Introduction to Classes and Objects

Outline

- **How to create classes and objects**
- **How to define member functions**
- **How to define constructor member functions**
- **How to define reader and writer member functions**
- **How to benefit from data abstraction**
- **How to protect member variables from harmful accesses**

How To Create Classes and Objects

- Let's first define and use classes as structures

- Classes correspond to naturally occurring categories
  - Enable you to describe and manipulate bundles of descriptive data items for categories with a single name
    - E.g., student, car, ….
    - E.g., Student has number, name, GPA... Car has speed, gear,…
  - Define a class once, you can construct any number of class objects that belong to that class
    - E.g., student objects, car objects

How To Create Classes and Objects

From programming languages perspectives,

- A class is a data type
  - User-defined data type (compared to built-in data types)
  - Class includes member variables for descriptive data items

```
class box_car {
      public:
         double height, width, length;
};

class tank_car {
              public:
                    double radious, length;
};
```

Box Car and Tank Car

Box car

Tank car

Pictures are from naver.com 이미지 검색

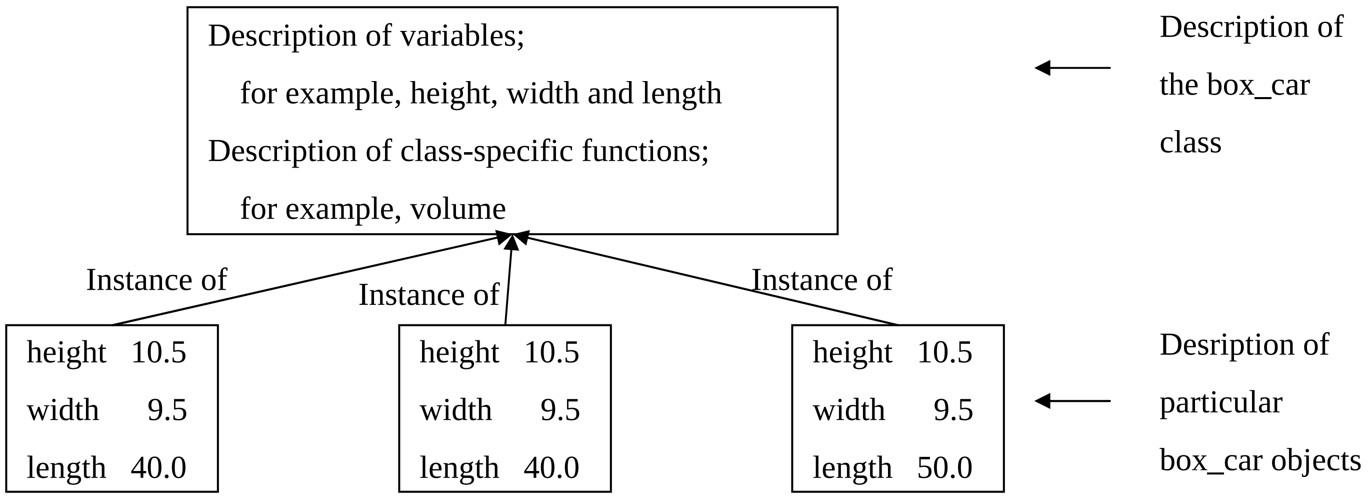
How To Create Classes and Objects

- Objects are data items of that type
  - Objects are created by declaring variables or dynamic allocation

```
box_car x;
box_car *y = new (box_car);
tank_car z;
tank_car *w = new (tank_car);
```

Example of Creating a Class

- Describe data items that mirror real-world categories




```
class box_car {
    public:
        double height, width, length;
};
```

