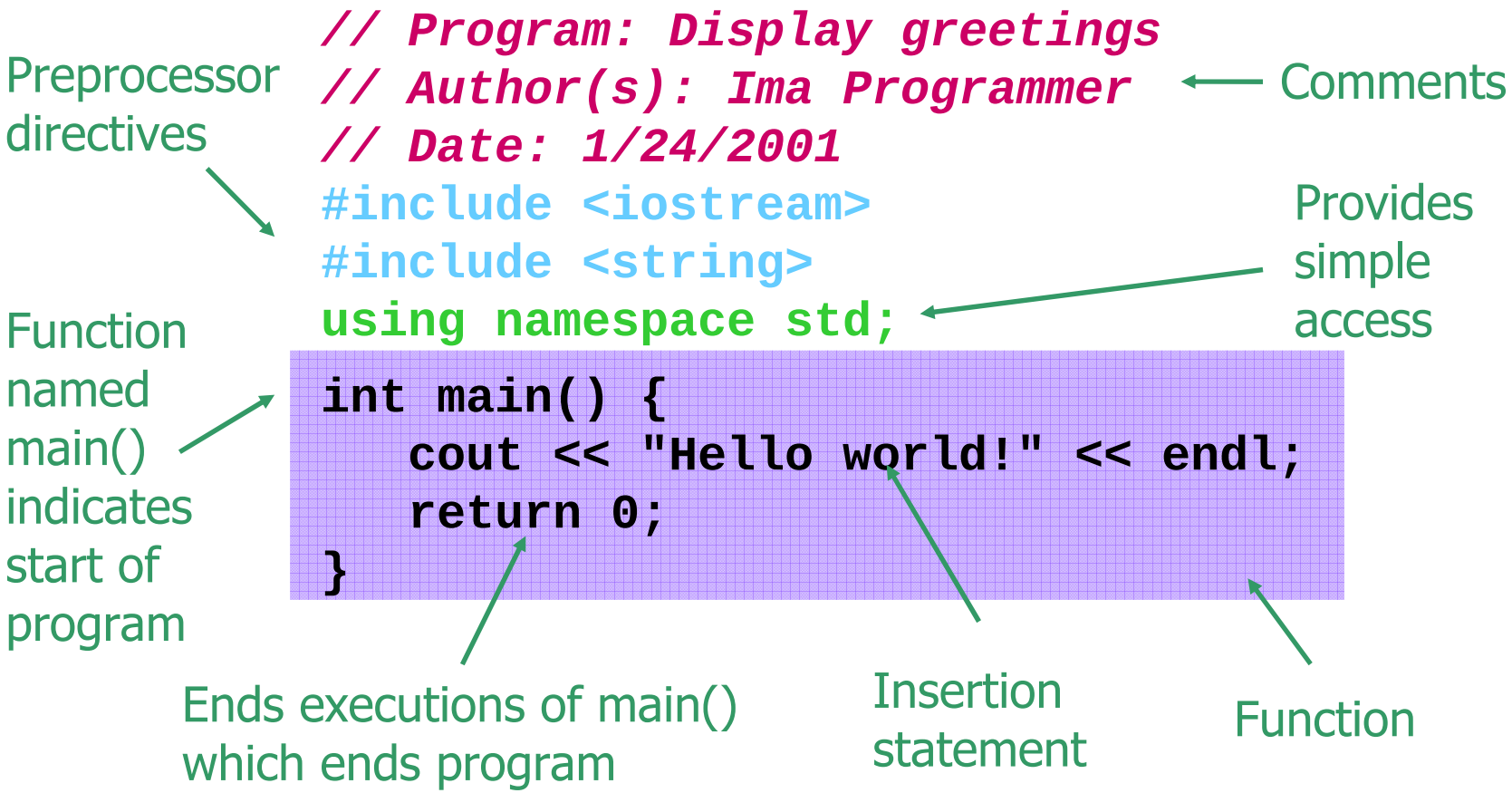
```
struct b
{
    int y;
};
```

Module 3 - C++ Basics

- First Glance at C++
- Member Function
- Assignment Operator
- *this* Pointer
- *New* & *Delete* Operators

First Glance at C++

A Simple Program - Greeting.cpp



Comments

- Objective
  - Allow prose or commentary to be included in programs
- Importance
  - Programs are read far more often than they are written
  - Programs need to be understood so that they can be maintained
- C++ has two conventions for comments
  - `//` single line comment (preferred)
  - `/*` long comment `*/` (save for debugging)
- Typical uses
  - Identify programs and authors/programmer
  - Record when programs are written
  - Add descriptions of modifications

A Program without Comments

- Can you identify the job done by each line in the program *hello_world.cc* shown below without comments?

```
#include <iostream.h>
void main()
{
        cout << “Hello World!” << endl;
}
```

Same Program but with Comments

- With comments, we can tell quickly what each line of the program *hello_world.cc* does

```
#include <iostream.h>
// Include a header file called iostream.h
// Header files contain some definitions that a program uses
// iostream.h has definitions for input/output, objects, etc
void main()
/* This indicates the beginning of the 'main' function same as
in C but in C++ every function must have return type */
{
        cout << "Hello World!" << endl;
        // Output "Hello World!" on screen
}
```

Input/Output in C++

- **cin**
  - The standard input device that is normally the keyboard
  - It replaces the scanf() statement in C

- **cout**
  - The standard output device that is normally the computer screen
  - It replaces the printf() statement in C

Note: The old C interface is still available

A Program with Input/Output

- A simple program with input/output, *age.cc*, is given below

```
#include <iostream.h>
void main() {
        int age;
        cout << “Enter your age: ”;
        cin >> age; //read the age
        cout << “You are ” << age << “years old”;
}
```

Declaring Variables

- Variables can be initialized and declared in the middle of a program

- Example

```
int age;
cout << “Enter your age: ”;
cin >> age; //read the age
cout << “You are ” << age << “years old” << endl;
int x=10, y=5;
char ch=‘a’;
float pi=3.1415;
cout << “Pi value is: ” << pi << endl;
```

Declarations and initializations of variables

Class Types

- Class Construct
  - Allows programmers to define new data types for representing information, i.e. class types
  - Class type objects can have both attribute components and behavior components
  - Makes C++ an object-oriented programming language

Terminology

- Client
  - A program using a class
- Object Behavior
  - Realized in C++ via member function, i.e. method
    - e.g. RectangleShapes can be drawn or resized
- Object Attribute
  - Known as data member in C++
    - e.g. RectangleShapes can have width, height, position, color, etc

Member Function

Role of Member Functions

- *Member functions* or *methods* provide a controlled interface to data members as well as object access and manipulation

- Three important roles of member functions include
  - creating objects of a class
  - inspecting, mutating, and manipulating objects of a class
  - keeping data members in a correct state

- Example: RectangleShapes may have the following methods
  - SetSize()
  - SetColor()
  - Draw()

Constructor

- Constructors are member functions that initialize an object during its definition
- Example:
  - `RectangleShape R(W, x, y, c, w, h);`
- Factoid:
  - Constructors do not have a type for it is considered superfluous

Inspector

- Inspectors are member functions that act as a messenger that returns the value of an attribute

- Example:
  - RectangleShapes may have an inspector GetColor()
    - `color CurrColor = R.GetColor();`

Mutator

- Mutators are member functions that can change the value of an attribute

- Example:
  - RectangleShapes may have a mutator SetColor()
    - `R.SetColor(Black);`

Facilitator

- Facilitators are member functions that cause an object to perform some action or service

- Example:
  - RectangleShapes may have a facilitator Draw()
    - `R.Draw();`

Member Access Specifiers

- Member access specifiers can have public or private labels
- Data members or member functions of a class declared under the public access specifier are accessible wherever the program has access to an object of that class
- Public functions implement the services that a class offers to users, which are also referred to as class interfaces
- Data members or member functions of a class declared under the private access specifier can only be accessed by the member functions of the class

