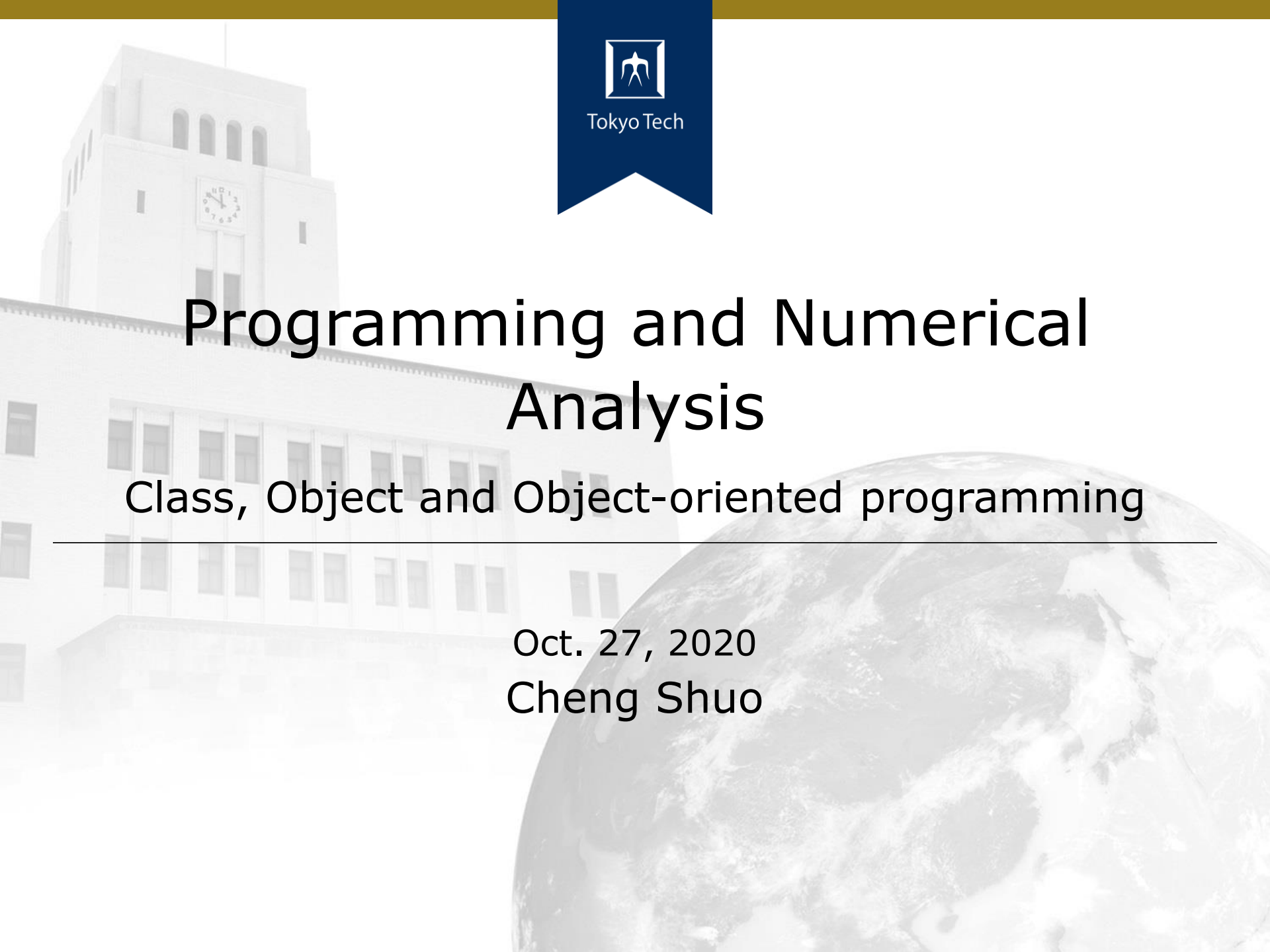


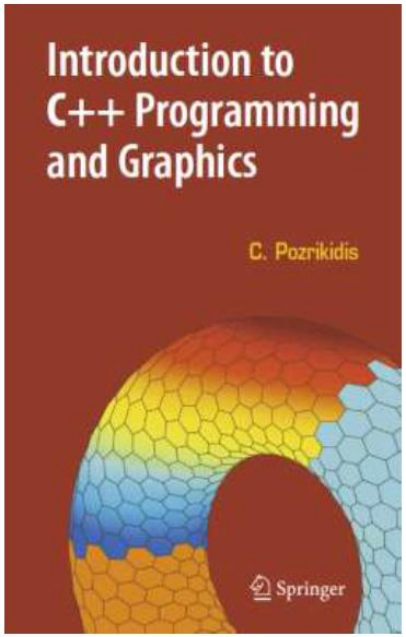
Programming and Numerical Analysis

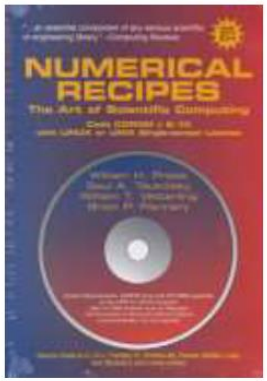
Class, Object and Object-oriented programming

Oct. 27, 2020

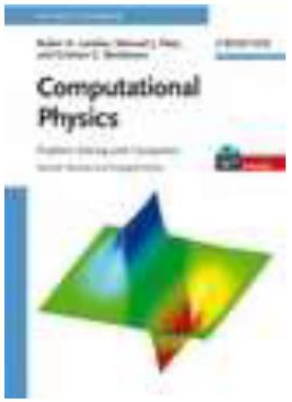
Cheng Shuo

Text Book





Numerical recipes in C++ : the art of scientific computing / William H. Press ... [et al.], ISBN: 9780521750332 [0521750334]



Introduction to C++ Programming and Graphics / Constantine Pozrikidis, Springer eBooks Computer Science, ISBN: 9780387689920 [0387689923], https://link.springer.com/10.1007/978-0-387-68993-7

Computational Physics - Problem Solving with Computers / Rubin H. Landau, Cristian C. Bordeianu, Manuel José Páez Mejía Wiley Online Library Online Books ISBN: 9783527406265 [3527406263] 9783527618835 [352761883X], https://onlinelibrary.wiley.com/book/10.1002/9783527618835

Remind: Defined data types

- C++ allows us to duplicate a data type into something that is either more familiar or more convenient. For example, if year is a non-negative integer, we may declare:

```
unsigned int year;          // "year" is expressed as a positive integer
```

Since the year is positive, we have exercised the unsigned option.