Class Objects and Member Variables

- Once a class is defined, we can create variables of that class, as we define variables of built-in types
  - E.g., `box_car x, y;`
- Member variables
  - Variables that appear inside class definitions (AKA, fields)
  - Refer to a member variable via the **class-member-operator .**
  - Use
    - class object's name . member-variable name
  - Assignment
    - class object's name . member-variable name = expression ;
  - Member variables are used as regular variables (e.g., parameters)

```
#include <iostream.h>

class box_car {    // Tells C++ that a class “box_car” is to be defined
  public:          // Specifies where variables can be referenced
    double height, width, length;    // Introduces variables
};                 // end class box_car
// Calculate the volume of a box
double box_car_volume(double h, double w, double l) {
  return h * w * l;
 }  // end function box_car_volume

int main() {
  box_car x;
  x.height = 10.5; x.width = 9.5; x.length = 40.0;
  cout << “The volume of the box_car is “
       << box_car_volume(x.height, x.width, x.length)
       << endl;
}  // end function main
```

```
The volume of the box_car is 3990
```

Passing Class Object as Function Argument

- You can pass class object as function argument
  - Instead of passing each member variables of class as function argument, you can takes just one argument

- You can overload functions with different class object arguments

```
#include <iostream.h>

class box_car {    // Tells C++ that a class “box_car” is to be defined
  public:          // Specifies where variables can be referenced
    double height, width, length;    // Introduces variables
};                 // end class box_car
// Calculate the volume of a box
double volume(box_car b) {
  return b.height * b.width * b.length;
 }

int main() {
  box_car x;
  x.height = 10.5; x.width = 9.5; x.length = 40.0;
  cout << “The volume of the box_car is “
       << volume(x) << endl;
}  // end function main
```

```
The volume of the box_car is 3990
```

```
#include <iostream.h>

 const double pi = 3.14159;
 class box_car {public: double height, width, length;};
 class tank_car {public: double radious, length;};
 // Calculate the volume of a box
 double volume(box_car b) {
   return b.height * b.width * b.length;
 }
 // Calculate the volume of a tank
  double volume(tank_car t) {
    return pi * t.radius * t.length;
  }
  int main() {
    box_car x; x.height = 10.5; x.width = 9.5; x.length = 40.0;
    tank_car y; y.radius = 3.5; y.length = 40.0;
    cout << “The volume of the box car is “ << volume(x) << endl
          << “The volume of the tank car is “ << volume(y) << endl;
  } // end function main
```



```
The volume of the box car is 3990
The volume of the tank car is 1539.38
```

Member Function

- Define functions into the class definition, like member variables

```
class box_car {
    public: double height, width, length
            double volume() {
                    return height * width * length;
            }
}
```

  - Calling member function is different from ordinary function calls
    - E.g., `box_car x;` …..; `x.volume();`
  - Member function has one special argument
    - Class object that belong to the same class which does not appear in parenthesis
  - In member functions, all member variables are taken to belong to the special, class object argument
    - When `x.volume()` is called, height, width, and length mean those of x

```
#include <iostream.h>
const double pi = 3.14159;
class box_car {
  public: double height, width, length;
  // Calculate the volume of a box
  double volume() {
    return height * width * length;
  }
};
class tank_car {
    public: double radious, length;
    // Calculate the volume of a tank
    double volume() {
      return pi * radius * length;
    }
};
int main() {
  box_car x; x.height = 10.5; x.width = 9.5; x.length = 40.0;
  tank_car y; y.radius = 3.5; y.length = 40.0;
  cout << “The volume of the box car is “ << x.volume() << endl
        << “The volume of the tank car is “ << y.volume() << endl;
} // end function main
```

```
The volume of the box car is 3990
The volume of the tank car is 1539.38
```

Member function arguments

- Member function can also have ordinary arguments
  - Example
    - Usage : `x.scaled_volume(0.95)`
    - Definition

```
class box_car {
  public: double height, width, length;
          double scaled_volume (double scale_factor)
          {
           return scaled_factor*height*width*length;
          }
  }
```

Member Function Prototype

- Function prototype of member functions
  - Like function definition without a body
  - Define the function outside of the class definition
  - Example)

```
class box_car {
    public: double height, width, length;
    double volume(); // prototype of member function
};
// definition of member function
double box_car::volume() {
    return height * width * length;
}
```