Example of a Complete Class

- Shown below is a complete class of *StudentRecord*:

```
class StudentRecord
{
   private:
       int SSN;
       char first_name[20], last_name[20], address[40],
                    grade;
   public:
       void get_name();
       void put_name() {  cout << first_name << " " <<
                                  last_name; }
       void get_address(); // Reads from keyboard
       void put_address(); // Displays on screen
};
```

Example of a Complete Class (cont.)

```
void StudentRecord :: get_address()      // Similar to C function prefixed
                                         // by the name of class
{
    cout << “Enter the address: ”;
    cin >> address;      // address can be accessed by the member function
                         // get_address() as address is private
                         // and get_address is a member function
}

void StudentRecord :: put_address()
{
    cout << “Address: ”;
    cout << address;
}
…
void main()
{
    StudentRecord s;
    s.getname();         //LEGAL because getname() is public
    cout << s.SSN;       //ILLEGAL because SSN is private
    cout << s.address  ;         //ILLEGAL because address is private
}
```

Assignment Operator

Assignment Operator in C++ Classes

- The assignment operator "=" is used to assign an object to another object of the same class

- Example:

```
…
StudentRecord r, s;
…
r.get_name();
s = r;
…
```

this Operator

this Pointer in C++ Classes

- The "this" pointer is used to point to an object itself
- Through "this" pointer, any member functions of an object can find out the address of the object
- Example

```
void bind (signal s);      // Function taking an object of type
                           // signal as an argument
...
class signal
{
  void bindme( )
  {
  bind (*this); // *this is the current object which is of type signal
  }
}
```

this Pointer in C++ Classes (cont.)

- The "this" pointer can also be used to distinguish state variables from arguments

- Example

```
class A
{
  int x = 3;
  void f (int x)
  {
  cout << "x= " << x << ", this->x= " << this->x << endl;
  }
};

int main()
{
A instance;
Instance(1); // x= 1, this->x= 3
}
```

New & *Delete* Operators

New & *Delete* Operators in C++

- *New* and *delete* operators are memory management operators in C++, which could be used on variables of any types (classes or not)

- *New* operator creates object of any type, e.g.

```
StudentRecord  * p;
p = new StudentRecord;
cout << p->getname();
```

-> invokes methods and access to data members from a pointer on an object

- If *new* operation fails, an exception is raised (it may return NULL if exception handler is appropriately set)

- *Delete* operator destroys a data object created with *new* operator, e.g.

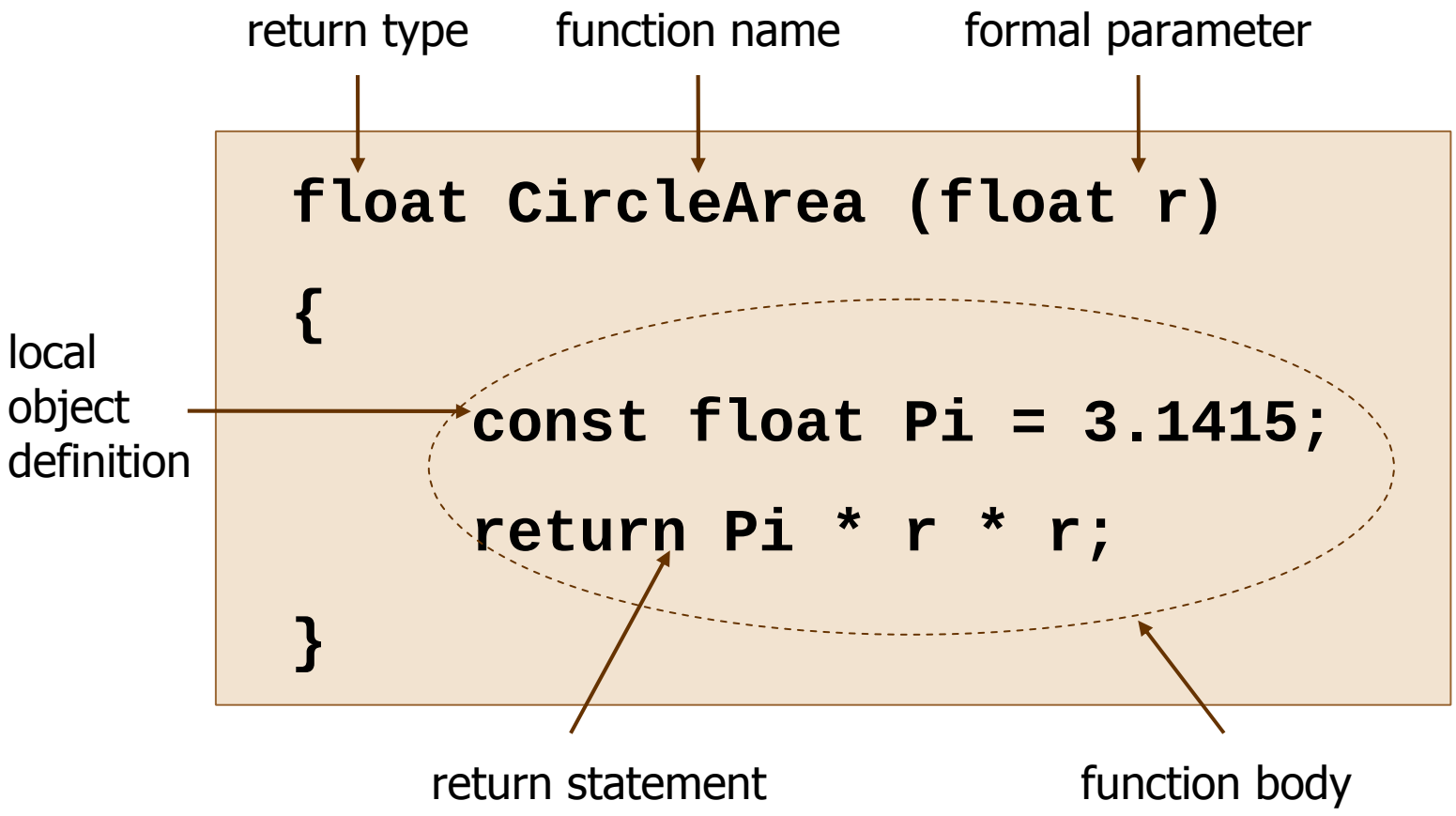
```
delete p;
```

Module 4 - C++ Functions & Overloading

- Definition of Function
- Functions in C++
- Function Overloading

Definition of Function

What to Define in a Function



Functions in C++

Passing Arguments in C++ Functions

- Arguments can be passed by three different ways in C++:
  - Value
  - Reference
  - Pointer
- So far in our discussion, arguments passed to functions do not change because whatever passed to the functions is only a copy
  - this method is called pass-by-value
- If you want the value of argument to change, there are two ways of to pass the arguments to functions:
  - pass-by-reference
  - pass-by-pointer (same as in C)

Pass-by-Reference

- To pass an argument by reference, declare a function prototype as follows:

```
void someFunction (int& argByRef);
```

- To call the function:

```
int a;
someFunction (a);
```

- The value of "a" can now be changed by the function

Pass-by-Pointer

- To pass an argument by pointer, declare a function prototype as follows:

```
void someFunction (int* argByPtr);
```

- To call the function:

```
int a;
someFunction (&a);
```

- The value of "a" can now be changed by the function

Constant Parameters

- The **`const`** modifier can be applied to formal parameter declarations
- The **`const`** indicates that the function may not modify the parameter
- Example:

```
void PromptAndGet (int &n, const string &s)
{
    cout << s ;
    cin >> n ;
    s = "Got it"; // illegal assignment that compiler will catch
}
```

- Sample invocation:

```
int x;
PromptAndGet (x, "Enter number (n): ");
```

Default Parameters

- Observations
  - Our functions up to this point require us to pass explicitly a value for each of the function parameters
  - It would be convenient to define functions that accept a varying number of parameters

- Solution: Default Parameters
  - Allow programmers to define a default behavior
  - The value for a parameter can be implicitly passed
  - Reduce needs for similar functions that differ only in the number of parameters accepted

Function Overloading

What is Function Overloading?