- We can duplicate "unsigned int" into "hronos" meaning year in Greek, by stating:

```
typedef unsigned int hronos;
```

The data types unsigned int and hronos are now synonyms. We may then declare:

```
hronos year;
```

Remind: Data structures-1

- Consider a group of *M* objects,

  o1, o2, . . . , oM,

  a group of *N* properties,

  p1, p2, . . . , pN,

  and denote the *j*th property of the *i*th object by:

  oi.pj

- The individual properties of the objects can be accommodated in a data structure defined, for example, as:

```
struct somename
{
        int p1;
        float p2;
        double p3;
        double p4;
}
o1, o2, o3;
```

Remind: Data structures-2

- Alternatively, we may define a data structure in terms of the properties alone by declaring:

```
struct somename
{
        int p1;
        float p2;
        double p3;
        double p4;
};
```

and then introduce members by declaring:

```
somename o1;
somename o2, o3;
```

- Objects and properties are threaded with a dot (.) into variables that convey expected meanings:

```
int o1.p1;
float o1.p2;
double o2.p3;
double o1.p4;
```

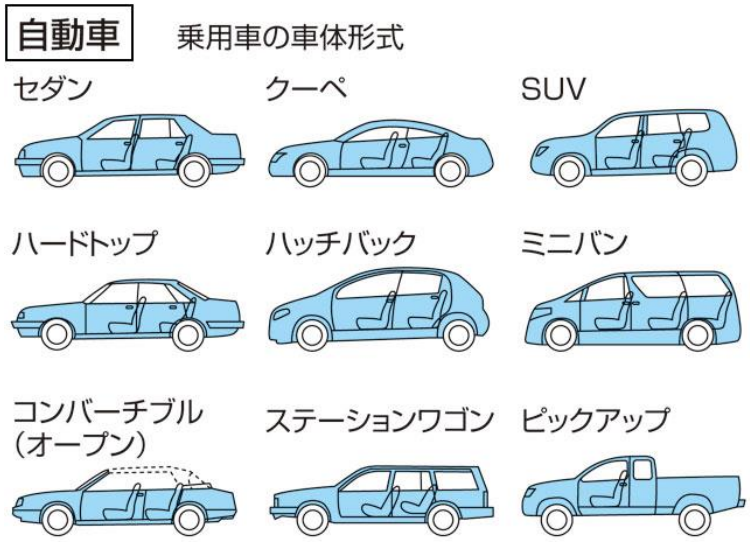
Remind: Data structures-3

- As an example, we define the used car lot structure:

```
struct car
{
        string make;
        int year;
        int miles;
        bool lemon;
}
vartburg1, skoda1, skoda2;
```

and then set:

```
skoda1.make = "skoda";
vartburg1.miles= 98932;
skoda1.lemon = true;
skoda2.lemon = false;
```



- Data structures and their members are preludes to classes and objects discussed in Chapter 6.

Classes and objects-1 Text p.149~

- The intelligent mind has a natural tendency to classify objects, items, concepts, and abstract notions into groups recognized by given names.

  Races in anthropology

  Species in biology

  Sets and spaces in mathematics

  Elementary particles in physics

  Elementary motions in fluid mechanics

- The groups are distinguished by common features and properties, concisely called attributes, and the members interact by a well-defined set of rules.
- An entity that belongs to a group is formally called a member, and an action that can modify a member, make a member disappear, or generate an offspring is called a member function.

Classes and objects-2

- Examples of groups are:

➢The set of natural numbers: 1, 2, 3, 4, . . . :

The member function "addition of unity" operating on the member "2" produces the member "3".

$$f(n) = n + 1$$

➢The set of integers: . . . , -4, -3, -2, -1, 0, 1, 2, 3, 4, . . . :

The member function "subtraction of unity" operating on the member "-3" produces the member "-4".

$$f(n) = n - 1$$

➢The set of rational numbers, *m/n*, where *m* and *n* are integers:

The member function "addition" operating on the members *m/n* and *k/l* produces the member (*lm* + *kn*)/(*nl*).

Classes and objects-3

- The set of real numbers registered as floating-point numbers in computer science:

  The member function "multiplication by zero" operating on a member produces the null member "0".

$$f(x) = 0x$$

- Vector spaces in mathematics:

  The member function "inner product" operating on a pair of members produces a number that is a measure of the angle subtended between the two vectors. If the inner product is zero, the two vectors are orthogonal.

$$\mathrm{f(x, y) = x \cdot y}$$

- The set of all two-index matrices a_{ij} :

  Each member is identified by the pair of integers *i* and *j*.

Classes and objects-4

- In calculus, a "member function" defined on the set of real numbers is a device that receives real numbers (input) and produces new numbers (output). Stated differently, a function maps the input to the output. When the output is the null point "0", the input has been destroyed.
- In **object oriented programming (OOP),** a group is a "class," a member is an "object," and a "member function" implements an operation. By operating on an object with a "function," we can read, record, and change some or all of its attributes.

For example, consider the class of all polygons. A member function can be defined that transforms a rectangle into a triangle in some sensible fashion.

Classes and objects-5

- Classes in **object oriented programming** can be as simple as the set of integers (int) or the set of floating point numbers stored in double precision (double), and as complex as a database whose members (entries) are described by names, numbers, and other fields.

  Table 6.1 displays the equivalence of groups, spaces, and objects and their relation in object oriented programming (OOP).

Table 6.1 Equivalence of groups, spaces, and objects and their relation in object oriented programming (OOP).

