ALGORITHMS AND DATA STRUCTURES

# CLASSES IN C++[14]

# REPRESENTING ABSTRACT DATA TYPES

- Implementation of ADT compels us to talk about language details. In C++ we need to introduce `class`es
- `class`es make it possible to combine several related pieces of information into a composite value that can be manipulated as a unit
- All DS will be objects that store data and have functions to manipulate those data; this is done in C++ using a `class`
- As an example, suppose that you want to work with integer coordinates in a *x-y* grid. **How to implement this?**

# STRUCTURE TYPE NAMES

- **Structure**: a combination of simple types names that are predefined

```
struct Point {
    int x;    // --> member or field
    int y;    // --> member or field
};
```

- This code defines the type `Point` as a structure with two components. In a structure, components are called fields or members
- The definition introduces a new type and does not declare any variables
- Once defined, you can use the type name to declare variables

```
Point p;
```

# STRUCTURE TYPE NAMES

- Can select the individual fields using the dot operator, which is written in the form

```
p.x // var.name; var =  struct variable, name = field
```

- The fundamental characteristic of a `struct` is that can be viewed both as a collection of individual fields and as a single value
- At the lower levels of the implementation, the values stored in the individual fields are likely to be important.
- At higher levels of detail, it makes sense to focus on the type as an integral unit

# CLASSES IN C++

- It is like a structure type with variable encapsulation. Its basic syntax is

```
class Point {
public:             // --> PUBLIC SECTION
    int x;          // --> instance variable [field or
    int y;          // --> instance variable  member]
};
```

- A `struct` = `class` with only `public` section. There is also a `private` section; they determine the visibility of the class members
- A member that is `public` may be accessed by any method in any class
- A `private` member may only be accessed by methods in its class

# CLASSES IN C++

- Typically, data members are declared `private` to restrict access
- While methods intended for general use are made `public`
- `class`es are composed by their members, which can be instance variables or methods (functions)
- when declaring an instance of a `class`, i.e. a variable or object, its creation is handled by the constructor which always has the same name of the `class`

# CLASSES IN C++

- A **constructor** is a method that describes how an instance of the `class` is constructed
- There are several types of constructors:
  1. Default constructor
  2. Copy constructor
  3. Explicit constructor
  4. Dynamic constructor
  5. Move constructor

# CLASSES IN C++

1. Default constructor: a constructor which can be called with no arguments (either defined with an empty parameter list, or with default arguments provided for every parameter)

```
struct A
{
    int x;
    // user-defined default constructor
    A(int x = 1): x(x) {}
};
```

# CLASSES IN C++

2. A Copy constructor of class T is constructor whose first parameter is T&, `const` T&, and either there are no other parameters, or the rest of the parameters all have default values [incomplete definition]

```
struct A
{
    int n;
    A(int n = 1) : n(n) {}
    // user-defined copy constructor
    A(const A& a) : n(a.n) {}
};
```

3. Explicit constructor: specifies that a constructor is explicit, that is, it cannot be used for implicit conversions and copy-initialization

```
struct A
{
    int n;
    A(int) {}          // converting constructor
    explicit A(int) {} // explicit constructor
};
```

- A constructor with a single non-default parameter that is declared without the function specifier explicit is called a converting constructor

# CLASSES IN C++

4. A constructor in which the memory for data members is allocated dynamically is called a dynamic constructor

```
struct A
{
    int *marks;
    // dynamic constructor
    A(int n) {
        marks = new int;
        marks = 3;
    }
};
```

# CLASSES IN C++

5. A move constructor of class T is a constructor whose first parameter is T& or `const T&`, and either there are no other parameters, or the rest of the parameters all have default values [incomplete definition]

```
struct A
{
    std::string s;
    A() : s("test") { }
    // move constructor
    A(A&& o) : s(std::move(o.s)) {}
};
```