- Overloading means using the same thing for different purposes
- C++ permits overloading of functions, i.e. we can use the same function name to create functions that perform various tasks
- This is known as *function polymorphism* in OOPS
- A family of functions can be designed with a single function name but of different argument lists
- The function would perform different operations depending on the argument list in the function call

Example of Function Overloading in C++

Example

- An overloaded add() function where the correct function to be invoked is determined by checking the number and type of argument

```
//Declarations
int add (int a,int b);          //prototype 1
int add (int a,int b,int c);    //prototype 2
int add (double a,int b);       //prototype 3

//Function calls
cout<<add(5,10)                          //uses
prototype 1
cout<<add(5,10,20)              //uses prototype 2
cout<<add(0.72,10)              //uses prototype 3
```

Lab 1
Basic Features of C++

Now, it is time to take a coffee break and work on your first lab!

Notes: Please download lab notes and database from the SPG training center

Module 5 – Constructor and Destructor

- Constructor
- Destructor

Constructor

What does a Constructor do?

- When a class object is created, its members can be initialized by the constructor function of that class
- The constructor is executed whenever an object is declared or created dynamically using *new* operator
- Constructors
  - have the same name as the class itself
  - do not return values
  - cannot be called explicitly
  - can be passed arguments

Example of Constructor

```
class StudentRecord
{
                private:
                        int SSN; //Attributes
                        char first_name[20], last_name[20],
                                        address[40], grade;
                public:
                        StudentRecord (int SSN); //Constructor
                        ...
};

StudentRecord :: StudentRecord (int SSN) //Constructor definition
{
        this->SSN = SSN;
}
```

Passing Arguments to Constructors

Arguments can be passed to a constructor when:

- declaring an instance

```
ClassName instance(arg1, arg2, …);
```

- using the *new* operator

```
ptr = new ClassName(arg1, arg2, …);
```

Destructor

What does a Destructor do?

- Destructors are used to clean up an object when the object is about to be destroyed
- For instance, if your object has some pointers as its members, you might want to free the memory pointed by those pointers sometimes
- Such operations can be performed by destructors

How to Use Destructors

- Destructors are called
  - by default for each class object at the end of a program
  - whenever the *delete* operation is called

- Example: the destructor for the class StudentRecord is

```
StudentRecord :: ~ StudentRecord(){…};
```

- Destructors are similar in nature to constructors except that
  - they are executed when an object is about to be destroyed
  - you cannot pass arguments to destructors

Lab 2
Classes and Objects in C++

Now, it is time to take a coffee break and work on your second lab!

Notes: Please download lab notes and database from the SPG training center

Module 6 – Operator Overloading

- Purpose
- Example
- Guidelines

Purpose

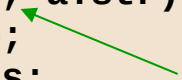
- C++ has several operators such as +, -, *, /, %, ++, --, =, +=, etc
- Sometimes it is more meaningful and natural to use these operators on objects
- Illustration
  - Using "+" operator is more intuitive than using the "strcat" function if we want to add two strings
  - This is done by treating an operator as a function that takes two arguments (or one argument in the case of a unary operator)
  - For adding two strings, think of *a = b+c;* as *a = +(b, c);*

Example

```
// string.cc
// This is a memory efficient string class
// It allocates just enough memory to store the string passed to it!
// It demonstrates overloading the += operator
// This is overloaded as a MEMBER FUNCTION

#include <iostream.h>
#include <string.h>
class String {
    private:
      char *str;
    public:
      // Constructor
      String (char* ptr);
      // Member functions
      void print();
      // Overloaded operators
      const String& operator+=(const String& a);
};
```

Example (cont.)

```
// constructor
String :: String (char *ptr) {
  str = new char [strlen (ptr) + 1];
  strcpy (str, ptr);
}
//print method implementation
void String :: print() {
  cout << str << endl;
}
//Overloading the += operator
const String& String :: operator+=(const String& a) {
  char *temp = str; // save the string in a temporary variable
  str = new char[strlen(temp) + strlen(a.str) + 1]; // allocate variable
  strcpy( str, temp);                               // with convenient size
  strcat( str, a.str);                              // concatenate strings
  delete temp;
  return *this;
}
//-----------------------------------------------------------
void main(){
  String s1("We ");  String s2("love ");  String s3("C++");
  s1 += s2 += s3;
  s1.print();
}
```



str is a data member of the class String

Guidelines

Guidelines for operator overloading are as follows:

- Some of the operators cannot be overloaded
  - : :: sizeof
- New operators cannot created
- Unary operators can be overloaded
- Binary operators can be overloaded

Module 7 - Inheritance

- Illustration: What is Inheritance?
- Inheritance in C++
- Features of Inheritance
- Examples of Inheritance

Illustration: What is Inheritance?

Inheritance in Programming

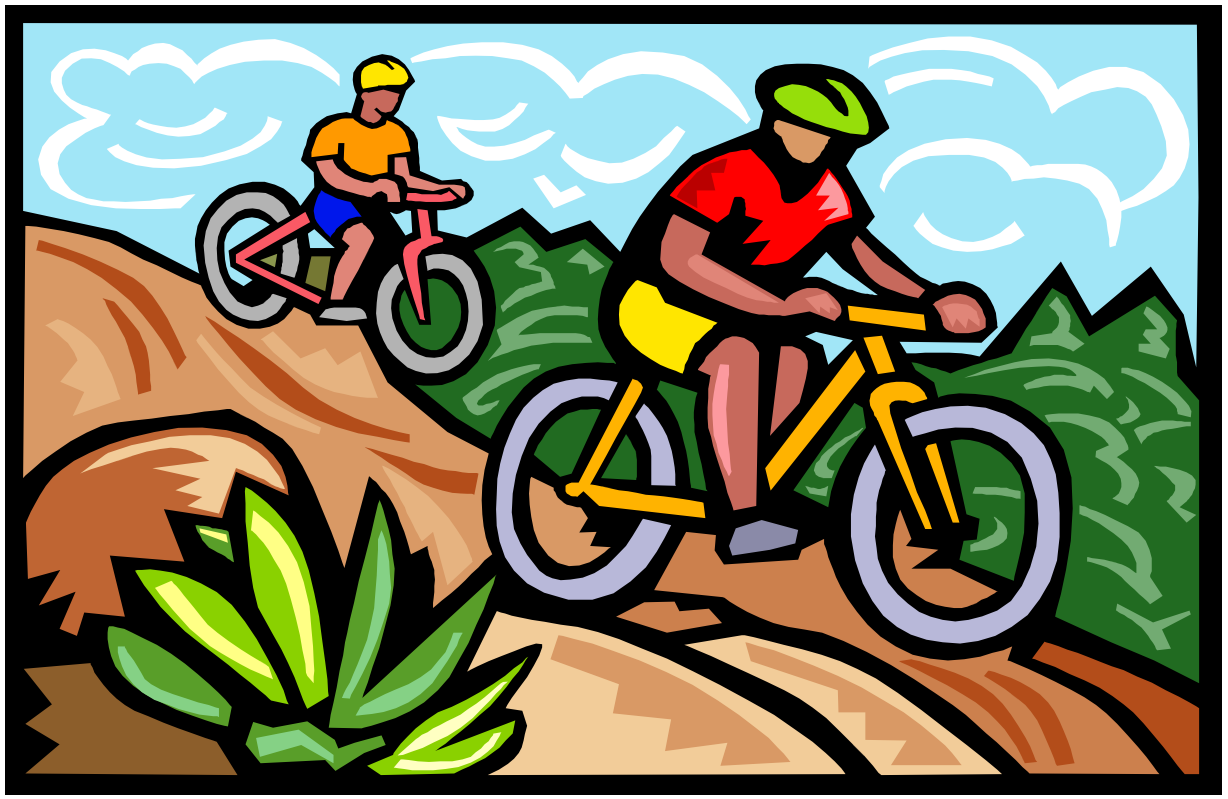
- Inheritance is a mechanism for deriving new classes from existing classes
- To illustrate the idea of inheritance, let us guide you through the analogy of bicycle family

