Objects and Classes

Sai-Keung Wong

National Yang Ming Chiao Tung University

Hsinchu, Taiwan

Intended Learning Outcomes

- Name the features of a class
- Identify the scopes of variables (local or global)
- Distinguish between accessors and mutators
- List the features of class abstraction and encapsulation
- List the benefits of class abstraction and encapsulation

Content

- Classes vs objects
- State, behavours
- Unified Modeling Language
- Constructors
- Constant object names
- Anonymous objects
- Class is a data type
- Class definition and implementation
- Preventing multiple Inclusions
- Scopes
- Attribute modifiers: public, protected, private

Design a class ROBOT

- A robot can change its **position**.
- A robot can change its **orientation**.
- A robot can change its **viewing direction**.

Design a class ROBOT

- A robot can change its **position**.
- A robot can change its **orientation**.
- A robot can change its **viewing direction**.

Object-Oriented Programming (OOP) Concepts

```
class ROBOT {
  protected:
      Vector3 pos;          // position
      Vector3 dir;          // movement direction
      Vector3 speed;
  public:
      ROBOT( );
      void stop( );
      void walkForward( );
      void backward( );
      void setDirection(const Vector3 & dir);
      void setPosition(const Vector3 & pos);
      void setSpeed( const Vector3 & speed);
      void update( double dt ); // simulation step size
};
```

Object-Oriented Programming (OOP) Concepts

```
ROBOT r1, r2; // create two ROBOT objects

ROBOT *robotA = nullptr; // one pointer
// instantiate a robot object
robotA = new ROBOT;

ROBOT *robots[NUM]; // an array of pointers
for (int I = 0; i < NUM; ++I ) {
   robots[ i ] = new ROBOT;
}
```

```
class ROBOT {
  protected:
      Vector3 pos;          // position
      Vector3 dir;          // movement direction
      Vector3 speed;
  public:
      ROBOT( );
      void stop( );
      void walkForward( );
      void backward( );
      void setDirection(const Vector3 & dir);
      void setPosition(const Vector3 & pos);
      void setSpeed( const Vector3 & speed);
      void update( double dt ); // simulation step size
};
```

Object-Oriented Programming (OOP) Concepts

```
ROBOT r1, r2; // create two ROBOT objects

ROBOT *robotA = nullptr; // one pointer
// instantiate a robot object
robotA = new ROBOT;

ROBOT *robots[NUM]; // an array of pointers
for (int I = 0; i < NUM; ++I ) {
   robots[ i ] = new ROBOT;
}
```

- An object has a A1 identity, state, and a set of A2 (implemented as A3).

```
class ROBOT {
  protected:
      Vector3 pos;          // position
      Vector3 dir;          // movement direction
      Vector3 speed;
  public:
      ROBOT( );
      void stop( );
      void walkForward( );
      void backward( );
      void setDirection(const Vector3 & dir);
      void setPosition(const Vector3 & pos);
      void setSpeed( const Vector3 & speed);
      void update( double dt ); // simulation step size
};
```

Object-Oriented Programming (OOP) Concepts

```
ROBOT r1, r2; // create two ROBOT objects

ROBOT *robotA = nullptr; // one pointer
// instantiate a robot object
robotA = new ROBOT;

ROBOT *robots[NUM]; // an array of pointers
for (int I = 0; i < NUM; ++I ) {
   robots[ i ] = new ROBOT;
}
```

- An object has a unique identity, state, and a set of behaviors (implemented as methods).
- Objects can be A1 identified.

```
class ROBOT {
  protected:
      Vector3 pos;          // position
      Vector3 dir;          // movement direction
      Vector3 speed;
  public:
      ROBOT( );
      void stop( );
      void walkForward( );
      void backward( );
      void setDirection(const Vector3 & dir);
      void setPosition(const Vector3 & pos);
      void setSpeed( const Vector3 & speed);
      void update( double dt ); // simulation step size
};
```

Object-Oriented Programming (OOP) Concepts

```
ROBOT r1, r2; // create two ROBOT objects

ROBOT *robotA = nullptr; // one pointer
// instantiate a robot object
robotA = new ROBOT;

ROBOT *robots[NUM]; // an array of pointers
for (int I = 0; i < NUM; ++I ) {
   robots[ i ] = new ROBOT;
}
```

- An object has a unique identity, state, and a set of behaviors (implemented as methods).
- Objects can be distinctly identified.

```
class ROBOT {
  protected:
      Vector3 pos;          // position
      Vector3 dir;          // movement direction
      Vector3 speed;
  public:
      ROBOT( );
      void stop( );
      void walkForward( );
      void backward( );
      void setDirection(const Vector3 & dir);
      void setPosition(const Vector3 & pos);
      void setSpeed( const Vector3 & speed);
      void update( double dt ); // simulation step size
};
```

Object-Oriented Programming (OOP) Concepts

```
class ROBOT {
  protected:
      Vector3 pos;          // position
      Vector3 dir;          // movement direction
      Vector3 speed;
  public:
      ROBOT( );
      void stop( );
      void walkForward( );
      void backward( );
      void setDirection(const Vector3 & dir);
      void setPosition(const Vector3 & pos);
      void setSpeed( const Vector3 & speed);
      void update( double dt ); // simulation step size
};
```

Object-Oriented Programming (OOP) Concepts

ROBOT r1, r2; // create two ROBOT objects

```
r1 {
  pos = Vector3( 1, 1, 0)
  dir = Vector3( 0, 1, 0)
  speed = 1.7
}

r2 {
  pos = Vector3(0, 0, 0)
  dir = Vector3(1, 0, 0)
  speed = 1.7
}
```

State: A set of data fields (or properties) with their current values

State of object r2

```
class ROBOT {
  protected:
      Vector3 pos;          // position
      Vector3 dir;          // movement direction
      Vector3 speed;
  public:
      ROBOT( );
      void stop( );
      void walkForward( );
      void backward( );
      void setDirection(const Vector3 & dir);
      void setPosition(const Vector3 & pos);
      void setSpeed( const Vector3 & speed);
      void update( double dt ); // simulation step size
};
```

Object-Oriented Programming (OOP) Concepts

ROBOT r1, r2; // create two ROBOT objects

```
r1 {
  pos = Vector3( 1, 1, 0)
  dir = Vector3( 0, 1, 0)
  speed = 1.7
}

r2 {
  pos = Vector3(0, 0, 0)
  dir = Vector3(1, 0, 0)
  speed = 1.7
}
```

State: A set of data fields (or properties) with their current values

State of object r2

The state of an object informs what can be performed A1

```
class ROBOT {
  protected:
      Vector3 pos;          // position
      Vector3 dir;          // movement direction
      Vector3 speed;
  public:
      ROBOT( );
      void stop( );
      void walkForward( );
      void backward( );
      void setDirection(const Vector3 & dir);
      void setPosition(const Vector3 & pos);
      void setSpeed( const Vector3 & speed);
      void update( double dt ); // simulation step size
};
```

Object-Oriented Programming (OOP) Concepts

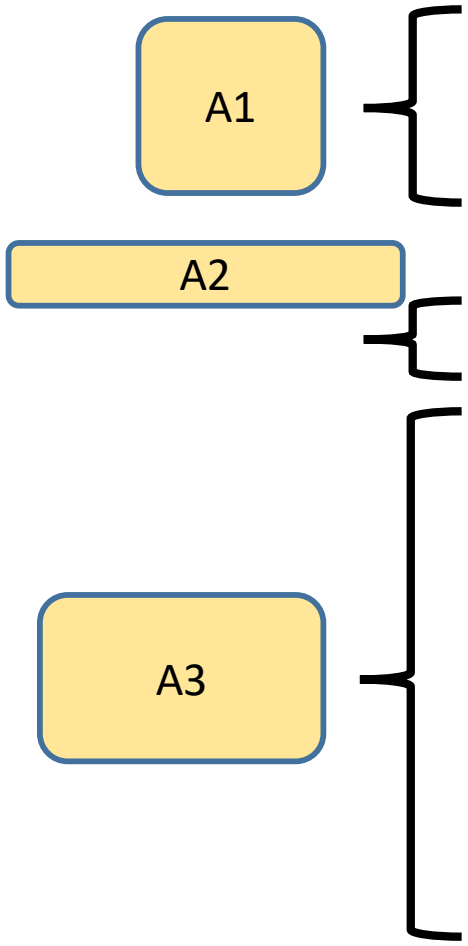
ROBOT r1, r2; // create two ROBOT objects

```
r1 {
  pos = Vector3( 1, 1, 0)
  dir = Vector3( 0, 1, 0)
  speed = 1.7
}

r2 {
  pos = Vector3(0, 0, 0)
  dir = Vector3(1, 0, 0)
  speed = 1.7
}
```

Behavior: a set of functions (methods)

```
class ROBOT {
  protected:
      Vector3 pos;          // position
      Vector3 dir;          // movement direction
      Vector3 speed;
  public:
      ROBOT( );
      void stop( );
      void walkForward( );
      void backward( );
      void setDirection(const Vector3 & dir);
      void setPosition(const Vector3 & pos);
      void setSpeed( const Vector3 & speed);
      void update( double dt ); // simulation step size
};
```

Objects




```
class ROBOT {
  protected:
      Vector3 pos;          // position
      Vector3 dir;          // movement direction
      Vector3 speed;
  public:
      ROBOT( );
      void stop( );
      void walkForward( );
      void backward( );
      void setDirection(const Vector3 & dir);
      void setPosition(const Vector3 & pos);
      void setSpeed( const Vector3 & speed);
      void update( double dt ); // simulation step size
};
```