Generic:	Group	Member	Action
Maths:	Space	Element	Operation
OOP:	Class	Object	Member function
Science:	Discipline	Phenomenon	Dynamics

6.1 Class objects and functions-1

- An apple can be **declared** and **initialized** as a member of the "fruit" class by stating:

fruit apple = fruit(q, w, ..., e);

The parentheses enclose names and numbers that define the apple, and can be thought of as a bar code. This line says:

Apple is a fruit uniquely defined by the properties (attributes): q, w, ... e.

The attributes can be words, sentences, or numbers.

6.1 Class objects and functions-2

- A member function can be defined to transform an apple to an orange. Assume that apple has been defined as an object of the fruit class, and change has been defined as a member function. The C++ command that carries out this operation is stated as:

apple.change(x, y, ..., q);

The parentheses enclose numbers and strings that ensure the apple-to-orange transformation. This line says:

Mutate the apple in a way that is uniquely determined by the parameters:
x, y, ..., q.

- The apple may **disappear** after the operation, or continue to co-exist with the orange. Which will occur depends on how the member function change has been defined.

6.1 Class objects and functions-3

- Classes define new data types and corresponding class functions beyond those implemented in the standard C++ library. We may consider the familiar declaration and initialization of a string:

string gliko = "koulouraki";

We note the similarity with the previously stated apple declaration, and conclude that the string data type is implemented in a corresponding class with a simplified implementation syntax. C++ endows us with unlimited degrees of freedom for defining new data types and thereby building a language inside another language.

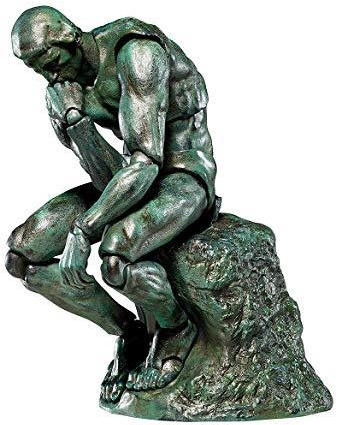
6.2 Class interfaces-1

- The member functions of a class accomplish a broad range of tasks.
  - First, they **construct (initialize)** native objects, that is, they evaluate the data fields that uniquely define an object.
  - Second, they allow us to **view** and **visualize** an object.
  - Third, they allow us to intrusively **operate** on an isolated object or groups of objects.

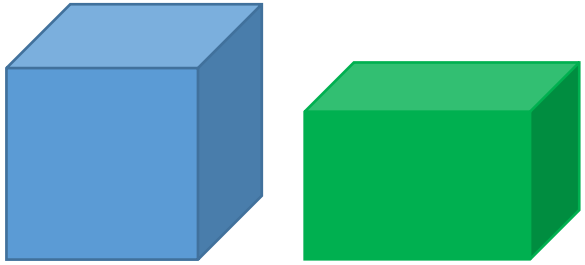
- The set of member functions pertinent to a particular class is the class interface.

6.2 Class interfaces-2

- Constructors
  - These member functions **initialize an object**.
  - Constructors come in two flavors: default constructors and parametered constructors.



Suppose that we want to create the beautiful Greek sculpture of the thinking man. To begin, we introduce the class of all sculptures, and use the default constructor to materialize the default sculpture, which can be a square block of clay. Alternatively, we may use the non-default constructor to materialize a rectangular block of clay.



6.2 Class interfaces-3

Accessor member functions

- These member functions non-intrusively query an object, that is, they do so without altering its properties.

Concerning the class of sculptures, an accessor member function may report the length of the fingernails without actually clipping them.

Mutator member functions

- These member functions are able to alter the members on which they operate.

Concerning the class of sculptures, a mutator function can act like a chisel.

6.2 Class interfaces-4

Destructors

- Destructors are member functions that **delete** an object for revenge or to free up memory and prevent memory leaks.
- Transient objects are generated when we call a function to perform certain operations, and then abandoned when we exit the function. Destructors allow us to abandon the objects before exiting a function.

6.3 Class definition-1

- The “fruit” class definition has the general appearance:

```
class fruit
{
        ...
};
```

Note the **semicolon** at the end of the class definition.

- Member attributes are declared as public if they are disclosed to the main program and functions of a different class, and private otherwise. Similarly, interface functions are declared as public if they can be called from the main program and from functions of a different class, and private otherwise. This distinction motivates the class-definition structure:

```
class fruit
{
public:
        ...
private:
        ...
};
```

6.3 Class definition-2

- Default constructor

- Our first public definition is the default constructor. The fruit class definition reads:

```
class fruit
{
public:
        fruit ();
        ...
private:
        ...
};
```

Note that the default constructor does not have a return type, not even void. The name of the default constructor is identical to the class name.

6.3 Class definition-3

- To define a fruit named “kiwi” using the default constructor, we state in the main program:

```
fruit kiwi;
kiwi = fruit();
```

or

```
fruit kiwi = fruit();
```

or

```
fruit kiwi;
```

It is ***erroneous*** to declare:

```
fruit kiwi();
```

as the compiler interprets this statement as the prototype of a function named kiwi that receives no arguments and returns a fruit.

6.3 Class definition-4

- Parametered constructor

- Including also the parametered constructor, we obtain the class declaration:

```
class fruit
{
public:
        fruit();
        fruit(q, w, ..., e);
        ...
private:
        ...
};
```

Like the default constructor, the parametered constructor does not have a return type, not even void. The name of the parametered constructor is identical to the class name.

6.3 Class definition-5

- To define a fruit named “kiwi” using the parametered constructor, we state in the main program:

```
fruit kiwi;
kiwi = fruit(q_value, w_value, ..., q_value);
```

or

```
fruit kiwi = fruit(q_value, w_value, ..., q_value);
```

or

```
fruit kiwi(q_value, w_value, ..., q_value);
```

6.3 Class definition-6

- Two constructors

- Since the default and parametered constructors have identical names, they are distinguished only by the number and type of arguments enclosed by the parentheses. This duplication is consistent with the notion of function overloading: two functions with the same name are distinguished by the data types of their arguments.