Think of a Bicycle

Think of a Tandem Bike

Think of a Racing Bike

Think of a Mountain Bike

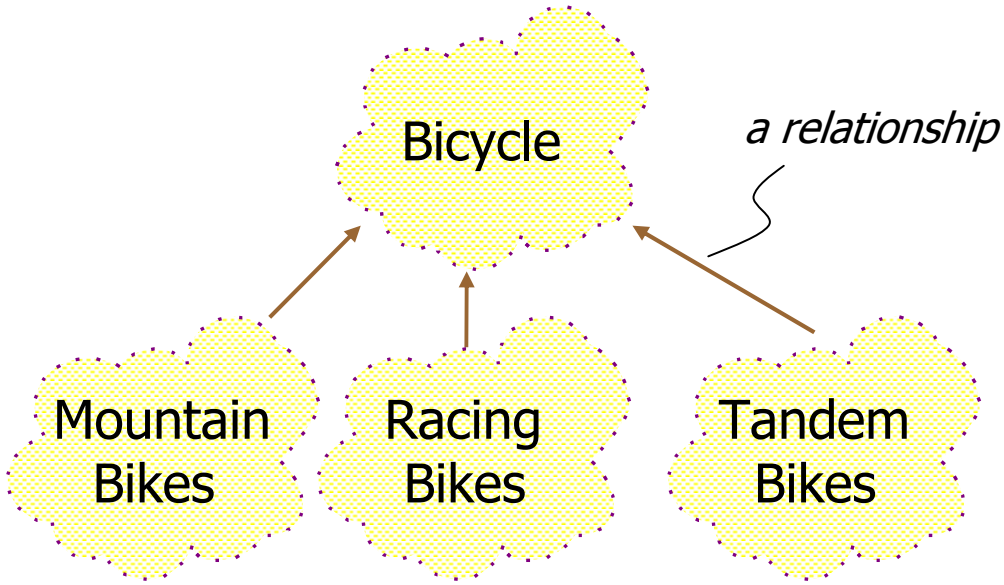


Now, think of all these bicycles together

- A tandem bicycle *is a kind of bicycle*
  - Bicycle with two seats
- A mountain bicycle *is a kind of bicycle*
  - Bicycle with shock-proof
- A racing bicycle *is a kind of bicycle*
  - Bicycle with lightweight aerodynamic construction
- Tandem, mountain, and racing bicycles are all *specialized* bicycles

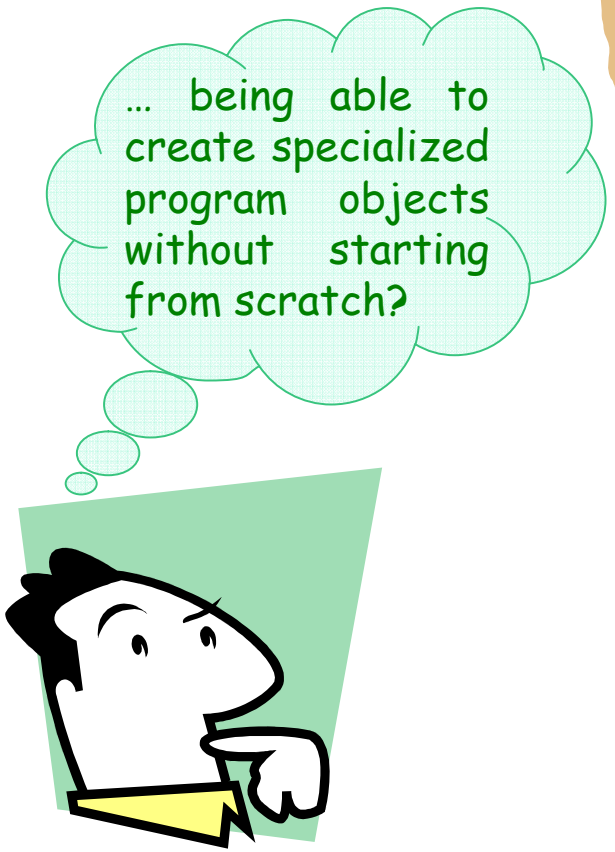
Relationship in Bicycle Family

- We can conclude that there exists a *relationship* between each specialized type of bicycle and the basic bicycle



Now, think of how such relationship can be helpful in terms of programming …

Wouldn’t it be nice…



Yep! We can qualify the previous relationship as inheritance in C++!

- Inheritance is the object-oriented programming mechanism for the object specialization
- For instance, you can create specialized objects from the class Bicycle, e.g.
  - Tandem bike
  - Racing bike
  - Mountain bike

Inheritance in C++

Definition

- Inheritance in C++ is the ability to define new classes from an existing parent class known as the base class
- Such ability is indeed a form of *software reusability* allowing new classes to be created from existing classes
- Each new class is a specialized version of the parent class that will inherit all attributes and behavior of the parent class
- The inherited properties can be overwritten if necessary

Motive

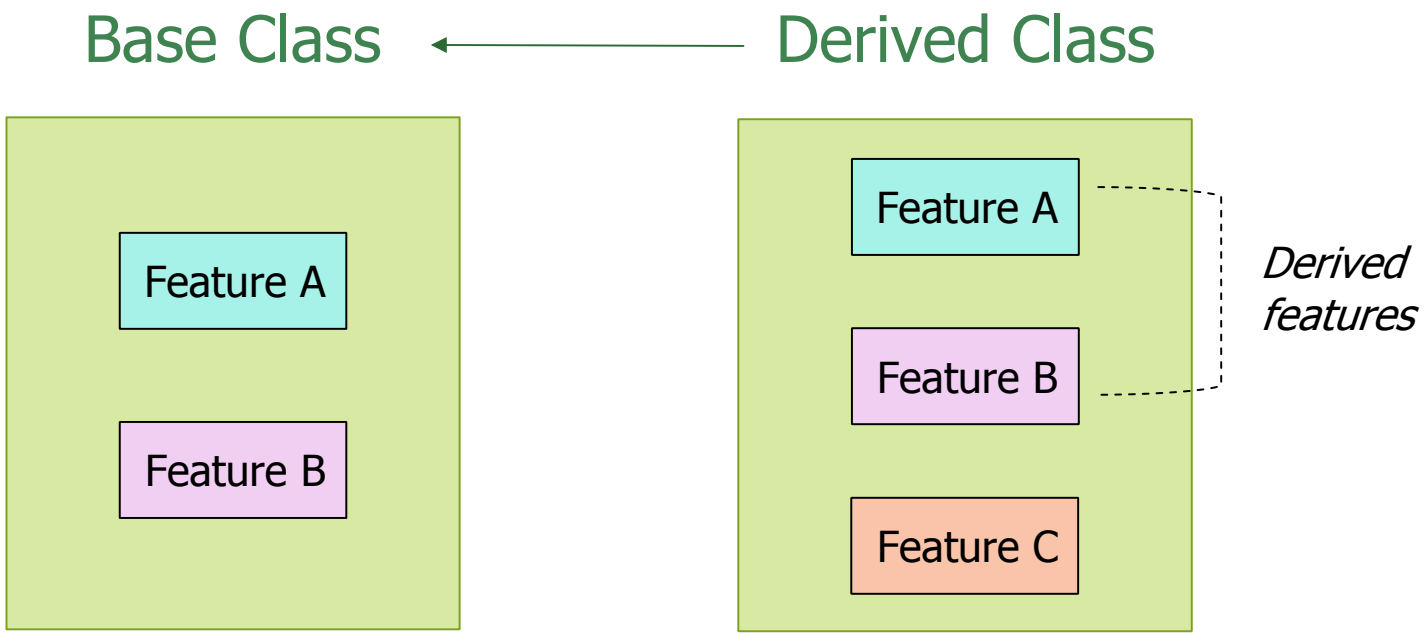
There are good reasons to use inheritance in C++:

- Provide a natural way to reuse codes
- Save time through software reuse
- Encourage uses of proven software
- Foster programming by extension rather than reinvention

Terminology

- **Base Class**
  - Original or existing parent class that is already well defined
- **Derived Class**
  - New class defined upon a base class
- **Single Inheritance**
  - Inheritance of a class derived from one single base class
- **Multiple Inheritance**
  - Inheritance of a class derived from more than one base class

Graphical Representation



- Depicted above is the graphical representation (a-la UML) of class inheritance
- The derived class is related to the base class with an arrow, the head being oriented towards the base class

Features of Inheritance

Single and Multiple Inheritance

- Recall that
  - Single Inheritance – class derived from one base class
  - Multiple Inheritance – class derived from several base classes

- Example of multiple inheritance

```
class seaplane : public plane, public boat
{
  seaplane(…) : plane(…), boat(…) // constructor-chaining
  { … }
};
```

Multiple inheritance could be a dangerous feature if not properly understood!

Member Access Specifier

- Three types of members exist in a class
  - Public Member
  - Protected Member
  - Private Member
- Such characteristic is indicated by the member access specifier of public, protected, and private
- The member functions of a derived class can access public or protected members of the base class
- Private members of the base class are not accessible by derived class
- A protected member can be accessed by member functions of its own class or any derived class based on it's own class

Member Access Specifier (cont.)

- The table shown below summarizes the access right to a class member according to its different access specifiers

Access Specifier	Accessible by its own class	Accessible by its derived classes	Accessible by objects of any other classes
Public	**Yes**	**Yes**	**Yes**
Protected	**Yes**	**Yes**	**No**
Private	**Yes**	**No**	**No**

Public and Private Inheritance

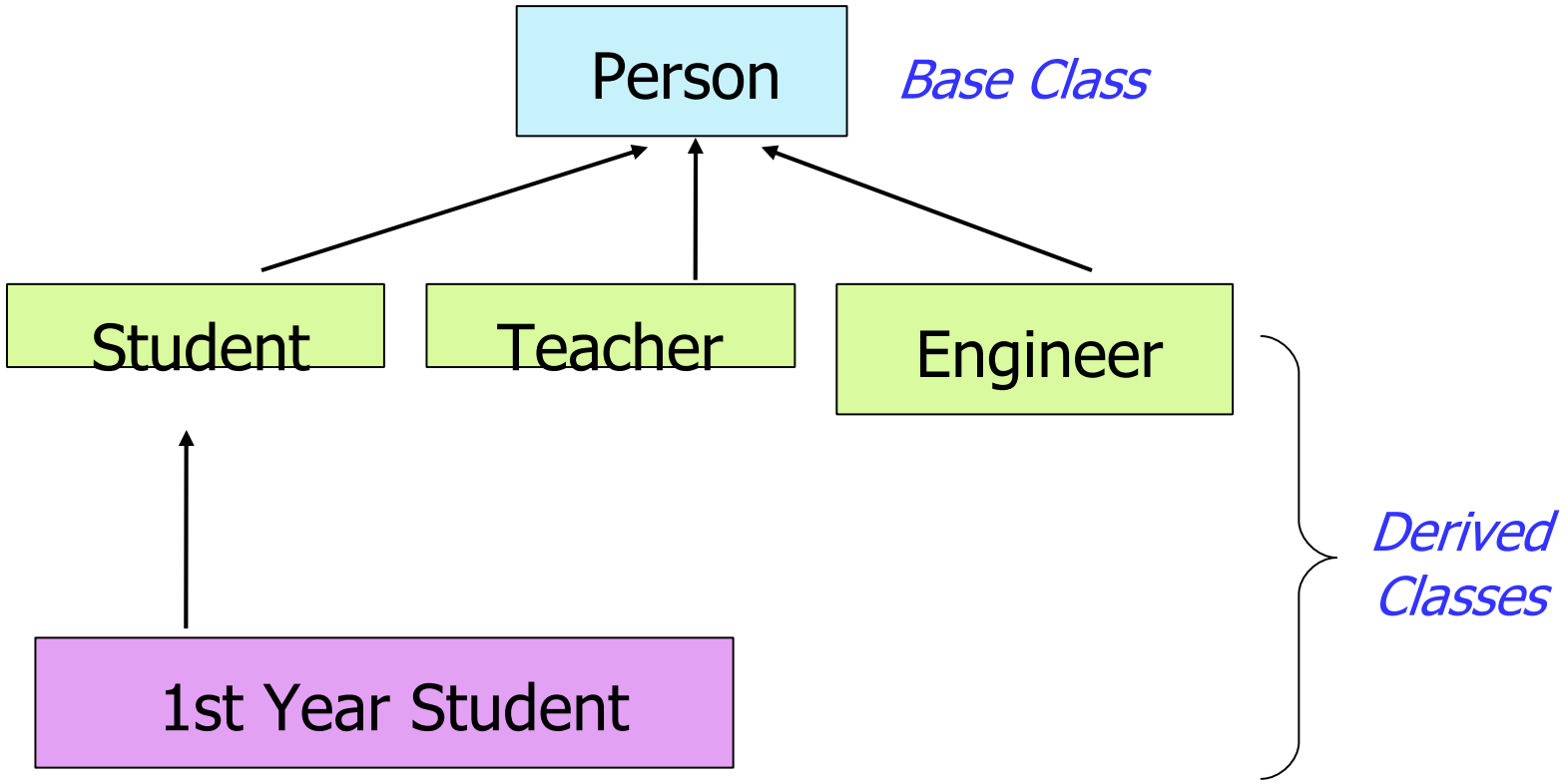
- Public Inheritance
  - Indicated by the keyword *public* in the derived class
  - It specifies that objects of such derived class can access *public* and *protected* members of the base class but not the *private* members

- Private Inheritance
  - Indicated by the keyword *private* in the derived class
  - It specifies that *public* and *protected* members of the base class become *private* members in the derived class (*not recommended*), as usual *private* can not be access.

- Examples

```
class student : public person {…};
class student : private person {…};
```

A Look at Class Hierarchy



Examples of Inheritance

Base Class "Person" and Derived Class "Student"

```
class Person { // Base Class
  private:
          char* name;
          int SSN;
  public:
          Person (const char* name, int SSN);
          void putDetails();
};
class Student : public Person { // Derived Class - a public inheritance
  private:
          char *univ;
          char grade;
  public:
     Student (const char* name, int SSN, const char* univ, char grade);
    void putDetails(); // Override the putDetails() of the base class
    void setGrade (char grade);
};
```

Example (cont.)