Constructors

Special member functions that are called when class objects are created

- Enable you to initialize the member variables in new class objects
- Default Constructor
  - Called automatically whenever a new class object is created
  - Function's name is the same as the name of the class
    - E.g., `tank_car(), box_car()`
  - No return-value data type
  - Cannot have a parameter

```
#include <iostream.h>
 const double pi = 3.14159;
class tank_car {
    public: double radious, length;
    // default constructor;
    tank_car() {radius = 3.5; length = 40.0;}
    // Calculate the volume of a tank
    double volume() {
      return pi * radius * length;
     }
 };
 int main() {
   tank_car t;
   cout << “The volume of the tank car is “ << t.volume() << endl;
 } // end function main
```

```
The volume of the tank car is 1539.38
```

Constructor with Parameters

- You can also define a constructor with parameters

```
class tank_car {
 public: double radius, length;
         // Default constructor
         tank_car() { radius = 3.5; length = 40.0;}
         // Constructor with two parameters;
         tank_car (double r, double l) {
             radius = r; length = l;
         }
         // volume function:
         double volume() {return pi*radius*radius*length;}
};
```

- Different constructors are called on variable declaration
  - `tank_car x, y(3.0, 4.0);`
  - Actually, constructors are overloaded

```
#include <iostream.h>
 const double pi = 3.14159;
class tank_car {
     public: double radious, length;
     // default constructor;
     tank_car() {radius = 3.5; length = 40.0;}
     // Constructor with two parameters:
     tank_car(double r, double l) { radius = r; length = l;}
     // Calculate the volume of a tank
      double volume() {
        return pi * radius * length;
      }
 };
 int main() {
   tank_car t1;
   tank_car t2(3.5, 50.0);
   cout << “The volume of the default tank car is “
        << t1.volume()
        << endl
        << “The volume of the specified tank car is “
        << t2.volume()
        << endl;
 } // end function main
```

```
The volume of the default tank car is 1539.38
The volume of the specified tank car is 1924.22
```

Outline

Reader and Writer Member Functions

- Also referred to as getter and setter functions
  - Not a part of the C++ language
  - Convention to access the member variables indirectly by defining a member function to access each member variable
    - e.g., double read_radius() {return radius;}
      double write_radius(double r) {radius = r;}
  - Why do we have readers and writers, while we can access member variables directly?
    - e.g., cout << x.radius;
      x.radius = r;
  - This is for practicing data abstraction or encapsulation

Reader Functions

- Extract information from an object
  - Example)

```
class tank_car {
  public:
      double radius, length;
      tank_car() { radius = 3.5; length = 40.0;}
      tank_car(double r, double l) {radius=r; length=l;}
      double read_radius() {return radius;}
      double volume() {return pi*radius*radius*length;}
}
```

  - Usage: `tank_car t; t.read_radius();`

Reader Functions

- Can include additional computation
  - Example

```
class tank_car {
 public:
   double radius, length;
   tank_car() { radius = 3.5; length = 40.0;}
   tank_car(double r, double l) {radius=r; length=l;}
   double read_radius() {
      cout << “Reading a tank_car’s radius …” << endl;
      return radius;
   }
   double volume() {return pi*radius * radius *length;}
}
```

Imaginary Member Variables

- Can provide access to imaginary value that can extract from member variables
  - Example: when there are access requests to diameters

```
class tank_car {
 public:
   double radius, length;
   tank_car() { radius = 3.5; length = 40.0;}
   tank_car(double r, double l) {radius=r; length=l;}
   double read_radius() {return radius;}
   double read_diameter() { return radius * 2.0; }
   double volume() {return pi*radius*radius*length;}
}
```

  - Usage: `tank_car t; cout << t.read_diameter();`

Writer Functions

Assign a member-variable value indirectly.