- Defining a class constructor is not mandatory. If we do not declare a constructor in the class definition, the compiler will assume that the class has a default constructor with no arguments. However, it is a good idea to always define a constructor.

6.3 Class definition-7

Default destructor

- The declaration of the default destructor is similar to that of the default constructor. The class definition with the default constructor, the parametered constructor, and the default destructor reads:

```
class fruit
{
public:
        fruit();
        fruit(q, w, ..., e);
        ~fruit();
        ...
private:
        ...
};
```

To abandon kiwi, we state

```
kiwi = ~fruit()
```

6.3 Class definition-8

- Accessor function

- To query the members of the fruit class on their color, we introduce the accessor member function read_color. The class definition reads:

```
class fruit
{
public:
        fruit();
        fruit(q, w, ..., e);
        ~fruit();
        string read_color(a, b, ..., c) const;
        ...
private:
        ...
};
```

The qualifier string indicates that the function read_color will return a string of characters in the form of a word or sentence describing the color. The qualifier const indicates that the function is non-intrusive, that is, it is an accessor.

- To read the color of kiwi, we state in the main program:

```
string chroma;
chroma = kiwi.read_color (a, b, ..., c);
```

6.3 Class definition-9

Mutator function

- To convert one type of fruit into another, we introduce the mutator member function **change**. The class definition reads:

```
class fruit
{
public:
        fruit();
        fruit(q, w, ..., e);
        string read_color(a, b, ..., c) const;
        void change(g, o, ..., x);
private:
        ...
};
```

The qualifier void indicates that the function change will return neither a number, nor a word, nor a sentence, but will quietly carry out the requested operation.

- To change kiwi, we state in the main program:

```
kiwi.change (g, o, ..., x);
```

6.3 Class definition-10

- Public and private functions

- If we declare a class function in the private section of the class, then this function could be called from other class functions, but not from the main program or any other external function.

6.3 Class definition-11

- Class implementation

Now we define the precise action taken by the member functions "fruit", "read_color", and "change" of the "fruit" class.

- The implementation of the default fruit constructor reads:

```
fruit::fruit()
{
q = dv_q;
w = dv_w;
...
e = dv_e;
}
```

where "dv_q", "dv_w", etc., are specified default values that describe an object of the fruit class.

6.3 Class definition-12

- The implementation of the parametered fruit constructor reads:

```
fruit::fruit(value_q, value_w, ..., value_e)
{
q = value_q;
w = value_w;
...
e = value_e;
}
```

where “value_q”, “value_w”, etc., are specified values or names that describe an object of the fruit class.

- The implementation of the default fruit destructor reads:

```
fruit::~fruit()
{
delete q;
delete w;
...
delete e;
}
```

In this case, *q, w, ..., e* are introduced as pointers.

6.3 Class definition-13

- The implementation of the non-intrusive ***read_color*** function reads:

```
string fruit::read_color(a, b, ..., c) const
{
...
return color;
}
```

The prefix string indicates that, after operating on a member, the function ***read_color*** will return the string *color*, which can be evaluated as “red”, “green”, or any other appropriate shade.

- The implementation of the mutator change function is:

```
void fruit::change(g, o, ..., x)
{
        ...
}
```

The prefix “void” indicates that, when operating on a member, the function “change” acts quietly and returns nothing.

6.4 Private fields, public fields, and global variables-1

- Next, we discuss the “private” variables of a class. To understand this concept, it is helpful to imagine that a class is a biological cell or capsule whose interior can be accessed, probed, altered or destroyed only by the member (capsule) functions. The capsule encloses data which, if declared “private,” can be accessed only by the member functions of the host class, but not by any other functions.

6.4 Private fields, public fields, and global variables-2

- For example, if the string variable color, the string variable shape, and the real variable size are private variables of the “fruit” class, we define:

```
class fruit
{
public:
  fruit();
  fruit(q, w, ..., e);
  ~fruit();
  string read_color(a, b, ..., c) const;
  void change(g, o, ..., x);
private:
  string color;
  string shape;
  float size;
};
```

- If we want to make the color of a fruit available to the main program and any other function that uses objects of the fruit class, we must move the declaration:

  string color;

  to the public section of the class definition.

6.4 Private fields, public fields, and global variables-3

- Suppose, for example, that a function *outside* the fruit class declares

kiwi = fruit();

If color is a private field, the statement:

cout << kiwi.color;

is unacceptable. However, if color is a public field, this statement is perfectly acceptable. Class member fields are routinely kept private to prevent inadvertent evaluation in unsuspected parts of a code.

6.4 Private fields, public fields, and global variables-4

- Before proceeding to discuss specific class implementations, we emphasize two important properties regarding variable availability:

➢ The arguments of the constructor that defines an object, whether public or private, are implicitly available to the member functions. Thus, the calling arguments of a member function operating on an object include by default the arguments of the constructor that defines the object.

For example, suppose that the **vendor** member function has been defined as:

```
void fruit::vendor()
{
        ...
}
```

6.4 Private fields, public fields, and global variables-5

To operate on an apple with this function, we write:

fruit apple = fruit(value_q, value_w, ,...value_e);
*apple.**vendor**();*

The attributes of the apple do not need to be passed explicitly to the **vendor**. The first line can be shrunk into:

fruit apple(value_q, value_w, ,...value_e);

- Global variables are available to all functions of all classes. Though global variables must be declared outside the main program and any classes or functions, they can be initialized and evaluated inside the main program or any function.