Definition of Base Class "Person"

```
// Definition of the base class "Person"

Person :: Person (const char* name, int SSN) { // constructor
        this->name = new char[strlen(name)+1];
        strcpy (this->name, name);
        this->SSN = SSN; // used here to distinguish the data member
                                // from the function parameter
}

void Person :: putDetails() {
        cout << name << " " << SSN << endl;
}
```

Example (cont.)
Definition of Derived Class "Student"

```
// Definition of the derived class "Student"

// constructor of the derived class
Student :: Student (const char* name, int SSN, const char* univ, char grade)
: Person (name, SSN)      // call constructor of the base class first
{
  this->univ = new char[strlen(univ)+1];
  strcpy (this->univ, univ);
  this->grade = grade;
}

void Student :: putDetails() {
  Person :: putDetails(); // call putDetails() of the base class
  cout << univ << " " << grade << endl;
}
```

Module 8 - Polymorphism

- Concept of Polymorphism
- Virtual Function

Concept of Polymorphism

Definition

- Literally, the Greek word "polymorphism" means having many different forms
- In object-oriented programming, polymorphism means being able to assign different meaning or usage to a given item under different context
- Particularly, polymorphism allows an entity such as a variable, a function or an object to carry multiple forms
- The concept of polymorphism will be illustrated in the coming example

Illustration

```
class Point {
   protected :
      int x, y;
   public :
      Point();
      Point (int x, int y);
      void printme ()
};

Point :: Point() {
 x = 0; y = 0 ;
}

Point :: Point (int x, int y) {
   (*this).x = x;
   (*this).y = y;
}

void Point :: printme() {
 cout<<“This is a point”<<endl;
}
```

← Base class, *Point*

Base class methods

Illustration (cont.)

```
class Circle : public Point {
   double radius;
   public:
        // constructor
        Circle (double r = 0.0, int x =0 , int y =0);
        void printme();
};

Circle :: Circle( double r, int x, int y) : Point(x,y){
   radius = r;
}

void Circle :: printme() {
   cout << “This is a circle” << endl;
}
```

Derived class, *Circle*

Base class, *printme ()*, overridden here as a derived class function

Illustration (cont.)

Can you figure this out?

```
int main() {
   Point p, *pPtr = &p;
   Circle c, *cPtr = &c;

   pPtr->printme();              // “This is a point”
   cPtr->printme();              // “This is a circle”
   cPtr->Point :: printme();     // “This is a point”
   pPtr = &c;
   pPtr->printme();              // Point or circle?
}
```

Analysis of Illustration

- Note that in the main() of the previous illustration:
  - a base class pointer, *pPtr*, points to a derived class object, *c*
  - pPtr is used to invoke the method, *printme()*, declared in the base class but overridden in the derived class

- The base class *printme()* is invoked even when *pPtr* points to *c*, an object of derived class Circle; hence
  - pPtr->printme(); prints "This is a point"

- What if we want the derived class function to be called instead? The answer is virtual function as explained next

Virtual Function

Declaring a Virtual Function

- To call the derived class function in our last illustration, you simply need to declare the base class function as a "virtual" function

- The base class will be declared as a virtual function as follows:

```
class Point {
   protected :
  int x, y;
   public :
  Point ();
  Point (int x, int y) ;
  virtual void printme();
};
```

- The rest of the individual method implementations remain the same

Static and Dynamic Binding

- Static Binding
  - When a virtual function is called by referring a specific object via its name and using the dot member selection operator, the reference is resolved during compilation
  - e.g. static binding for a virtual funciton *printme()* and an object *p* will look like *p.printme()*

- Dynamic Binding
  - If a base class pointer pointing to a derived class object is used to invoke a virtual function, say *printme()*, the program will dynamically choose the right *printme()* from the appropriate class

Pure Virtual Function

- What if we do not want any code in the base class methods?
- What if the only reason we want to have a particular method in the base class is to override it in derived classes and then use base class pointers to dynamically call derived class implementations?
- In this case, we can make the base class method a pure virtual function by initializing it to 0 in the declaration, e.g.

```
class Drawable_object
{
  virtual int drawme () = 0;
};
```

Abstract Class

- An abstract class is a class with one or more pure virtual functions
- Note that abstract classes cannot be instantiated
- The idea of abstract class demonstrates polymorphism in object-oriented programming, i.e.
  - the ability for objects of different classes, related by inheritance, to respond differently to the same message
- The virtual functions of abstract classes thus display different “forms” and hence the term polymorphism

Module 9 - Class and Function Templates

- Class Template
- Function Template
- Notes

Class Template

Definition

- Class template is the idea of using the same class for different kinds of data types

- Example
  - Stacks are data structures where the last value stored on the stack will be the first value out of the stack (Last In First Out - LIFO structure)
  - We can now represent a stack as a class that consists of an array of elements, an integer indicating the current top of the stack, and its associated methods such as push and pop

Purpose

- Class templates help us to create stacks of integers, doubles, floats, etc, without having to write the code more than once

- Example of Declaring a Class Template

```
template <class T>
class Stack
{
   T* array;
   int max, top;
   public:
      T pop();
      void push (T element);
      Stack (int max=50);
};
```

type parameter, T

keywords indicating the declaration of a class template

Illustration: Problem

Imagine that you need two classes for two very similar jobs:

a class using *int* for stack operations while another class using *string* for stack operations

```
Class stackInt
{
Int SP;
  Push (int);
  Int pop ();
};
```

```
Class stackString
{
 Int SP;
  Push (string);
  string pop ();
};
```

Illustration: Solution

You can avoid using two different classes in this case by simply applying class template!

A class template *stack* defined with the type parameter *T*

```
template class <T>
Class stack
{
 Int SP;
  Push (T);
  T pop ();
};
```

Since the template simply refers to an array of type T, we can create an actual class by specifying what T is in each declaration below:

```
Stack<int> iStack;
Stack<string> fStack;
```

Multiple Type Parameter

- We can create class templates with more than one type parameter

- Example

```
template <class X, class Y>
class SampleClass
{
   // code
};
```

- In this example, both X and Y must be specified when an object is created, e.g.

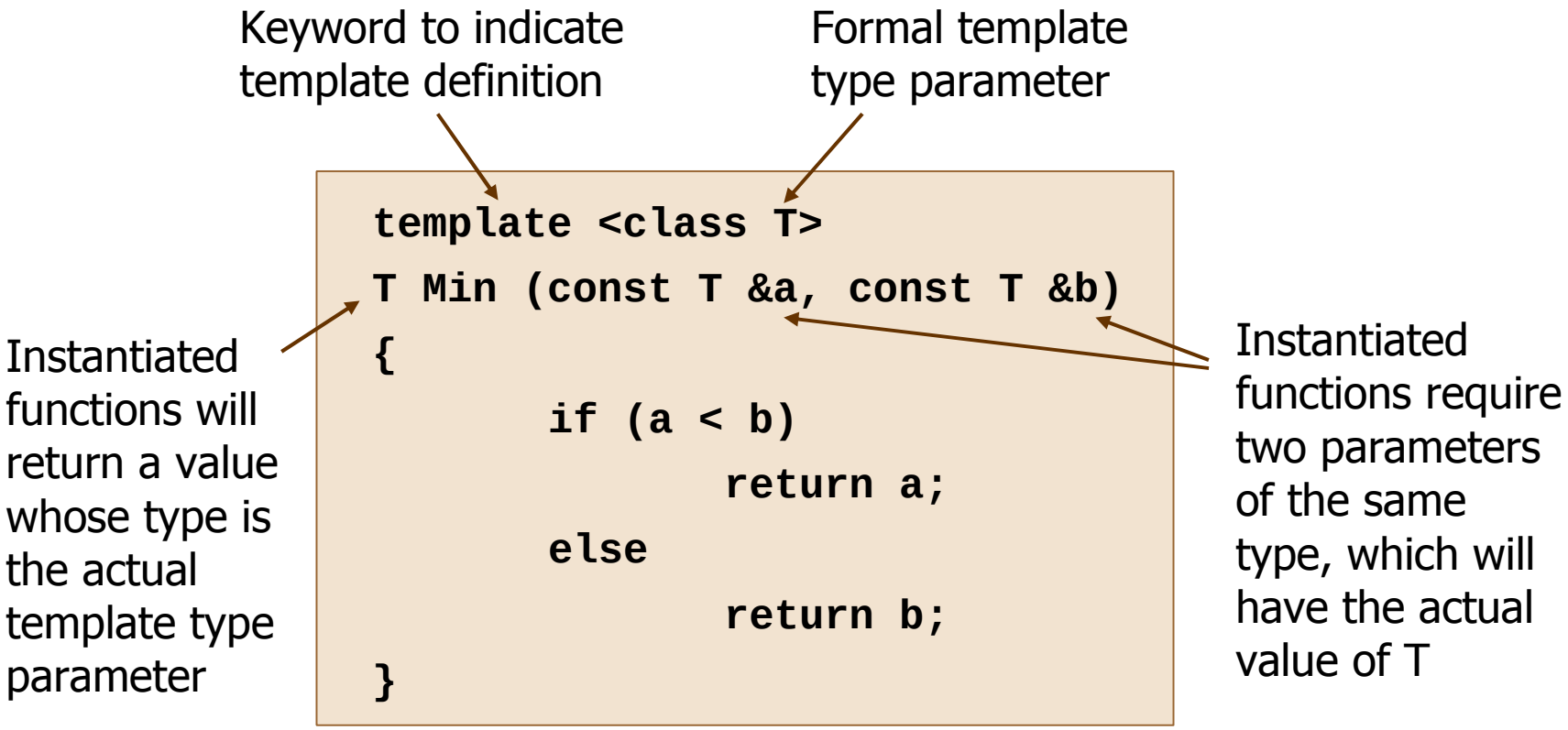
```
SampleClass <int, bool> S;
```

Function Template

Definition

Function template is the idea of using the same function for different types of data