Objects

Data Fields

Constructor(s)

Functions (methods)

```
class ROBOT {
  protected:
      Vector3 pos;          // position
      Vector3 dir;          // movement direction
      Vector3 speed;
  public:
      ROBOT( );
      void stop( );
      void walkForward( );
      void backward( );
      void setDirection(const Vector3 & dir);
      void setPosition(const Vector3 & pos);
      void setSpeed( const Vector3 & speed);
      void update( double dt ); // simulation step size
};
```

Objects

```
class ClassName {
        Data Fields
        Constructors
        Functions
};
```

Data Fields

Constructor(s)

Functions (methods)

```
class ROBOT {
  protected:
      Vector3 pos;          // position
      Vector3 dir;          // movement direction
      Vector3 speed;
  public:
      ROBOT( );
      void stop( );
      void walkForward( );
      void backward( );
      void setDirection(const Vector3 & dir);
      void setPosition(const Vector3 & pos);
      void setSpeed( const Vector3 & speed);
      void update( double dt ); // simulation step size
};
```

Objects

```
class ClassName {
        Data Fields
        Constructors
        Functions
};
```

Objects

```
class ClassName {
        Data Fields
        Constructors
        Functions
};
```

```
class Circle {
public:
   Circle( ) { radius = 0.0; }      //Constructor
   double getArea( ) const;         //Function
   double radius;                   //Data Field(s)
};
```

Circle Object1
Data field:
Object1.radius = 13.0

Circle Object2
Data field:
Object2.radius = 12.0

Circle Object3
Data field:
Object3.radius = 15.0

Objects

```
class ClassName {
        Data Fields
        Constructors
        Functions
};
```

```
class Circle {
public:
   Circle( ) { radius = 0.0; }      //Constructor
   double getArea( ) const;         //Function
   double radius;                   //Data Field(s)
};
```

Circle Object1
Data field:
Object1.radius = 13.0

Circle Object2
Data field:
Object2.radius = 12.0

Circle Object3
Data field:
Object3.radius = 15.0

State of a circle: A1 A2 of the circle

Behavior of a circle: A3

Objects

```
class ClassName {
        Data Fields
        Constructors
        Functions
};
```

```
class Circle {
public:
   Circle( ) { radius = 0.0; }      //Constructor
   double getArea( ) const;         //Function
   double radius;                   //Data Field(s)
};
```

Circle Object1
Data field:
Object1.radius = 13.0

Circle Object2
Data field:
Object2.radius = 12.0

Circle Object3
Data field:
Object3.radius = 15.0

State of a circle: **radius value** of the circle

Behavior of a circle: getArea

Objects

- Classes are constructs that define objects of the same type.
- Data fields: variables
- Behaviors: functions
- Constructors: they are invoked to construct objects from the class.

```
class Circle {
public:
   Circle( ) { radius = 0.0; }      //Constructor
   double getArea( ) const;         //Function
   double radius;                   //Data Field(s)
};
```

```
Circle Object2
Data field:
Object2.radius = 12.0
```

```
Circle Object3
Data field:
Object3.radius = 15.0
```

State of a circle: **radius** **value** of the circle

Behavior of a circle: getArea

Objects

- Classes are constructs that define objects of the same type.
- Data fields: variables
- Behaviors: functions
- Constructors: they are invoked to construct objects from the class.

```
class Circle {
public:
   Circle( ) { radius = 0.0; }      //Constructor
   double getArea( ) const;         //Function
   double radius;                   //Data Field(s)
};
```

Circle Object2
Data field:
Object2.radius = 12.0

Circle Object3
Data field:
Object3.radius = 15.0

State of a circle: **radius** **value** of the circle

Behavior of a circle: getArea

The state of an object is the set of the data field

A1

Objects

- Classes are constructs that define objects of the same type.
- Data fields: variables
- Behaviors: functions
- Constructors: they are invoked to construct objects from the class.

```
class Circle {
public:
   Circle( ) { radius = 0.0; }      //Constructor
   double getArea( ) const;         //Function
   double radius;                   //Data Field(s)
};
```

Circle Object2
Data field:
Object2.radius = 12.0

Circle Object3
Data field:
Object3.radius = 15.0

State of a circle: **radius** **value** of the circle

Behavior of a circle: getArea

The state of an object is the set of the data field values.

Objects

- Classes are constructs that define objects of the same type.
  - Data fields: variables
  - Behaviors: functions
  - Constructors: they are invoked to construct objects from the class.

```
class Circle {
public:
   Circle( ) { radius = 0.0; }      //Constructor
   double getArea( ) const;         //Function
   double radius;                   //Data Field(s)
};
```

Circle Object2

Circle Object3

State of a circle: **radius** **value** of the circle

Behavior of a circle: getArea

Objects

- Classes are constructs that define objects of the same type.
- Data fields: variables
- Behaviors: functions
- Constructors: they are invoked to construct objects from the class.

```
class Circle {
public:
   Circle( ) { radius = 0.0; }      //Constructor
   double getArea( ) const;         //Function
   double radius;                   //Data Field(s)
};
```

Circle Object2

Circle Object3

State of a circle: **radius** **value** of the circle

Behavior of a circle: getArea

Objects

- Classes are constructs that define objects of the same type.
- Data fields: variables
- Behaviors: functions
- Constructors: they are invoked to construct objects from the class.

```
class Circle {
public:
   Circle( ) { radius = 0.0; }      //Constructor
   double getArea( ) const;         //Function
   double radius;                   //Data Field(s)
};
```

Circle Object2

Circle Object3

State of a circle: **radius** **value** of the circle

Behavior of a circle: getArea

Objects

- Classes are constructs that define objects of the same type.
- Data fields: variables
- Behaviors: functions
- Constructors: they are invoked to construct objects from the class.

```
class Circle {
public:
  Circle( ) { radius = 0.0; }      //Constructor
  double getArea( ) const;         //Function
  double radius;                   //Data Field(s)
};
```

Circle Object2

Circle Object3

State of a circle: **radius** **value** of the circle

Behavior of a circle: getArea

Classes

```
class Circle {
public:
    double radius;                          //Data Fields. Variables
public:
  Circle( ) {                               //Constructor. To construct an object
            radius = 1.0;
    }
    Circle( double newRadius ) {            //Constructor. To construct an object
            radius = newRadius;
    }
    double getArea( ) const {               // behavior
        return radius*radius*3.141592654;
   }
};

Circle a, b(12.5), c(21);                   // a, b, and c are of the same type.
```

After a, b, and c are constructed, **what are their states**?

a.radius =?
b.radius = ?
c.radius = ?

Classes

```
class Circle {
public:
    double radius;                          //Data Fields. Variables
public:
  Circle( ) {                               //Constructor. To construct an object
            radius = 1.0;
    }
    Circle( double newRadius ) {            //Constructor. To construct an object
            radius = newRadius;
    }
    double getArea( ) const {               // behavior
        return radius*radius*3.141592654;
   }
};

Circle a, b(12.5), c(21);                   // a, b, and c are of the same type.
```

The state of an object is the set of the data field **values**.

After a, b, and c are constructed, **what are their states**?

a.radius = A1

b.radius = A2

c.radius = A3

Classes

```
class Circle {
public:
    double radius;                          //Data Fields. Variables
public:
  Circle( ) {                               //Constructor. To construct an object
            radius = 1.0;
    }
    Circle( double newRadius ) {            //Constructor. To construct an object
            radius = newRadius;
    }
    double getArea( ) const {               // behavior
        return radius*radius*3.141592654;
   }
};

Circle a, b(12.5), c(21);                   // a, b, and c are of the same type.
```

To invoke a function:

obj.funcName

a.getArea( );

b.getArea( );

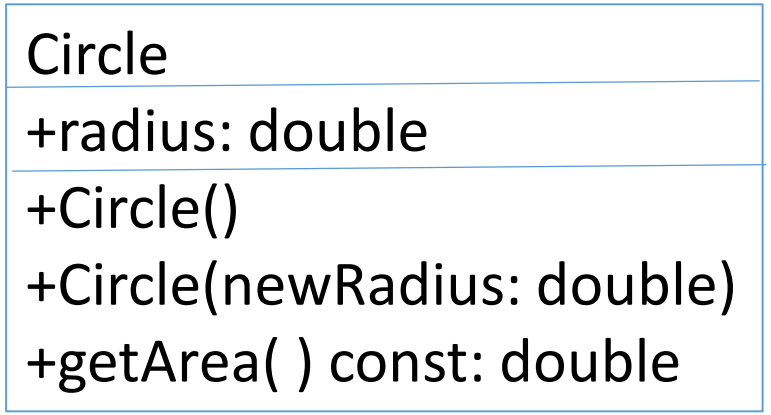
c.getArea( );

The access operator:

the dot operator (.)