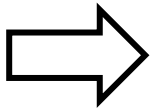
6.5 The fruit class-1

- Our definition of the fruit class involves the default constructor, a parameter constructor, and two member functions:

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

//--- CLASS FRUIT DEFINITION

class fruit
{
public:
        fruit();
        fruit(string color, string shape, float size);
        string read_color(bool Iprint) const;
        void change_color(string newcolor);
private:
        string color;
        string shape;
        float size;
};
```



By way of choice, the three fruit attributes – color, shape, and size – have been declared private.

6.5 The fruit class-2

- The implementation of the **default constructor** is:

```
fruit::fruit()
{
color = "green";
shape = "spindle";
size = 1.2;
}
```

- The implementation of the **parametered constructor** is:

```
fruit::fruit(string clr, string shp, float sz)
{
color = clr;
shape = shp;
size = sz;
}
```

6.5 The fruit class-3

- The implementation of the **non-intrusive** read_color function is:

```
string fruit::read_color(bool Iprint) const
{
if(Iprint==true)
cout << color << endl;                    // end of line
return color;
}
```

- The implementation of the **mutator** change_color function is:

```
void fruit::change_color(string clr)
{
color = clr;
}
```

6.5 The fruit class-4

- The following **main program** defines and manipulates fruit class members:

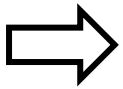
```
int main()
{
bool Iprint = true;

fruit fig = fruit();
string fig_color = fig.read_color(Iprint);        // print
cout << fig_color << endl;                 // print

fruit apple = fruit("red", "round", 2.0);
string apple color = apple.read_color(Iprint);      // print

apple.change_color("yellow");
apple_color = apple.read_color(Iprint);        // print

return 0;
}
```



- Running this program prints on the screen:

  green
  green
  red
  Yellow

6.6 Friends-1

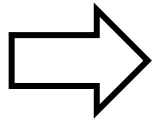
- Privacy exceptions can be made to **friends**. If we want to disclose the private fields of the class members to an external function named **package**, we state this in the class definition. In the case of the fruit class, we state:

```
//--- CLASS FRUIT DEFINITION
class fruit
{
friend void package(fruit item);
public:
        fruit();
        fruit(string color, string shape, float size);
        string read_color(bool Iprint) const;
        void change_color(string newcolor);
private:
        string color, shape;
        float size;
};
```

6.6 Friends-2

- The function **package** now has access to color, shape, and price. We will implement this function as:

```
//--- FRIEND FUNCTION
void package(fruit item)
{
if(item.size<1.0)
cout << "box" << endl;
else
cout << "crate" << endl;
}
```



The second statement will print on the screen:

crate

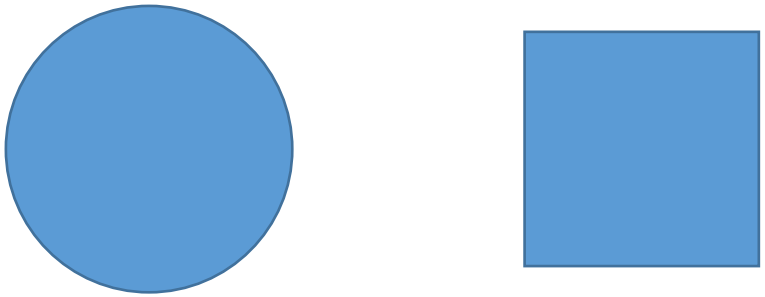
We may then state in the main program:

```
fruit watermelon("green", "oval", 12.0);
package(watermelon);
```

We recall that, if we want to disclose a private field of an object to *all* non-member functions, we must declare it as public.

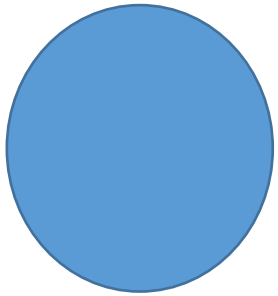
6.7 Circles and squares-1

- To further illustrate the concept of private variables, we consider a code defining two classes, one containing circles and the second containing horizontal squares.
- The circles are defined by their center and radius, and the squares are defined by their center and side length. In both cases, the *x* and *y* coordinates of the center are hosted by a two-slot vector named *center*[2].



6.7 Circles and squares-2

- The circle class definition is:



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

//--- CIRCLE CLASS DEFINITION

class circle
{
public:
        circle(double, double, double);
        void print() const;        // prohibit to change member variables
private:
        double center[2], rad;
};
```

6.7 Circles and squares-3

- The square class definition is:

```
//--- SQUARE CLASS DEFINITION

class square
{
public:
        square(double, double, double);
        void print() const;
private:
        double center[2], side;
};
```

6.7 Circles and squares-4

- The circle class implementation is:

```
//--- CIRCLE CLASS IMPLEMENTATION  (member function)
circle::circle(double center_x, double center_y, double radius)
{
center[0] = center_x;
center[1] = center_y;
rad = radius;
}

void circle::print() const
{
cout << center[0] << " " << center[1] << " " << rad << endl;
}
```

6.7 Circles and squares-5

- The square class implementation is:

```
//--- SQUARE CLASS IMPLEMENTATION
square::square(double center_x, double center_y, double edge)
{
center[0] = center_x;
center[1] = center_y;
side = edge;
}
void square::print() const
{
cout << center[0] << " " << center[1] << " " << side << endl;
}
```

Note that the variable *center* is defined separately in each class.

6.7 Circles and squares-6

- The following main program defines one object in each class and prints its properties:

```
int main()
{
circle A = circle(0.1, 0.2, 0.3);
A.print();

square B = square(0.9, 1.2, 5.3);
B.print();

return 0;
}
```

Running the code produces on the screen:

0.1 0.2 0.3

0.9 1.2 5.3

Note that the **print** statement behaves in one way when it applies to A, and in another way when it applies to B. This is an example of **polymorphism**. Polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types. The composite Greek word “polymorphism” consists of “poly,” which means “many,” and “morphi,” which means “appearance.”

6.8 Algebra on real numbers-1

- As a further example, we introduce the class of points along the *x* axis described by the *x* value and their color. If *x* is positive, the color is black, if *x* is negative, the color is red, and if *x* is zero, the color is white.

- We will endow the algebra class with several member functions that perform the following tasks:

  - Initialize a new point using the default constructor.
  - Initialize a new point using the parametered constructor.
  - Determine the color from the value of *x*.
  - Get the value of *x* and the color of a specified point.
  - Print the value of *x* and the color of a specified point.
  - Shift a point along the *x* axis.