Who is who in a Function Template



Example: General Usage

- We can declare a function template as follows:

```
template <class T>
T square (T x)
{ return x*x; }
```

- We can call the square function for different data types:

```
int a=5, b;
float c=10.99, d;
b = square (a);
d = square (c);
```

Example: Printing an Array

- Consider the example below:

```
template <class T>
void printArray (T* array, int n)
{
for (int j=0; j<n; j++)
    cout << array[j];
}
```

- This template function can be used to print arrays of any data type

- If the array contains user-defined objects, the << operator must be overloaded correctly

Further on printArray …

- What if an overloaded "printArray" function exists?

```
void printArray (char* array, int n);
```

- And there are also extra lines of code to be executed as below?

```
char array[5] = {'a', 'b', 'c', 'd', 'e'};
printArray (array, 5);
```

- Which function will be called? Will there be any ambiguity?

Indeed, there will be no error. To print an array of characters, the *printArray* function specific to char will be called. C++ always looks for a template function as the last solution.

Overloading Template Function

- We can overload template functions

- Example

```
template<class T>
void printArray (T* a, int n);

template<class T>
void printArray (T* a, int lower, int upper);
```

Notes

When to Use Templates

Templates are recommended for:

- expressing algorithms that apply to many argument types
- expressing containers
- replacing inheritance in certain cases because they could do better thanks to their compile-time mechanisms (however, templates cannot add new variants without recompilation)

Lab 3
Finer Aspects of C++

Now, it is time to take a coffee break and work on your third lab!

Notes: Please download lab notes and database from the SPG training center

Module 10 - Advance Features

- Container Class
- Const Function and Object
- Friend Function
- File Operations
- Exception Handling

Container Class

Purpose

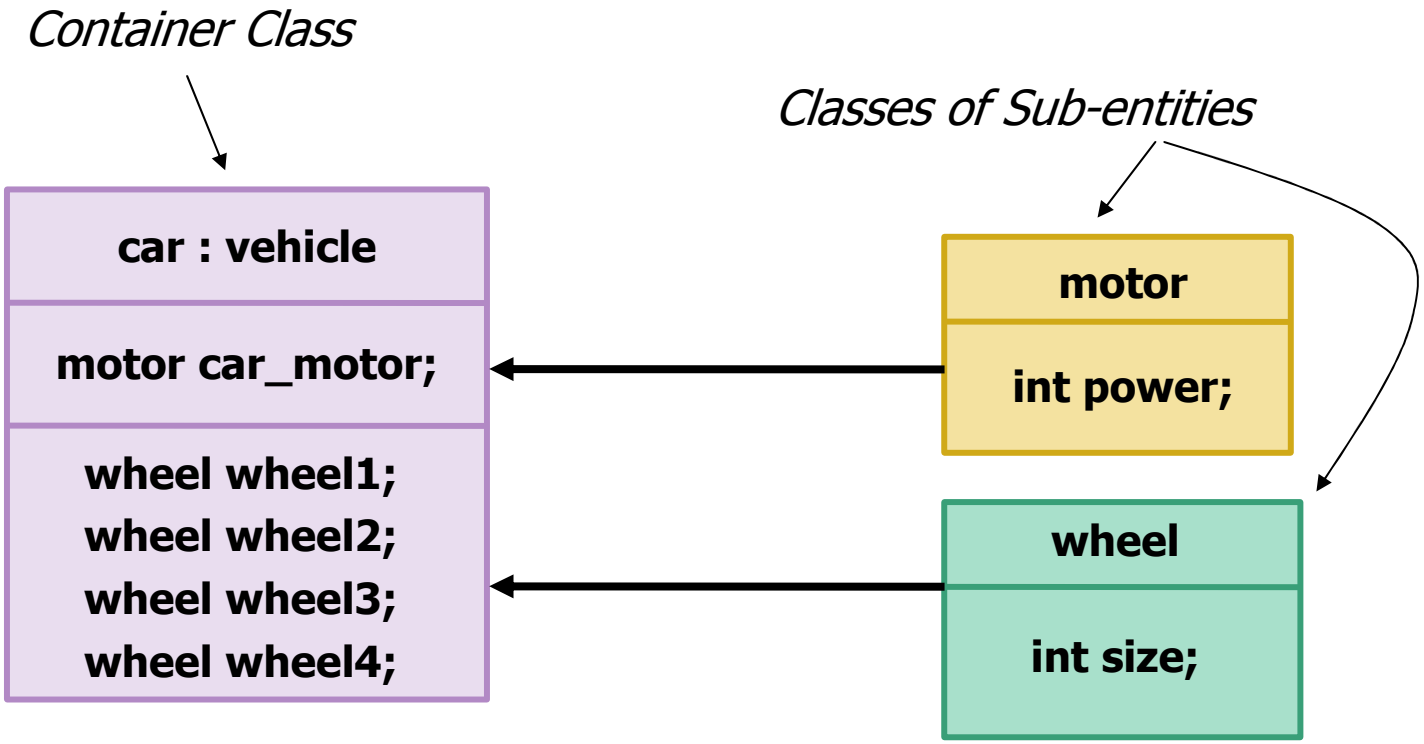
- We often need to model entities comprising several sub-entities with particular behavior, e.g.
  - A car is a vehicle composed of 4 wheels, a motor, etc, which have their own particularities