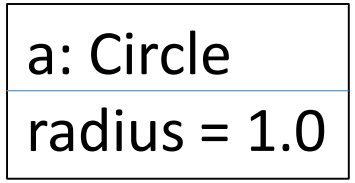
UML Class Diagram

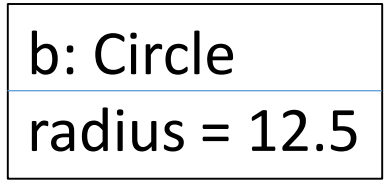
```
class Circle {
public:
    double radius;
public:
    Circle( ) {
            radius = 1.0;
    }
    Circle( double newRadius ) {
            radius = newRadius;
    }
    double getArea( ) const {
        return radius*radius*3.141592654;
    }
};

Circle a, b(12.5), c(21);
```

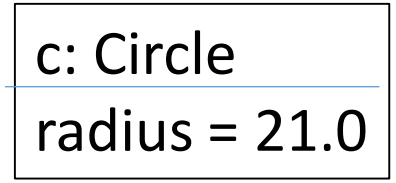


\+ : means public

\- : means protected or private







UML: Unified Modeling Language

Constructors

```
class Circle {
public:
    double radius;
public:
    Circle( ) {                    // no-arg constructor
            radius = 1.0;
    }
    Circle( double newRadius ) {   // constructor
            radius = newRadius;
    }
    double getArea( ) const {
        return radius*radius*3.141592654;
   }
};

Circle a, b(12.5), c(21);
```

- The constructor has the same name as the class name.
- A no-arg or no-argument constructor does not have any arguments
- Constructors can be **overloaded**. They have the same name but with different signatures. They initialize data members. They do not return a value (no void).

Constructors

```
class Circle {
public:
    double radius;
public:
    Circle( ) {                    // no-arg constructor
        radius = 1.0;
    }
    Circle( double newRadius ) {   // constructor
        radius = newRadius;
    }
    double getArea( ) const {
        return radius*radius*3.141592654;
    }
};

Circle a, b(12.5), c(21);
```

- The constructor has the same name as the class name.
- A no-arg or no-argument constructor does not have any arguments
- Constructors can be **overloaded**. They have the same name but with different signatures. They initialize data members. They do not return a value (no void).
- If **a class does not have any constructors**, **a default constructor** is implicitly declared (by compiler). It is a no-arg constructor with an empty body.

Parameters

```
class Circle {
public:
    double radius;
public:
    Circle( ) {
            radius = 1.0;
    }
    Circle( double radius) { // newRadius
            this->radius = radius;
    }
    double getArea( ) const {
        return radius*radius*3.141592654;
   }
};

Circle a, b(12.5), c(21);
```

Circle
+radius: double
+Circle()
+Circle(**radius**: double)
+getArea( ) const: double

← Class name
← Data fields
← Constructors and Functions

+ : means public
- : means protected or private

a: Circle
radius = 1.0

b: Circle
radius = 12.5

c: Circle
radius = 21.0

UML: Unified Modeling Language

Class as a Data Type

```
class Circle {
public:
    double radius;
public:
    Circle( ) {
            radius = 1.0;
    }
    Circle( double radius) { // newRadius
            this->radius = radius;
    }
    double getArea( ) const {
        return radius*radius*3.141592654;
   }
};

Circle a, b(12.5), c(21);
```

- Use class names to declare object names.
- A class is a data type.
- Use primitive data types to declare variables.

Constant Object Name

Square mySquare;

- Once an object name is declared, it represents an object.
- It cannot be reassigned to represent another object.
- An object name is a constant.
- The contents of the object may change.

Anonymous Object? When to use it?

- Use it once.
- Do not refer to it later.

```
void f( A x );

void g( ) {
   f( A( ) );
}
```

```
void f( A *x ); // forward declaration

void g( ) {
   f( new A );

   f( new A( ) );
}
// may have the memory leak problem
```

Anonymous Object? When to use it?

- Use it once.
- Do not refer to it later.

```
void f( A x );

void g( ) {
  f( A( ) );
}
```

```
void f( A *x ); // forward declaration

void g( ) {
  f( new A );

  f( new A( ) );
}
// may have the memory leak problem
```

```
void f( A *x ) {
  …
  delete x;

  …
}
```

Anonymous Object? When to use it?

- Use it once.
- Do not refer to it later.

```
void f( A x );

void g( ) {
   f( A( ) );
}
```

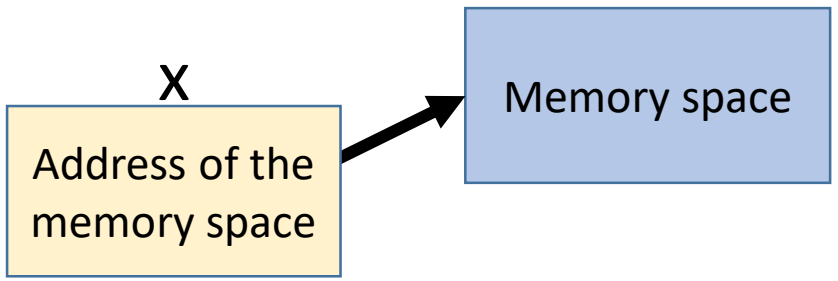
```
void f( A *x ); // forward declaration

void g( ) {
   f( new A );

   f( new A( ) );
}
// may have the memory leak problem
```

```
void f( A *x ) {
   …
   delete x; // A1 the memory space
             // A2
   …
}
```

Anonymous Object? When to use it?

- Use it once.
- Do not refer to it later.

```
void f( A x );

void g( ) {
  f( A( ) );
}
```



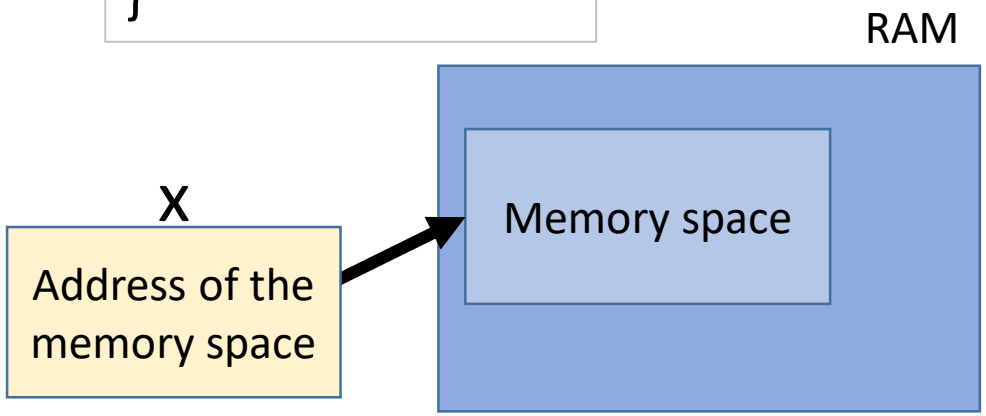

```
void f( A *x ); // forward declaration

void g( ) {
  f( new A );

  f( new A( ) );
}
// may have the memory leak problem
```

```
void f( A *x ) {
  …
  delete x; // release the memory space
            // pointed to by x
  …
}
```

Anonymous Object? When to use it?

- Use it once.
- Do not refer to it later.

```
void f( A x );

void g( ) {
  f( A( ) );
}
```



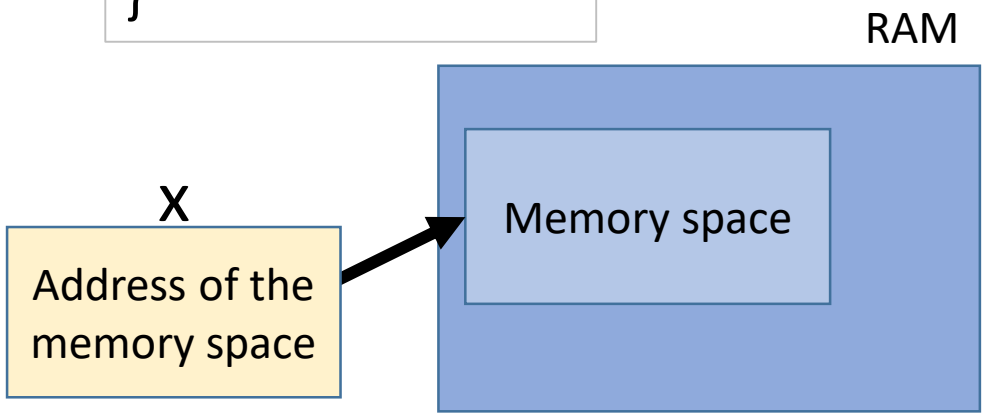

```
void f( A *x ); // forward declaration

void g( ) {
  f( new A );

  f( new A( ) );
}
// may have the memory leak problem
```

```
void f( A *x ) {
  …
  delete x; // release the memory space
            // pointed to by x
  …
}
```

Anonymous Object? When to use it?