6.8 Algebra on real numbers-1

- The algebra class definition is:

```
/*--------------------Algebra on real numbers---------------------*/
#include <iostream>
using namespace std;
                //--- CLASS DEFINITION
class algebra
{
public:
        algebra(); // default constructor
        algebra(double); // parametered constructor
        double get(string&) const;
        void print() const;
        void shift(double string);
private:
        double x;
        string color;
        string set color(float);
};
```

6.8 Algebra on real numbers-2

- The algebra class implementation is:

```
//--- CLASS IMPLEMENTATION
algebra::algebra() // default constructor
{
x=0.0;
color = "white";
}
//---
algebra::algebra(double value_x) // parametered
constructor
{
x=value_x;
color = set_color(x);
}
```

6.8 Algebra on real numbers-3

- （Continued）The algebra class implementation is:

```
string algebra::set_color(float x)        // set the color:
{
string color;
if(x>eps)
        color="black";
else if(x<-eps)
        color="red";
else
        color="white";
return color;
}
//---
double algebra::get(string& color) const
{
chroma=color;
return x;
}
```

6.8 Algebra on real numbers-4

- （Continued）The algebra class

implementation is:

```
void algebra::print() const
{
cout << x << " " << color << endl;
}
//---
void algebra::shift(double y)
{
color = set color(x+y);
x = x+y;
}
```

- Following is a main program that uses the algebra class:

```
int main()
{
string chroma;

algebra A = algebra();
A.print();
cout << A.get(chroma) << " " << chroma << endl;

algebra B = algebra(-0.9);
B.print();

B.shift(2.1);                    // -0.9+2.1 = 1.2
B.print();

return 0;
}
```

6.8 Algebra on real numbers-5

- Running this program produces on the screen:

```
0 white
0 white
-0.9 red
1.2 black
```

- Two features are worth emphasizing:
  - The get function returns the value of *x* through the function return and passes the color through an argument endowed with the reference declarator (&).
  - Because the function set color has been declared private, it cannot be called from the main program.

Short test-1+Rest (20 minutes)

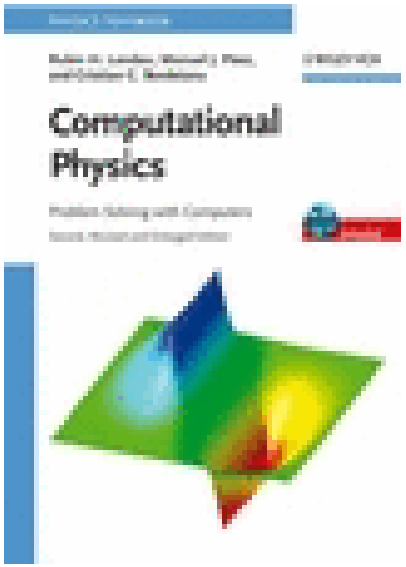
Problem: Refer to the method in section 6.5, try to output:

```
Output:
        red
        red
        yellow
        green
```

(Upload the answers with your program to OCW-i or send to cheng.s.ab@m.titech.ac.jp)

Chapter 4 Object-Oriented Programming: Kinematics

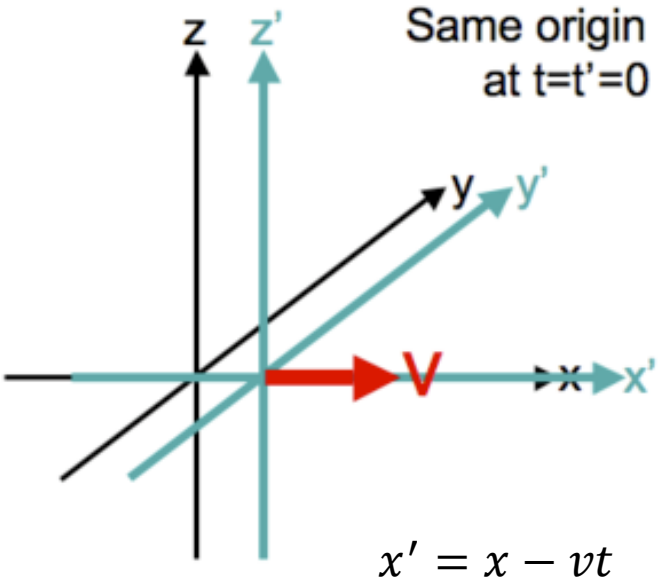
Computational physics Text p.45~



- In this chapter we provide examples of object-oriented programming (OOP) using the C++ language. Even though this subject fits in well with our earlier discussion of programming principles, we have put it off until now since we do not use explicit OOP in the projects and because it is difficult for the new programmer. We suggest that everyone read through Section 4.2.1.

4.1 Problem: Superposition of Motions

- The isotropy of space implies that motion in one direction is independent of motion in other directions. So, for example, when a soccer ball is kicked, we have acceleration in the vertical direction and simultaneous, yet independent, uniform motion in the horizontal direction.
- Our **problem** is to use the computer to describe motion in such a way that velocities and accelerations in each direction are treated as separate entities or objects, independent of motion in other directions. In this way the problem is viewed consistently by both the programming philosophy and the basic physics.



$$\boldsymbol{F} = m\boldsymbol{a}$$

$$x' = x - vt$$
$$y' = y$$
$$z' = z$$
$$t' = t$$

4.2 Theory: Object-Oriented Programming

- We will analyze the problem from an object-oriented programming viewpoint. While the objects in OOP are often graphical on a computer screen, the objects in our problem are the motions in each dimension.

- Object-oriented programming (OOP) has a precise definition. The concept is general and the *object* can be a component of a program with the properties of *encapsulation*, *abstraction*, *inheritance*, and *polymorphism*.

- Of interest to us is OOP's programming paradigm, which aims to simplify writing large programs by providing a framework for reusing components developed and tested in previous problems.

4.2.1 OOP Fundamentals-1

- A true object-oriented language has four characteristics:

➢**Encapsulation: カプセル化**

The data and the *methods* used to produce or access the data are encapsulated into an entity called an *object.* For our 1D problem, we take the data as the initial position and velocity of the soccer ball, and the object as the solution $x(t)$ of the equations of motion that gives the position of the ball as a function of time. As part of the OOP philosophy, data are manipulated only via distinct *methods*.