- In C++, container class can manage this concept

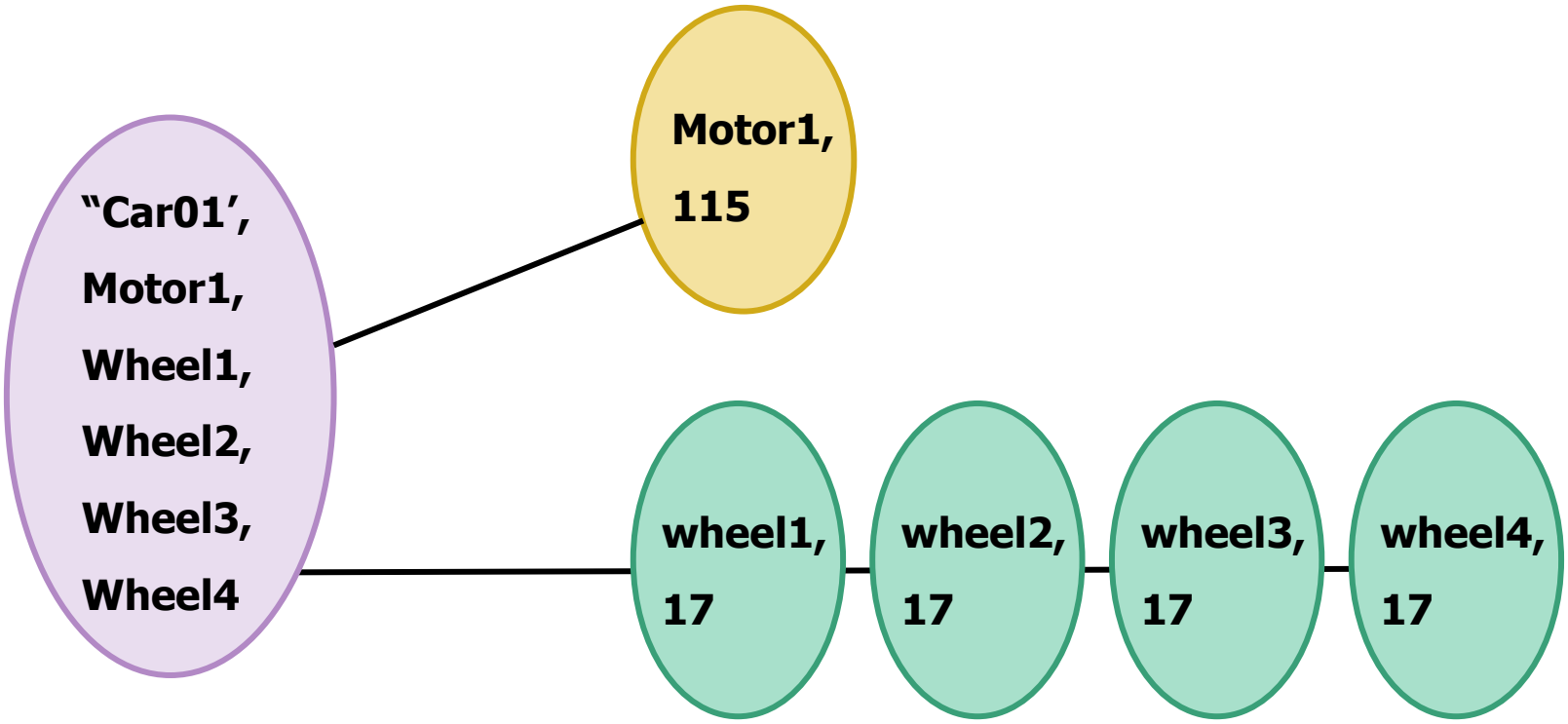
Definition

Container class is a class that can hold objects of other classes

Example: Car Container Class



Example: Car Container Class (cont.)



Const Function and Object

Consider the *Time* Class

```
class Time
{
   private:
      int hh, mm, ss;
   public:
      Time (int Ihh, int Imm, int Iss);
      void getTime() const;
      void setTime(int Chh, int Cmm, int Css);
};
```

Member Functions of *Time* Class

```
Time :: Time(int Ihh, int Imm, int Iss)
{
   hh = Ihh; mm = Imm; ss = Iss;
}

void Time :: getTime() const
{
   cout << hh << " " << mm << " " << ss;
}

void Time :: setTime(int Chh, int Cmm, int Css)
{
   hh = Chh; mm = Cmm; ss = Css;
}
```

How to Handle *const* Objects

```
int main()
{
   const Time noon(12, 0, 0);
   Time now(10, 0, 0);

   noon.setTime(3, 30, 0);    // Not allowed: setTime is not
                              // a const function
   now.setTime(4, 0, 0);      // Allowed: now is not const
   noon.getTime();            // Allowed: getTime is a const
                              // function
}
```

Conclusion

- You cannot call a non-constant function on a constant object
- However, constructors/destructors do not fall under this category
- A constructor can be used to initialize a constant class

Friend Function

What is a Friend Function?

- Friend functions can be considered as “ordinary” global functions, which can access private members of a class

- Friend functions are declared in a class as shown below:

```
class complex{
  double re, im;
public:
  complex (double r, double i) {re=r; im=i;}
  complex (double r) {re=r; im=0;}
  complex() {re=im=0;}

 friend complex operator+(complex,complex);
}
```

File Operations

File Handling Methods

Listed are classes that define file handling methods in C++:

- **ifstream**
  - Provides input operations
  - Contains open(), get(), getline(), read(), seekg(), tellg()
- **ofstream**
  - Provides output operations
  - Contains open(), put(), write(), seekp(), tellp()
- **fstream**
  - Provides support for simultaneous input and output

Open and Close Files

- Opening and closing files:

```
ofstream outfile (“result”);
```

Or

```
ofstream outfile;
outfile.open (“result”);
outfile.close ();
```

Multiple Files

```
#include<iostream.h>
#include<fstream.h>
main(){
   ofstream fout;
   fout.open("country");
   fout << "USA\n";
   fout.close();
   char line[80];
   ifstream fin;
   fin.open("country");
   while(fin){      // check end of file
      fin.getline(line,80);
      cout << line;
   }
   fin.close();
}                   // end of main
```

eof, End of File

- End of file is signified as *eof*
- We can check for end of file by

  `while(fin)` *or* **`if (fin.eof()!=0)`**

- Files can be opened in various file modes, e.g.
  - **`ios::app`** - append to end of file
  - **`ios::in`** - open file for reading only
  - **`ios::out`** - open file for writing only

Example of Modes and I/O on char

```
main()
{
   char string[80];
   cin >> string;
   fstream file;
   file.open (“text”, ios::in|ios::out);
   for (int i=0; i<80; i++)
       file.put(string[i]);  //put a char to file
   file.seekg(0);            //go to beginning of file
   char ch;
   while(file){
       file.get(ch);         //get a character from the file
       cout << ch;
   }
}
```

- *seekg() - moves get pointer to a location*
- *seekp() - moves put pointer to a location*

read() and *write()* as Binary Files

- Just as put() and get(), read() and write() can also read and write from/to a file but in binary form
- These features help to read and write objects or structures to a file
- Example

```
#include<iostream.h>
#include<fstream.h>
class Test{
  int a,b;
  public:
   void get_data();
   void write_data();
};
```

Example of *read()* and *write()*

```
main(){
    float height[4]={12,12,15,17};
    ofstream outfile(filename);
    outfile.write((char *) &height,sizeof(height));
    outfile.close();

    Test test1[3];//array of objects of class test
    fstream file;
    file.open("stock",ios::in|ios::out);
    for (int i=0;i<3;i++){
         test1[i].get_data();
         file.write((char *)&test1[i], sizeof(test1[i]));
    }
    file.seekg(0);
    for (i=0;i<3;i++){
         file.read((char*) &test1[i], sizeof(test1[i]));
         test1[i].write_data();
    }
    file.close();
}
```

Notes: Objects are read and written in binary form

Exception Handling

Purpose

- Exception handling provides means to detect and report an exception such as out of range or overflow (i.e. divided by zero during run time)

- Such mechanism suggests a separate error-handling code that performs the following tasks:
  - Find the problem => Hit the exception
  - Inform about error occurrence => Throw the exception
  - Receive error information => Catch the exception
  - Take corrective actions => Handle the exception

Error Handling Process

- The error handling code has 3 segments: one to detect exception, one to throw exception, one to catch and handle exception

- Example

```
class DivideByZeroException {
  char  message[80];
  public:
   DivideByZeroException() //constructor
   {     strcpy (message, ”Divide by Zero”);   }
   char * what()
   {     return message;    }
};
```

Summary

What You Have Learned

- Class and objects
- Data abstraction and encapsulation
  - Data Abstraction
    - Classes use this concept as they are defined as list of abstract variables, e.g. size, weight, cost, etc, along with methods to operate on these variables
  - Data Encapsulation
    - Encapsulation means wrapping of data and methods into a single class
- Data hiding and access mechanism
- Automatic initialization and clear-up of objects

What You Have Learned (cont.)

- Inheritance
  - The objects of a class acquire the properties of another class
- Polymorphism
  - Different objects have the same external interface but different internal interface
  - DAG
- Dynamic binding
  - The code associated with a given procedure call is not known until the time of its call during runtime
- Dynamic initialization of variables

Thank you for your attention!

You have successfully completed the training of "Basic Concepts of C++".

Should you have any questions or comments, send us an e-mail at spg@st.com