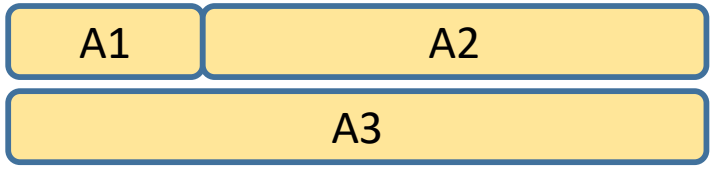
- Use it once.
- Do not refer to it later.

```
void f( A x );

void g( ) {
  f( A( ) );
}
```




```
void f( A *x ); // forward declaration

void g( ) {
  f( new A );

  f( new A( ) );
}
// may have the memory leak problem
```

```
void f( A *x ) {
  …
  delete x; //
            //
  …
}
```



Anonymous Object

- When we create an object and use it once, we do not need to name it.

```
// create an anonymous object using the no-arg constructor
        ClassName( );

// create an anonymous object using the constructor with argument
        ClassName(arguments);


        new ClassName(arguments);      // must be careful for memory leak
```

Anonymous Object as a Time Reporter

```
#pragma once
#include <ctime>
#include <iostream>
class FUNC_PROFILE {
private:
    clock_t start;
    double duration;
public:
    FUNC_PROFILE() {
        start = clock();
    }
    ~FUNC_PROFILE() {
        duration = (clock() - start) / (double) CLOCKS_PER_SEC;
        std::cout << "duration:" << duration << std::endl;
    }
};
```

Anonymous Object as a Time Reporter

```
#pragma once
#include <ctime>
#include <iostream>
class FUNC_PROFILE {
private:
    clock_t start;
    double duration;
public:
    FUNC_PROFILE() {
        start = clock();
    }
    ~FUNC_PROFILE() {
        duration = (clock() - start) / (double) CLOCKS_PER_SEC;
        std::cout << "duration:" << duration << std::endl;
    }
};
```

```
void foo( ) {
        // create and
        // destroy at the end of
        // the function call.
        FUNC_PROFILE a;
        ……
}
```

Anonymous Object as a Time Reporter

```
#pragma once
#include <ctime>
#include <iostream>
class FUNC_PROFILE {
private:
    clock_t start;
    double duration;
public:
    FUNC_PROFILE() {
        start = clock();
    }
    ~FUNC_PROFILE() {
        duration = (clock() - start) / (double) CLOCKS_PER_SEC;
        std::cout << "duration:" << duration << std::endl;
    }
};
```

```
void foo( ) {
        // create and
        // destroy at the end of
        // the function call.
        FUNC_PROFILE a;
        ……
}
```

```
void foo( ) {
        //create and destroy at once
        FUNC_PROFILE();
        ……
}
```

Class and Structure

```
struct Record {
   char id[16];
   char phoneNumber[16];
   int age;
};
```

Need to implement functions to handle the structure objects.

```
class Record {
   protected:
      string id;
      string phoneNumber;
      int age;
   public:
      void printf() const;
      void setID(const string &id);
};
```

Class and Structure

- A structure and a class defines data members.
- A class has methods.
- A structure does not have any methods.

```
struct Record {
   char id[16];
   char phoneNumber[16];
   int age;
};
```

Need to implement functions to handle the structure objects.

```
class Record {
   protected:
      string id;
      string phoneNumber;
      int age;
   public:
      void printf() const;
      void setID(const string &id);
};
```

Memberwise Copy (shallow copy)

Copy the content of one object to another

Square x, y;

x = y; // use the A1

The data members of y are copied to their counterparts in object x.

Memberwise Copy (shallow copy)

Copy the content of one object to another

Square x, y;

x = y; // use the assignment copy

The data members of y are copied to their counterparts in object x.

Memberwise Copy (shallow copy)

Copy the content of one object to another

Square x, y;

x = y; // use the assignment copy

The data members of y are copied to their counterparts in object x.

```
class Square {
        public:
        double side;
        double area;
};
```

Memberwise Copy (shallow copy)

Copy the content of one object to another

Square x, y;

x = y; // use the assignment copy

The data members of y are copied to their counterparts in object x.

```
class Square {
        public:
        double side;
        double area;
};
```

```
x = y;
```

Same as

```
x.side = y.side;
x.area = y.area;
```

Memberwise Copy (shallow copy)

Copy the content of one object to another

Square x, y;

x = y; // use the assignment copy

The data members of y are copied to their counterparts in object x.

```
class Square {
        public:
        double side;
        double area;
        int *arr;
};
```

```
x = y;
```

Same as

```
x.side = y.side;
x.area = y.area;
x.arr = y.arr;
```

Separating Definition from Implementation

```
class Record {
   protected:
      string id;
      string phoneNumber;
      int age;
   public:
      Record( );
      void printf() const;
      void setID(const string &id);
};
```

Separating Definition from Implementation

```
class Record {
   protected:
      string id;
      string phoneNumber;
      int age;
   public:
      Record( );
      void printf() const;
      void setID(const string &id);
};
```

Record.h

Class declaration

Separating Definition from Implementation

```
class Record {
   protected:
      string id;
      string phoneNumber;
      int age;
   public:
      Record( );
      void printf() const;
      void setID(const string &id);
};
```

Record.h

Class declaration

```
Record::Record( ) { id=“”; age=… }
void Record::printf( ) const { … }
void setID(const string &id) {
   this->id = id;
}
……
```

Separating Definition from Implementation

```
class Record {
   protected:
      string id;
      string phoneNumber;
      int age;
   public:
      Record( );
      void printf() const;
      void setID(const string &id);
};
```

Record.h

Class declaration

```
Record::Record( ) { id=“”; age=… }
void Record::printf( ) const { … }
void setID(const string &id) {
   this->id = id;
}
……
```

Record.cpp

Class Implementation

Define content

Separating Definition from Implementation

Class definition: describe the contract of the class.

Class declaration: simply list all the data fields, constructor prototypes, and the function prototypes.

Class implementation: implement the contract, including constructors and functions.

- The class declaration and implementation are in two separate files.
- Both files should have the same name, but with different extension names.

- Class declaration file: the extension name is .h
- Class implementation file: the extension name is .cpp

```
class Record {
  protected:
    string id;
    string phoneNumber;
    int age;
  public:
    Record( );
    void printf() const;
    void setID(const string &id);
};
```

Record.h

Class declaration

```
Record::Record( ) { id=“”; age=… }
void Record::printf( ) const { … }
void setID(const string &id) {
  this->id = id;
}
……
```

Record.cpp

Class Implementation

Define content

Inline Declaration in a class

Inline function may improve the execution time and speed of the program.

```
class A {
  ……
  // foo is defined inside the class body of A
   int foo( int a, int b) { return a + b; }     // A1 method
}
```

Inline Declaration in a class

Inline function may improve the execution time and speed of the program.

```
class A {
  ……
  // foo is defined inside the class body of A
   int foo( int a, int b}; // declaration
}
```

A1 int A::foo(int a, int b) { return a + b; } // A2 method

Inline Declaration in a class

Inline function may improve the execution time and speed of the program.

```
class A {
  ……
  // foo is defined inside the class body of A
   int foo( int a, int b}; // declaration
}

inline int A::foo( int a, int b) { return a + b; }  // inline method
```

Data Field Encapsulation

```
class Record_N {
   public:
      string id;
      string phoneNumber;
      int age;
   public:
      Record( );
      void printf() const;
      void setID(const string &id);
};
```

```
void main( ) {
   Record_N r;
   r.id = “012345678”; // ok
   r.age = 18;
}
```

main is the client of Record_N

Data Field Encapsulation

```
class Record_N {
   public:
      string id;
      string phoneNumber;
      int age;
   public:
      Record( );
      void printf() const;
      void setID(const string &id);
};
```

```
class Record {
   protected:
      string id;
      string phoneNumber;
      int age;
   public:
      Record( );
      void printf() const;
      void setID(const string &id);
};
```

```
void main( ) {
   Record_N r;
   r.id = "012345678"; // ok
   r.age = 18;
}
```

```
void main( ) {
   Record r;
   r.id = "012345678"; //
   r.age = 18;         //
}
```

A1

A2

main is the **client** of Record_N and Record.

Data Field Encapsulation

Non-hidden data members

- Clients can access them directly.
- There are potential problems.
- They can be modified directly from outside.

Potential problems

- Data may be tampered.
- It makes the class difficult to maintain.
- It is vulnerable to bugs.
- Clients may modify the data members. Subsequently, the state of an object can be wrong.

```
class Record_N {
  public:
    string id;
    string phoneNumber;
    int age;
  public:
    Record( );
    void printf() const;
    void setID(const string &id);
};
```

```
class Record {
  protected:
    string id;
    string phoneNumber;
    int age;
  public:
    Record( );
    void printf() const;
    void setID(const string &id);
};
```

```
void main( ) {
  Record_N r;
  r.id = "012345678"; // ok
  r.age = 18;
}
```

```
void main( ) {
  Record r;
  r.id = "012345678"; // error
  r.age = 18;          // error
}
```

main is the **client** of Record_N and Record.

Data Field Encapsulation

Non-hidden data members

- Clients can access them directly.
- There are potential problems.
- They can be modified directly from outside.

Potential problems

- Data may be tampered.
- It makes the class difficult to maintain.
- It is vulnerable to bugs.
- Clients may modify the data members. Subsequently, the state of an object can be wrong.

```
class Record_N {
   public:
   int num;
   int *arr;

 public:
      Record( );
      void printf() const;
      void setID(const string &id);
};
```

```
class Record {
   protected:
   int num;
   int *arr;

 public:
      Record( );
      void printf() const;
      void setID(const string &id);
};
```

```
void main( ) {
   Record_N r;

   r.arr = (int*) 1234;
}
```

```
void main( ) {
   Record r;

   r.arr = (int*) 1234;        // error
}
```

main is the **client** of Record_N and Record.