➢**Abstraction: 抽象化**

Operations applied to objects must give expected results according to the nature of the objects. For example, summing two matrices always gives another matrix. The programmer can in this way concentrate on solving the problem rather than on details of implementation.

4.2.1 OOP Fundamentals-2

- **Inheritance: 継承**

  Objects inherit characteristics (including code) from their ancestors, yet may be different from their ancestors. In our problem, motion in two dimensions inherits the properties of 1D motion in each of two dimensions, and accelerated motion inherits the properties of uniform motion.

- **Polymorphism: 多態性**

  Different objects may have *methods* with the same name, yet the method may differ for different objects. Child objects may have *member* functions with the same name but properties differing from those of their ancestors. In our problem, a member function *archive*, which contains the data to be plotted, will be redefined depending on whether the motion is uniform or accelerated.

4.3 Theory: Newton's Laws, Equation of Motion-1

- Newton's second law of motion relates the force vector ***F*** acting on a mass *m* to the acceleration vector ***a*** of the mass:

$$\mathbf{F} = m\mathbf{a}$$

- When the vectors are resolved into Cartesian components, each component yields a second-order differential equation

$$F_i = m\frac{d^2x_i}{dt^2} \quad (i = 1, 2, 3) \qquad (4.2)$$

- If the force in the x direction vanishes, Fx = 0, the equation of motion (4.2) has a solution corresponding to uniformmotion in the x direction with a constant velocity v0x:

$$x = x_0 + v_{0x}t \qquad (4.3)$$

- Equation (4.3) is the base or parent object in our project. If the force in the y direction also vanishes, then there also will be uniform motion in the y direction:

$$y = y_0 + v_{0y}t$$

4.3 Theory: Newton's Laws, Equation of Motion-2

- Equation (4.2) tells us that a constant force in the x direction causes a constant acceleration ax in that direction. The solution of the x equation of motion with uniform acceleration is:

$$x = x_0 + v_{0x}t + \tfrac{1}{2}a_x t^2$$

- We usually have no *x* acceleration and a negative *y* acceleration due to gravity, $a_y = -g = -9.8\ \text{m/s}^2$. The *y* equation of motion is then

$$y = y_0 + v_{0y}t - \tfrac{1}{2}gt^2$$

4.4 OOP Method: Class Structure-1

- The class structure we use to solve our problem contains the objects:

➢**Parent class Um1D:** 1D uniform motion for given initial conditions,

➢**Child class Um2D:** 2D uniform motion for given initial conditions,

➢**Child class Am2d:** 2D accelerated motion for given initial conditions.

- The member functions include

➢**x:** position after time *t*,

➢**archive:** creator of a file of position versus time.

For our projectile motion, encapsulation is the combination of the initial conditions (x_0, v_{x0}) with the member functions used to compute $x(t)$.

4.4 OOP Method: Class Structure-2

- Our member functions are:
  - the creator of the class of uniform 1D motion Um1D,
  - its destructor ~Um1D, and
  - the creator x(t) of a file of *x* as a function of time *t*.
  - *Inheritance* is the child class Um2D for uniform motion in both *x* and *y* directions, it being created from the parent class Um1D of 1D uniform motion.
  - *Abstraction* is present (although not used powerfully) by the simple addition of motion in the *x* and *y* directions.
  - *Polymorphism* is present by having the member function that creates the output file be different for 1D and 2D motions.
  - In our implementation of OOP, the class Am2D for accelerated motion in two dimensions inherits uniform motion in two dimensions (which, in turn, inherits uniform 1D motion), and adds to it the attribute of acceleration.

4.5 Implementation: Uniform 1D Motion, unim1d.cpp

- For 1D motion we need a program that outputs positions along a line as a function of time, (x, t).
- For 2D motion we need a program that outputs positions in a plane as a function of time (x, y, t).
- Time varies in discrete steps of Δt =delt = T/N, where the total time T and the number of steps N are input parameters.

- Our parent class Um1D of uniform motion in one dimension contains
  - **x00:** the initial position,
  - **delt:** the time step,
  - **vx0:** the initial velocity,
  - **time:** the total time of the motion,
  - **steps:** the number of time steps.
- To create it, we start with the C++ headers:

```
#include <stdio.h>                    /* Input-output libe */
#include <stdlib.h>                        /* Math libe */
```

4.5.1 Uniform Motion in 1D, Class Um1D-1

- The encapsulation of the data and the member functions is achieved via Class Um1D:

```
class Um1D  {                                   /* Create base class
*/
  public:
  double x00,delt,vx,time;          /* Initial position, velocity, dt */
  int steps;                                          /* Time steps */
  Um1D(double x0,double dt,double vx0,double ttot); /* Constructor */
  ~Um1D(void);                                    /* Class Destructor */

  double x(double tt);                                        /* x(t) */
    void archive();                          /* send  x vs t to file */
};
```

- Next, the variables *x0*, *delt*, *vx*, and *time* are initialized by the constructor of the class Um1D:

```
Um1D::Um1D(double x0,double dt,double vx0,double ttot) {
                                                  /* Constructor Um1D */
  x00 = x0;
  delt = dt;
  vx = vx0;
  time = ttot;
  steps = ttot/delt;
}
```

4.5.1 Uniform Motion in 1D, Class Um1D-2

- After that, we make the destructor of the class, which also prints the time when the class is destroyed:

```
Um1D::~Um1D(void)  {                    /* Destructor of class Um1D */
  printf("Class Um1D destroyed \ n");
}
```

- Given the initial position *x*0, the member function returns the position after time *dt*:

```
double Um1D::x(double tt) {                              /* x=x0+dt*v
*/
  return x00+tt*vx;
}
```