- Insert information into an object
- Example)

```
class tank_car {
  public:
   double radius, length;
   tank_car() { radius = 3.5; length = 40.0;}
   tank_car(double r, double l) {radius=r; length=l;}
   void  write_radius(double r) {radius = r;}
   double volume() {return pi*radius*radius*length;}
}
```

- Usage: `tank_car t; t.write_radius(4.0)`

Imaginary Member Variables

- Can provide access to imaginary value that can extract from member variables.
  - Example

```
class tank_car {
  public:
   double radius, length;
   tank_car() { radius = 3.5; length = 40.0;}
   tank_car(double r, double l) {radius=r; length=l;}
   void  write_radius(double r) {radius = r;}
   void  write_diameter(double d) {radius = d/2.0;}
   double volume() {return pi*radius*radius*length;}
}
```

  - Usage: `tank_car t; t.write_diameter(8.0)`

Encapsulation, Data Abstraction, Information Hiding

- Object-oriented programming encapsulates data (states) and functions (behaviors) into *class* packages
  - A class is like a blueprint
  - Out of a class, one can create objects
- When you move implementation details into access functions and when users of the class access only thru those functions, you are practicing data abstraction
  - Constructors, readers, writers help practicing data abstraction
  - Data abstraction allows easier update for implementation details

Data Abstraction Benefit Example

```
class tank_car {
    public:
    double radius, length;

    tank_car() {radius = 3.5; length = 40.0;}
    tank_car(double r, double l) {radius = r; length = l;}

    double read_radius() {return radius;}
    void write_radius(double r) { radius = r; }
    double read_diameter() { return 2.0 * radius; }
    void write_diameter(double d) { radius = d / 2.0; }
    double read_length() {return length; }
    void write_length(double l) { length = l; }

    double volume() {return pi * radius * radius * length;}
};
```

- If we find that `read/write_diameter()` is called more often than `read/write_radius()`, what can we do?

Data Abstraction Benefit Example

- Change Implementations
  - Change member variable "`radius`" to "`diameter`"
  - Your member function implementations need also to be changed

```
class tank_car {
    public:
    double diameter, length;

    tank_car() {diameter = 7.0; length = 40.0;}
    tank_car(double r, double l) {diameter=r*2.0; length=l;}

    double read_radius() {return diameter / 2.0;}
    void write_radius(double r) { diameter = r * 2.0; }
    double read_diameter() { return diameter; }
    void write_diameter(double d) { diameter = d; }
    double read_length() {return length; }
    void write_length(double l) { length = l; }

    double volume() {return .25 *pi*diameter*diameter*length;}
};
```

Data Abstraction Benefit Example

- How do the user sides need to be changed?
  - When you practiced data abstraction, no change

```
… t.read_diameter()  => … t.read_diameter()
```

  - If you did not, you must change all the used places

```
…. t.radius … => … 0.5 * t.diameter …
```

- Data abstraction leads to information hiding
  - Class objects can communicate with one another via well-defined interfaces (which are member functions, not member variables), but do not know (should not know) how each class is implemented
  - Making programs easier to maintain
    - Your programs become easier to reuse.
    - You can easily augment what a class provides
    - You can easily improve the way data are stored

Protecting Member Variables

- You can practice data abstraction by providing access functions in your class definition and by encouraging users to use them
  - Obviously, this alone cannot prevent access to implementation details
    - i.e., direct access to member variables
  - Is there any way to prevent direct accesses to member variables?

- Protection of member variables from harmful references.
  - Marked with the "`private`" symbol for protection
  - Cannot access member variables via the class-member operator

Example of Protecting Member Variables

```
class tank_car {
    public:
        tank_car() {radius = 3.5; length = 40.0;}
        tank_car(double r, double l) {radius=r; length=l;}

        double read_radius() {return radius;}
        void write_radius(double r) { radius = r; }
        double read_diameter() { return 2.0 * radius; }
        void write_diameter(double d) { radius = d / 2.0; }
        double read_length() {return length; }
        void write_length(double l) { length = l; }

        double volume() {return pi*radius*radius*length;}

    private:
       double radius, length;
};
```

Effect of Employing Private Member Variables

- Reference to an object's member variables
  - Attempt to refer to a object's member-variable values via the class-member operator fail to compile.
    - `t.radius` : Evaluation fails to compile
    - `t.radius` : Assignment fails to compile
  - Reference and assignment via member functions located in the public part of the class definition are still allowed
    - `t.read_radius()` : evaluation compiles
    - `t.write_radius(6)` : assignment compiles
- Member variables and functions in private section cannot be accessed directly from outside of the class
  - They can still be accessible inside the class definition