Accessor and Mutator

```
class Record {
public:
      string getID( ) const;
      int getAge( ) const;

      bool isMale( ) const;
      bool isFemale( ) const;

      void setID( const string &id );
      void setAge( int age );
public:
      Record( );
      void printf() const;
 protected:
      string id;
      string phoneNumber;
      int age;
      GENDER gender;
};
```

Accessor and Mutator

get functions

get functions return a bool data type

set functions

```
class Record {
public:
      string getID( ) const;
      int getAge( ) const;

      bool isMale( ) const;
      bool isFemale( ) const;

      void setID( const string &id );
      void setAge( int age );
public:
      Record( );
      void printf() const;
 protected:
      string id;
      string phoneNumber;
      int age;
      GENDER gender;
};
```

Accessor and Mutator

get functions

get functions return a bool data type

set functions

```
class Record {
public:
      string getID( ) const;
      int getAge( ) const;

      bool isMale( ) const;
      bool isFemale( ) const;

      void setID( const string &id );
      void setAge( int age );
public:
      Record( );
      void printf() const;
 protected:
      string id;
      string phoneNumber;
      int age;
      GENDER gender;
};
```

Accessor and Mutator

- A get function: a getter (or accessor)
- A set function: a setter (or mutator)

The signature of a get function:

A1 getPropertyName( )

The get function returns a bool data type

A2 isPropertyName( )

The signature of a set function:

A3

setPropertyName(dataType propertyValue)

get functions

get functions return a bool data type

set functions

```
class Record {
public:
      string getID( ) const;
      int getAge( ) const;

      bool isMale( ) const;
      bool isFemale( ) const;

      void setID( const string &id );
      void setAge( int age );
public:
      Record( );
      void printf() const;
 protected:
      string id;
      string phoneNumber;
      int age;
      GENDER gender;
};
```

Accessor and Mutator

- A get function: a getter (or accessor)
- A set function: a setter (or mutator)

The signature of a get function:
returnType getPropertyName( )

The get function returns a bool data type
bool isPropertyName( )

The signature of a set function:
public void
setPropertyName(dataType propertyValue)

get functions

get functions return a bool data type

set functions

```
class Record {
public:
      string getID( ) const;
      int getAge( ) const;

      bool isMale( ) const;
      bool isFemale( ) const;

      void setID( const string &id );
      void setAge( int age );
public:
      Record( );
      void printf() const;
 protected:
      string id;
      string phoneNumber;
      int age;
      GENDER gender;
};
```

Accessor and Mutator

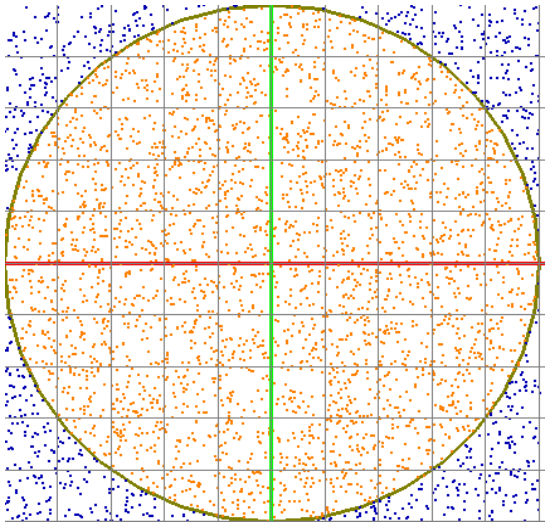
- When we decide to implement accessors and mutators for data members with different attributes, we need to think A1
- The modifications of data members do not lead to a A2
- Use the right attribute modifiers for declaring the data members: private, A3

Accessor and Mutator

- When we decide to implement accessors and mutators for data members with different attributes, we need to think carefully.
- The modifications of data members do not lead to a mistake.
- Use the right attribute modifiers for declaring the data members: private, protected, public

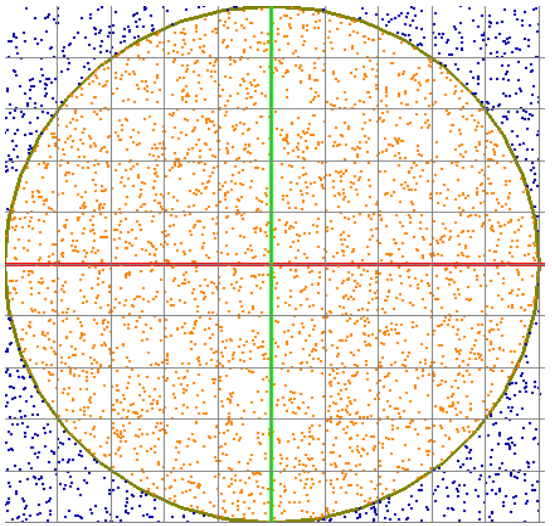
Class Abstraction and Encapsulation

Class Abstraction and Encapsulation

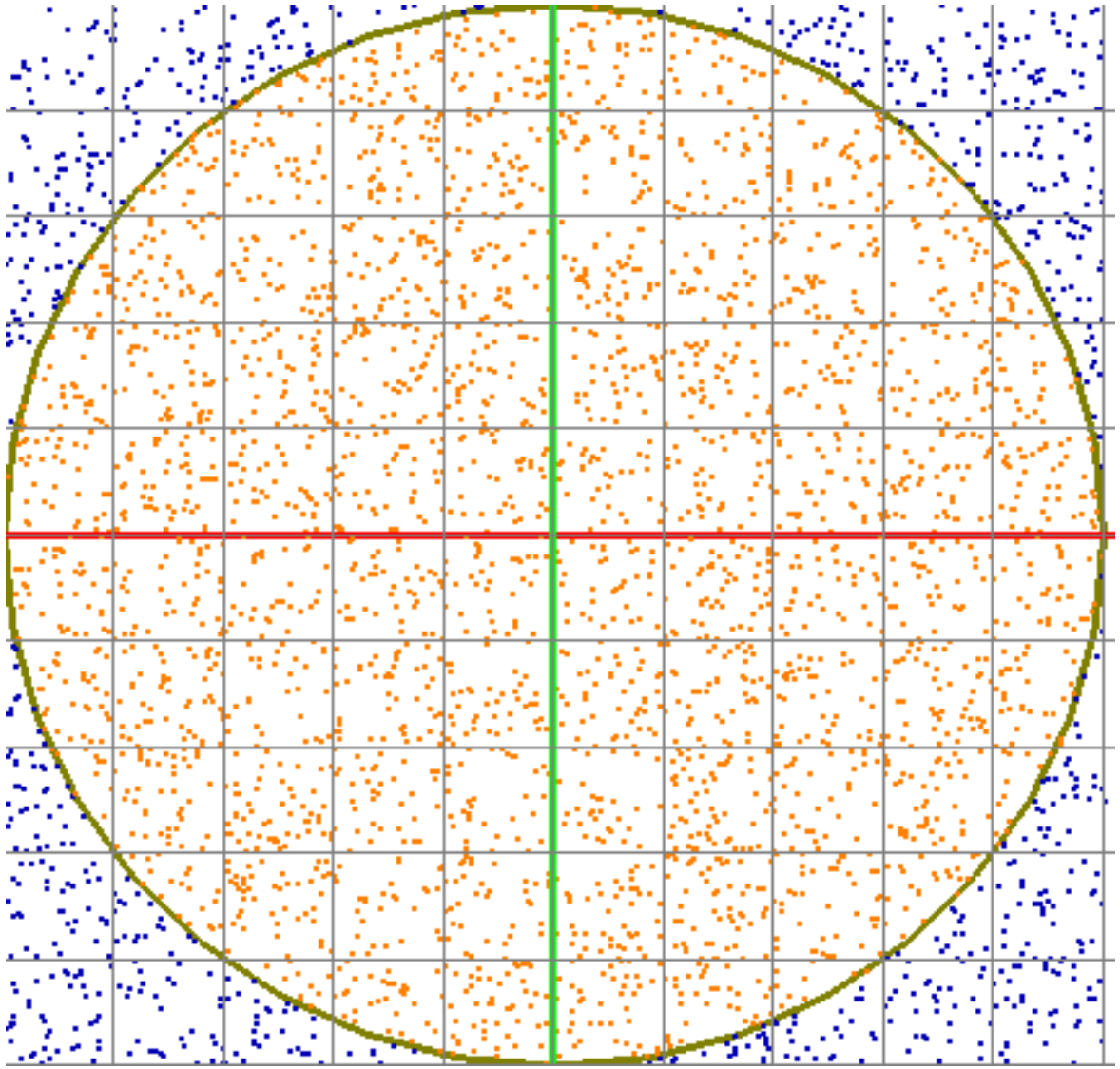


Class Abstraction and Encapsulation

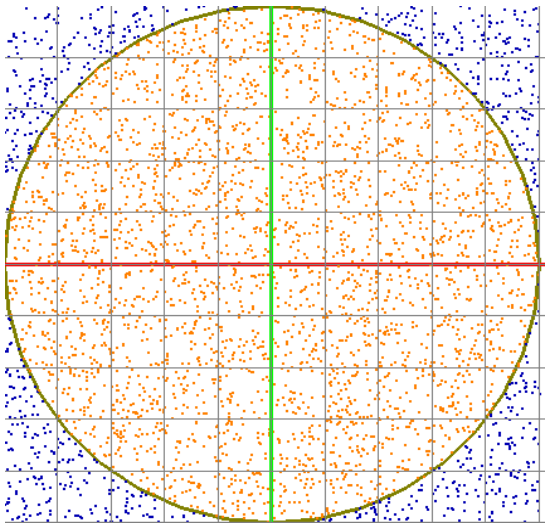
```
class MONTE_CARLO_SYSTEM {
protected:
   double mRadius;
   int mNumSamples;  // number of sample points
   vector<float> mX;   // x-coordinate
   vector<float> mY;   // y-coordinate
   void generateUniformSamples( );
public:
   MONTE_CARLO_SYSTEM( );
   void askForInput( );
   void reset();
   double computePI() const;
   double getRadius( ) const;
   int getNumSamples( ) const;  // get the number of samples
   bool getSample(
      int sampleIndex, float &x, float &y) const;
};
```