4.5.1 Uniform Motion in 1D, Class Um1D-3

- The algorithm is implemented in a member routine, and the positions and times are written to the file Motion1D.dat:

```
void Um1D::archive()  {                    /* Produce x vs t file
*/
  FILE  *pf;
  int i;
  double xx, tt;
  if ( (pf = fopen("Motion1D.dat","w+")) == NULL ) {
    printf("Could not open file \ n");
    exit(1);
  }
  tt = 0.0;
  for ( i = 1; i <= steps; i++ ) {
    xx = x(tt);                            /* Computes x=x0+t*v */

    fprintf(pf," %f  %f \ n",tt,xx);
    tt = tt+delt;
  }
  fclose(pf);
}
```

4.5.1 Uniform Motion in 1D, Class Um1D-4

- The main program defines an object (class) **unimotx** of type Um1D and gives initial numeric values to the data:

```
main() {
  double inix, inivx, dtim, ttotal;
  inix = 5.0;
  dtim = 0.1;
  inivx = 10.0;
  ttotal = 4.0;
  Um1D unimotx(inix, dtim, inivx, ttotal);     /* Class constructor */
  unimotx.archive();                          /* Produce y vs x file */
}
```

4.5.2 Implementation: Uniform Motion in 2D, Child Um2D, unimot2d.cpp-1

- The first part of the program is the same as before. We now make the child class Um2D from the class Um1D.

```
#include <stdio.h> #include <stdlib.h> class Um1D {
/* Base class created */
  public:
  double x00, delt, vx, time;     /* Initial conditions, parameters */
  int steps;                                                /* Time step */
                        /* constructor */
  Um1D(double x0, double dt, double vx0, double ttot);
  ~Um1D(void);                                          /* Class destructor */

  double x(double tt);                                          /* x=xo+v*t */
    void archive();                                 /* Send x vs t to file */
};
                                                          /* Um1D Constructor */
  Um1D::Um1D(double x0, double dt, double vx0, double ttot)  {
     x00 = x0;
     delt = dt;
     vx = vx0;
     time = ttot;
     steps = ttot/delt;
  }
  Um1D::~Um1D(void)  {                          /* Class Um1D destructor */
     printf("Class Um1D destroyed \ n");
  }
  double Um1D::x(double tt)  {                                  /* x=x0+dt*v */
```

4.5.2 Implementation: Uniform Motion in 2D, Child Um2D, unimot2d.cpp-2

Continued:

```
    return x00+tt*vx;
  }
  void Um1D::archive() {                          /* Produce x vs t file */
    FILE  *pf;
    int i;
    double xx, tt;
    if ( (pf = fopen("Motion1D.dat", "w+"))==NULL ) {
      printf("Could not open file \ n");
    exit(1);
    }
    tt = 0.0;
    for ( i = 1; i <= steps; i++ ) {
      xx = x(tt);                                  /* computes x=x0+t*v */
      fprintf(pf,"%f  %f \ n", tt, xx);
      tt = tt+delt;
    }
    fclose(pf);
}
```

4.5.3 Class Um2D: Uniform Motion in 2D-1

- To include another degree of freedom, we define a new class that inherits the *x* component of uniform motion from the parent Um1D, as well as the *y* component of uniform motion from the parent Um1D. The new data for the class are the initial *y* position y00 and the velocity in the *y* direction vy0. A new member y is included to describe the *y* motion. Note, in making the constructor of the Um2D class, that our interest in the data *y* versus *x* leads to the member archive being redefined. This is polymorphism in action.

4.5.3 Class Um2D: Uniform Motion in 2D-2

```
class Um2D : public Um1D {                    /* Child class, parent Um1D
*/
  public:                                  /* Data accessible to other code */
  double y00, vy;                    /* member functions accessible to all */
  Um2D(double x0,double dt,double vx0,double ttot,double y0,double
       vy0);
  ~Um2D(void);                                /* destructor of Um2D class */

  double y(double tt);                          /* Added motion for 2D */
    void archive();                             /* redefinition for 2D */
};
```

- Observe how the Um2D constructor initializes the data in Um1D, y00, and vy:

```
Um2D::Um2D(double x0, double dt, double vx0, double tott,
       double y0, double vy0):
Um1D(x0,dt,vx0,tott)  {
   y00 = y0;
   vy  = vy0;
}
```

4.5.3 Class Um2D: Uniform Motion in 2D-3

- The destructor of the new class is

```
Um2D::~Um2D(void) {
   printf("Class Um2D is destroyed \ n");
}
```

- The new member of class Um2D accounts for the y motion:

```
double Um2D::y(double tt)  {
   return y00+tt*vy;
}
```

4.5.3 Class Um2D: Uniform Motion in 2D-4

- The new member function for two dimensions contains the data of the *y* and *x* positions, and incorporates the polymorphism property of the objects:

```
void Um2D::archive()  {                    /* Uniform motion in 2D
*/
  FILE  *pf;
  int i;
  double xx, yy, tt;
  if ( (pf = fopen("Motion2D.dat","w+") ) == NULL )  {
   printf("Could not open file \ n");
   exit(1);
  }
  tt = 0.0;
  for (i = 1; i <= steps; i++) {
    xx = x(tt);                          /* uses member function x */
    yy = y(tt);                         /* add the second dimension */
    fprintf(pf,"%f  %f \ n", yy, xx);
    tt = tt + delt;
  }
  fclose(pf);
}
```

4.5.3 Class Um2D: Uniform Motion in 2D-5

- The differences with the previous main program are the inclusion of the *y* component of the motion and the constructor unimotxy of class type Um2D:

```
main() {
  double inix, iniy, inivx, inivy, dtim, ttotal;
  inix = 5.0;
  dtim = 0.1;
  inivx = 10.0;
  ttotal = 4.0;
  iniy = 3.0;
  inivy = 8.0;
                                          /* class constructor */
  Um2D unimotxy(inix, dtim, inivx, ttotal, iniy, inivy);
  unimotxy.archive();                  /* To obtain file of y vs x */
}
```

Short test-2

Problem: Add to the circle class a member function that computes and prints the area of a circle, and to the square class a member function that computes and prints the area of a square.

(Upload the answers with your program to OCW-i or send to cheng.s.ab@m.titech.ac.jp)

Tokyo Tech

Thank You