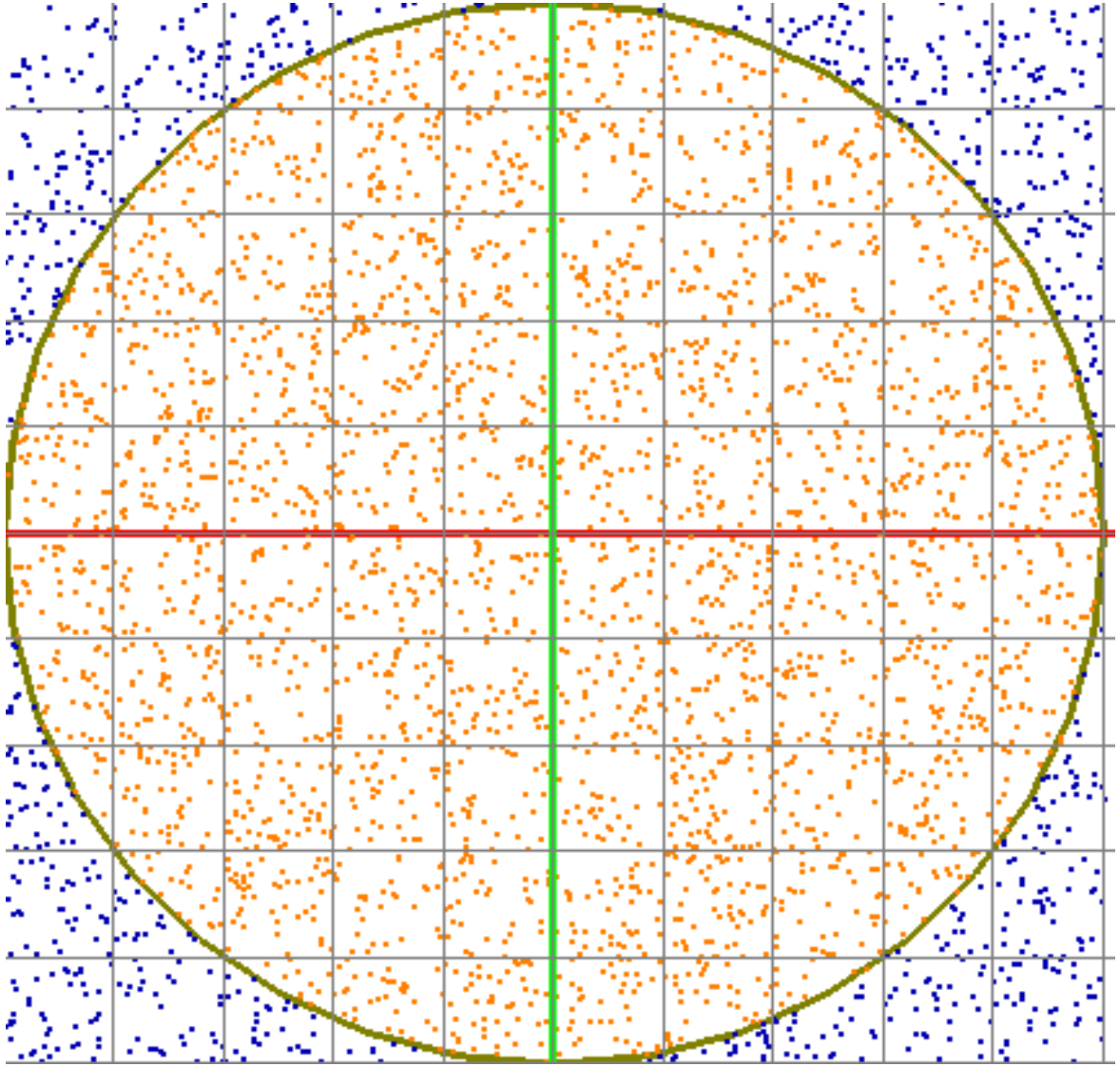
Class Abstraction and Encapsulation



```
class MONTE_CARLO_SYSTEM {
protected:
   double mRadius;
   int mNumSamples;  // number of sample points
   vector<float> mX;   // x-coordinate
   vector<float> mY;   // y-coordinate
   void generateUniformSamples( );
public:
   MONTE_CARLO_SYSTEM( );
   void askForInput( );
   void reset();
   double computePI() const;
   double getRadius( ) const;
   int getNumSamples( ) const;  // get the number of samples
   bool getSample(
      int sampleIndex, float &x, float &y) const;
};
```

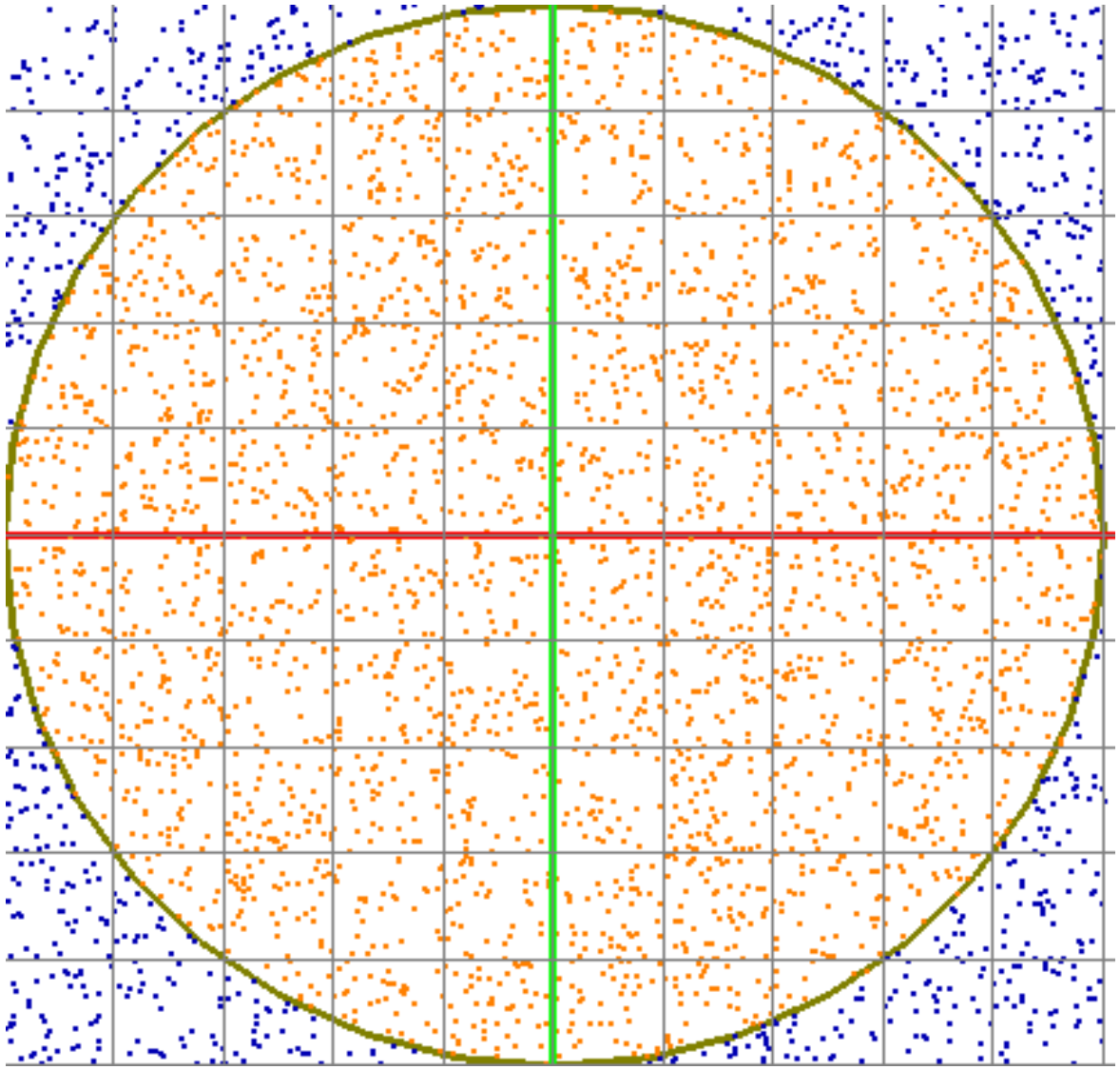


Class Abstraction and Encapsulation



```
class MONTE_CARLO_SYSTEM {
protected:
   double mRadius;
   int mNumSamples;  // number of sample points
   vector<float> mX;   // x-coordinate
   vector<float> mY;   // y-coordinate
   void generateUniformSamples( );
public:
   MONTE_CARLO_SYSTEM( );
   void askForInput( );
   void reset();
   double computePI() const;
   double getRadius( ) const;
   int getNumSamples( ) const;  // get the number of samples
   bool getSample(
      int sampleIndex, float &x, float &y) const;
};
```

Class Abstraction and Encapsulation



```
class MONTE_CARLO_SYSTEM {
protected:
   double mRadius;
   int mNumSamples;  // number of sample points
   vector<float> mX;   // x-coordinate
   vector<float> mY;   // y-coordinate
   void generateUniformSamples( );
public:
   MONTE_CARLO_SYSTEM( );
   void askForInput( );
   void reset();
   double computePI() const;
   double getRadius( ) const;
   int getNumSamples( ) const;  // get the number of samples
   bool getSample(
      int sampleIndex, float &x, float &y) const;
};
```

The client does not need to know how the sample points are stored.

Class Abstraction and Encapsulation

Class abstraction: separate

A1

A2

from

```
class MONTE_CARLO_SYSTEM {
protected:
   double mRadius;
   int mNumSamples;  // number of sample points
   vector<float> mX;   // x-coordinate
   vector<float> mY;   // y-coordinate
   void generateUniformSamples( );
public:
   MONTE_CARLO_SYSTEM( );
   void askForInput( );
   void reset();
   double computePI() const;
   double getRadius( ) const;
   int getNumSamples( ) const;  // get the number of samples
   bool getSample(
      int sampleIndex, float &x, float &y) const;
};
```

The client does not need to know how the sample points are stored.

Class Abstraction and Encapsulation

Class abstraction: separate class implementation from **the use of the class**.

```
class MONTE_CARLO_SYSTEM {
protected:
   double mRadius;
   int mNumSamples;  // number of sample points
   vector<float> mX;   // x-coordinate
   vector<float> mY;   // y-coordinate
   void generateUniformSamples( );
public:
   MONTE_CARLO_SYSTEM( );
   void askForInput( );
   void reset();
   double computePI() const;
   double getRadius( ) const;
   int getNumSamples( ) const;  // get the number of samples
   bool getSample(
      int sampleIndex, float &x, float &y) const;
};
```

The client does not need to know how the sample points are stored.

Class Abstraction and Encapsulation

Class abstraction: separate class implementation from **the use of the class**.

- The creator of the class **provides a** A1 of the class.

```
class MONTE_CARLO_SYSTEM {
protected:
   double mRadius;
   int mNumSamples;  // number of sample points
   vector<float> mX;   // x-coordinate
   vector<float> mY;   // y-coordinate
   void generateUniformSamples( );
public:
   MONTE_CARLO_SYSTEM( );
   void askForInput( );
   void reset();
   double computePI() const;
   double getRadius( ) const;
   int getNumSamples( ) const;  // get the number of samples
   bool getSample(
      int sampleIndex, float &x, float &y) const;
};
```

The client does not need to know how the sample points are stored.

Class Abstraction and Encapsulation

Class abstraction: separate class implementation from **the use of the class**.

➢The creator of the class **provides a description** of the class.

```
class MONTE_CARLO_SYSTEM {
protected:
   double mRadius;
   int mNumSamples;  // number of sample points
   vector<float> mX;   // x-coordinate
   vector<float> mY;   // y-coordinate
   void generateUniformSamples( );
public:
   MONTE_CARLO_SYSTEM( );
   void askForInput( );
   void reset();
   double computePI() const;
   double getRadius( ) const;
   int getNumSamples( ) const;  // get the number of samples
   bool getSample(
      int sampleIndex, float &x, float &y) const;
};
```

The client does not need to know how the sample points are stored.

Class Abstraction and Encapsulation

Class abstraction: separate class implementation from **the use of the class**.

- The creator of the class **provides a description** of the class.
- Let the user know **how the class can**

A1

```
class MONTE_CARLO_SYSTEM {
protected:
   double mRadius;
   int mNumSamples;  // number of sample points
   vector<float> mX;   // x-coordinate
   vector<float> mY;   // y-coordinate
   void generateUniformSamples( );
public:
   MONTE_CARLO_SYSTEM( );
   void askForInput( );
   void reset();
   double computePI() const;
   double getRadius( ) const;
   int getNumSamples( ) const;  // get the number of samples
   bool getSample(
      int sampleIndex, float &x, float &y) const;
};
```

The client does not need to know how the sample points are stored.

Class Abstraction and Encapsulation

Class abstraction: separate class implementation from **the use of the class**.

- The creator of the class **provides a description** of the class.
- Let the user know **how the class can be used**.
- The user of the class **does not need to know how the class** A1

```
class MONTE_CARLO_SYSTEM {
protected:
   double mRadius;
   int mNumSamples;  // number of sample points
   vector<float> mX;   // x-coordinate
   vector<float> mY;   // y-coordinate
   void generateUniformSamples( );
public:
   MONTE_CARLO_SYSTEM( );
   void askForInput( );
   void reset();
   double computePI() const;
   double getRadius( ) const;
   int getNumSamples( ) const;  // get the number of samples
   bool getSample(
      int sampleIndex, float &x, float &y) const;
};
```

The client does not need to know how the sample points are stored.

Class Abstraction and Encapsulation

Class abstraction: separate class implementation from **the use of the class**.

- The creator of the class **provides a description** of the class.
- Let the user know **how the class can be used**.
- The user of the class **does not need to know how the class is implemented**.
- The detail of implementation is A1 and A2 from the user.

```
class MONTE_CARLO_SYSTEM {
protected:
   double mRadius;
   int mNumSamples;  // number of sample points
   vector<float> mX;   // x-coordinate
   vector<float> mY;   // y-coordinate
   void generateUniformSamples( );
public:
   MONTE_CARLO_SYSTEM( );
   void askForInput( );
   void reset();
   double computePI() const;
   double getRadius( ) const;
   int getNumSamples( ) const;  // get the number of samples
   bool getSample(
      int sampleIndex, float &x, float &y) const;
};
```

The client does not need to know how the sample points are stored.

Class Abstraction and Encapsulation

Class abstraction: separate class implementation from **the use of the class**.

- The creator of the class **provides a description** of the class.
- Let the user know **how the class can be used**.
- The user of the class **does not need to know how the class is implemented**.
- The detail of implementation is encapsulated and hidden from the user.

```
class MONTE_CARLO_SYSTEM {
protected:
   double mRadius;
   int mNumSamples;  // number of sample points
   vector<float> mX;   // x-coordinate
   vector<float> mY;   // y-coordinate
   void generateUniformSamples( );
public:
   MONTE_CARLO_SYSTEM( );
   void askForInput( );
   void reset();
   double computePI() const;
   double getRadius( ) const;
   int getNumSamples( ) const;  // get the number of samples
   bool getSample(
      int sampleIndex, float &x, float &y) const;
};
```

The client does not need to know how the sample points are stored.

Intended Learning Outcomes

- Name the features of a class
- Identify the scopes of variables (local or global)
- Distinguish between accessors and mutators
- List the features of class abstraction and encapsulation
- List the benefits of class abstraction and encapsulation

Supplemental Materials

Variable Scope

Where can you find a variable?

```
void foo ( int a ) {
   int r;
   r = 12.0;

}

void main( ) {
   int r;
   foo( 10 );
}
```

Variable Scope

```
void foo ( int a ) {
   int r;
   r = 12.0;

}

void main( ) {
   int r;
   foo( 10 );
}
```

```
double r;
void foo ( int a ) {
   int r;
   r = 12.0;
}

void main( ) {
   double r;
   foo( 10 );
}
```

Variable Scope

```
void foo ( int a ) {
   int r;
   r = 12.0;

}

void main( ) {
   int r;
   foo( 10 );
}
```

```
double r;
void foo ( int a ) {
   int r;
   r = 12.0;
}

void main( ) {
   double r;
   foo( 10 );
}
```

```
double r;
void foo ( int a ) {
   int r;
   r = 12.0;
   for ( int i = 0; i < 10; ++i ) {
      double r = 0.0;
      r += i;
   }
}

void main( ) {
   double r;
   foo( 10 );
}
```

Variable Scope

- The (lexical) scope of a binding: visibility of an entity

```
void foo ( int a ) {
   int r;
   r = 12.0;

}

void main( ) {
   int r;
   foo( 10 );
}
```

```
double r;
void foo ( int a ) {
   int r;
   r = 12.0;
}

void main( ) {
   double r;
   foo( 10 );
}
```

```
double r;
void foo ( int a ) {
   int r;
   r = 12.0;
   for ( int i = 0; i < 10; ++i ) {
      double r = 0.0;
      r += i;
   }
}

void main( ) {
   double r;
   foo( 10 );
}
```

The Scope of Variables

Global variables

- they are declared A1 all functions
- they are accessible to A2 functions in its scope
- The scope of a global variable: from its declaration and to A3

```
double r;
void foo ( int a ) {
  int r;
  r = 12.0;
  for ( int i = 0; i < 10; ++i ) {
     double r = 0.0;
     r += i;
  }
}

void main( ) {
  double r;
  foo( 10 );
}
```

The Scope of Variables

Global variables

- they are declared outside all functions
- they are accessible to all functions in its scope
- The scope of a global variable: from its declaration and to the end of the program.

```
double r;
void foo ( int a ) {
  int r;
  r = 12.0;
  for ( int i = 0; i < 10; ++i ) {
     double r = 0.0;
     r += i;
  }
}

void main( ) {
  double r;
  foo( 10 );
}
```

The Scope of Variables

Global variables

- they are declared outside all functions
- they are accessible to all functions in its scope
- The scope of a global variable: from its declaration and to the end of the program.

Local variables

- They are defined A1 functions
- The scope of a local variable: from its declaration and continues to the A2 A3 containing the variable.

```
double r;
void foo ( int a ) {
  int r;
  r = 12.0;
  for ( int i = 0; i < 10; ++i ) {
     double r = 0.0;
     r += i;
  }
}

void main( ) {
  double r;
  foo( 10 );
}
```

The Scope of Variables

Global variables

- they are declared outside all functions
- they are accessible to all functions in its scope
- The scope of a global variable: from its declaration and to the end of the program.

Local variables

- They are defined inside functions
- The scope of a local variable: from its declaration and continues to the end of the block containing the variable.

```
double r;
void foo ( int a ) {
  int r;
  r = 12.0;
  for ( int i = 0; i < 10; ++i ) {
     double r = 0.0;
     r += i;
  }
}

void main( ) {
  double r;
  foo( 10 );
}
```

The Scope of Variables

Global variables

- they are declared outside all functions
- they are accessible to all functions in its scope
- The scope of a global variable: from its declaration and to the end of the program.

Local variables

- They are defined inside functions
- The scope of a local variable: from its declaration and continues to the end of the block containing the variable.

```
double r;
void foo ( int a ) {
  int r;
  r = 12.0;
  for ( int i = 0; i < 10; ++i ) {
     double r = 0.0;
     r += i;
  }
}

void main( ) {
  double r;
  foo( 10 );
}
```

The Scope of Variables

Global variables

- they are declared outside all functions
- they are accessible to all functions in its scope
- The scope of a global variable: from its declaration and to the end of the program.

Local variables

- They are defined inside functions
- The scope of a local variable: from its declaration and continues to the end of the block containing the variable.

```
double r;
void foo ( int a ) {
  int r;
  r = 12.0;
  for ( int i = 0; i < 10; ++i ) {
     double r = 0.0;
     r += i;
  }
}

void main( ) {
  double r;
  foo( 10 );
}
```

The Scope of Variables

Global variables

- they are declared outside all functions
- they are accessible to all functions in its scope
- The scope of a global variable: from its declaration and to the end of the program.

Local variables

- They are defined inside functions
- The scope of a local variable: from its declaration and continues to the end of the block containing the variable.

```
double r;
void foo ( int a ) {
  int r;
  r = 12.0;
  for ( int i = 0; i < 10; ++i ) {
    double r = 0.0;
    r += i;
  }
}

void main( ) {
  double r;
  foo( 10 );
}
```

The Scope of Variables

Global variables

- they are declared outside all functions
- they are accessible to all functions in its scope
- The scope of a global variable: from its declaration and to the end of the program.

Local variables

- They are defined inside functions
- The scope of a local variable: from its declaration and continues to the end of the block containing the variable.

```
double r;
void foo ( int a ) {
  int r;
  r = 12.0;
  for ( int i = 0; i < 10; ++i ) {
    double r = 0.0;
    r += i;
  }
}

void main( ) {
  double r;
  foo( 10 );
}
```

The Scope of Variables

Global variables

- they are declared outside all functions
- they are accessible to all functions in its scope
- The scope of a global variable: from its declaration and to the end of the program.

Local variables

- They are defined inside functions
- The scope of a local variable: from its declaration and continues to the end of the block containing the variable.

```
double r;
void foo ( int a ) {
  int r;
  r = 12.0;
  for ( int i = 0; i < 10; ++i ) {
     double r = 0.0;
     r += i;
  }
}

void main( ) {
  double r;
  foo( 10 );
}
```

The Scope of Data Members (Data Fields)

Data fields can be accessible by all methods, constructors, and the destructor.

Data fields and functions can be declared in any order in a class.

```
class Rectangle {
  private:
        double side1, side2;
  public:
        Rectangle( );
        double getSide1( ) const;
};
```

```
class Rectangle {
public:
        Rectangle( );
        double getSide1( ) const;
private:
        double side1, side2;
};
```

The Scope of Data Members (Data Fields)

Data fields can be accessible by all methods, constructors, and the destructor.

Data fields and functions can be declared in any order in a class.

```
class A {
private:
  int a; // data member( data field)
public:
  void foo( ) {
     int a;   // local variable has the same name as a data field
     a = 5;  // 5 is assigned to the local variable. Higher precedence
  }
};
```

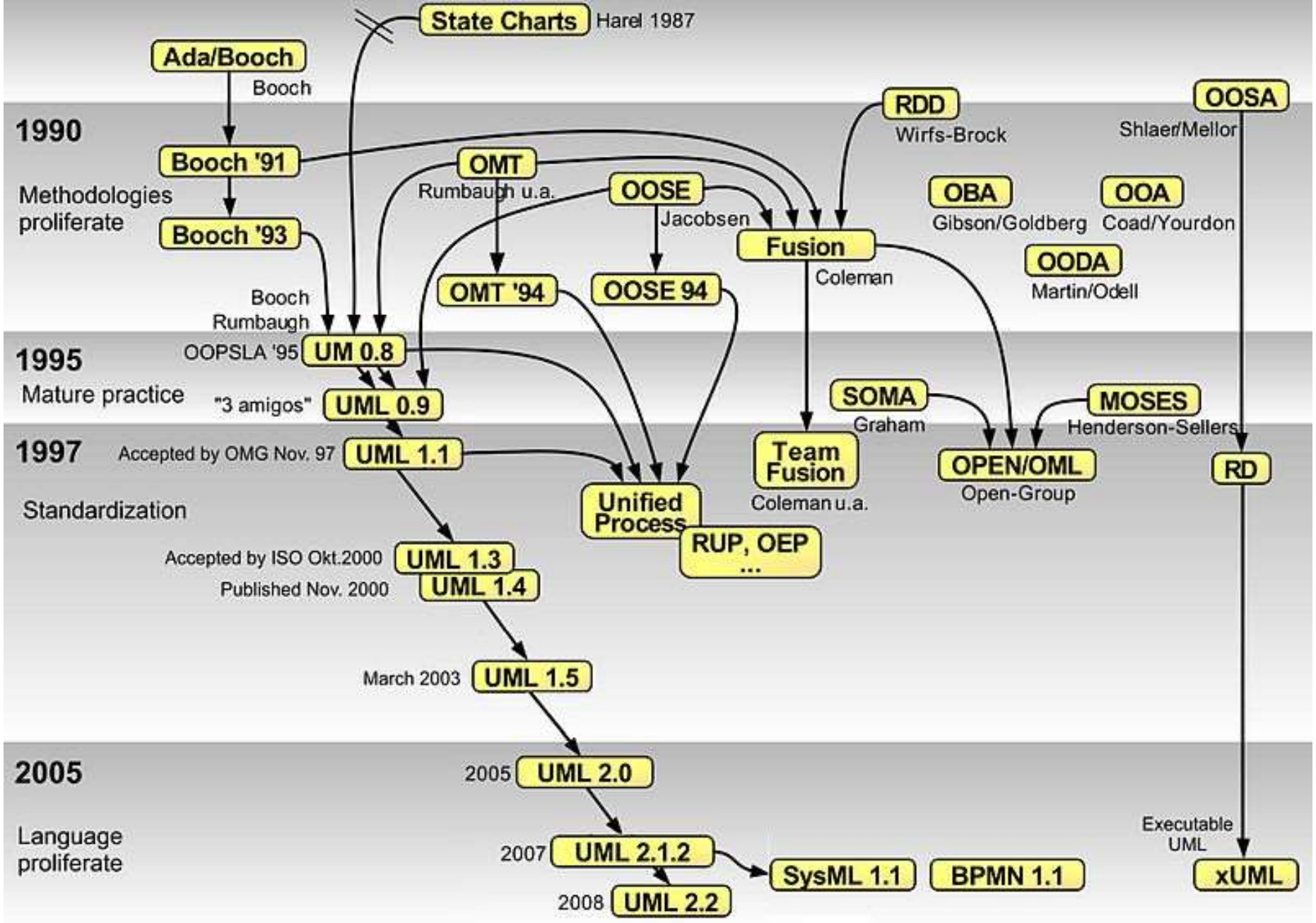
Data Encapsulation

- Data hiding

```
class Circle {
        private:
                double radius; // the client needs not to know what it is.
        …
};

Circle c;
c.radius = 10.1; // not allowed
```

Object-Oriented Programming (OOP) Concepts

- We use objects in implementing programs.
- An object represents an entity in the real world.
- An object has a unique identity, state, and a set of behaviors (implemented as methods).
- Objects can be distinctly identified.
- The state of an object consists of a set of data fields (or properties) with their current values.
- The behavior of an object is defined by a set of functions.



https://en.wikipedia.org/wiki/Unified_Modeling_Language#/media/File:OO_Modeling_languages_history.jpg

Object Names

Assign a name when we create an object.

A constructor is invoked when an object is created.

ClassName objectName; // Create an object using the no-arg constructor

For example,

SQUARE sq;

Constructing an object with arguments

// Declare an object using a constructor with arguments

ClassName objectName(arguments);

For example,

SQUARE sq(3.2);
TRIANGLE triangle(1.2, 1. , .8);

Access Operator

- *The access operator:* the dot operator (.)
- We can use this operator to access data members and methods.

objectName**.**dataField : reference a data field in the object.

objectName**.**function(arguments): invoke a function on the object.

Naming Objects and Classes

Capitalize the first letter of each word in a class name;

Examples: Circle, Square

inline functions

The binary code of the function is copied to the position that the function is invoked.

```
inline void g( int a, int b ) { ……}
void foo( ) {
        g( 1, 4);
        g( 3, 6);
}
```

cache

Rectangle Class

Rectangle
side1 : double side2: double area : double perimeter: double
+Rectangle( ) +Rectangle(double len1, double len2) +getSide1( ) +getSide2( ) +setSide1(side1: double): void +setSide2(side2: double): void +getArea( ) +getPerimeter( ) +computeArea( ) …