# CLASSES IN C++

- Functions are called **methods** in OOP language. Two types:
- We have ***accessors*** or getters which extract information about instance variables, *without changing the state* of its object
- We also have ***mutators*** or setters that modify the information kept in the fields, i.e. *they change the object's state*
- Mutators cannot be applied to constant objects. By default, all member functions are mutators

# BACK TO THE POINT CLASS EXAMPLE...

```
class Point {
public:                              // PUBLIC SECTION
    Point() {                        // default constructor
        x = 0;
        y = 0;
    }
    Point(int xc, int yc) {          // param. constructor
        x = xc;
        y = yc;
    }
    int getX() { return x; }         // accessor
    int getY() { return y; }         // or getter
    void setX(int v) { x = v; }      // mutator
    void setY(int v) { y = v; }      // or setter

private:                             // PRIVATE SECTION
    int x, y;                        // member or fields
};
```

# IMPLEMENTATION AND INTERFACE

- Usually interface and implementation are separated in two files: `*.hh` or `*.hpp` for the interface and `*.cc` or `*.cpp` for the implementation
- In the implementation each member method must identify the class that it is part of
- The syntax is `ClassName::member`. The `::` is called the **scope resolution operator**
  - Let's use the `Point` class as an example

# INTERFACE AND IMPLEMENTATION OF TYPE POINT

```cpp
#ifndef _point_h_
#define _point_h_

#include <string>

class Point {
public:
    Point();
    Point(int xc, int yc);

    int getX();
    int getY();

    void setX(int v);
    void setY(int v);

private:
    int x, y;
};

#endif //_point_h_
```

```cpp
#include "point.hpp"

Point::Point() {
    x = 0;
    y = 0;
}
Point::Point(int xc, int yc) {
    x = xc;
    y = yc;
}

int Point::getX() { return x; }
int Point::getY() { return y; }

void Point::setX(int v) { x = v; }
void Point::setY(int v) { y = v; }
```

# OPERATOR OVERLOADING

- Operator overloading: technology that makes it possible to extend the standard operators so that they apply to new types [objects]
- We can define operator to add, subtract, and multiply `Points` by overloading the corresponding operators
- Can also test if two Points are equal using the `operator==` [`friend` keyword is necessary]
- Operators can be overloaded not only for `class`es but also for `enum`erated types

# OPERATOR OVERLOADING FOR POINT

```
class Point {
public:
    // other methods here...
    friend bool operator==(Point p, Point q);
    friend ostream & operator<<(ostream & os, Point pt);

    // private section...
};

bool operator==(Point p, Point q) {
    return p.x == q.x && p.y == q.y;
}

ostream & operator<<(ostream & os, Point p) {
    return os << "(" << p.x << "," << p.y << ")";
}
```

# OPERATOR OVERLOADING FOR POINT

```
int main(void) {
    Point a(1, 3), b(1, 3);

    cout << "a = " << a << endl << "b = " << b << endl;
    string msg = a == b ? "" : "not ";
    cout << "Points are " + msg + "equal" << endl;

    return 0;
}
```

# MEMBER VS NONMEMBER OVERLOADING

- Assignment ‘`=`’, subscript ‘`[]`’, call ‘`()`’, and member access arrow ‘`->`’ operators **must** be defined as members
- `I/O` operators **must** be nonmember functions

# MEMBER VS NONMEMBER OVERLOADING

- Compound-assignment operators **usually should** be members; though it’s not a requirement
- Relational (‘<’, ‘>=’, …), equality ‘==’, and arithmetic (‘+’, ‘−’, ‘*’, …) operators **usually should** be nonmember functions
- Changing-state operators: ‘++’, ‘&’, **should** be members

# OPERATOR OVERLOADING

- Operators that change the state of the object **should** be defined as members of the class
- Output operators **should** print contents, with minimal formatting
- Input operators **must** consider with the fact that the input might fail and decide what to do in such cases

# OPERATOR OVERLOADING

- Assignment and compound-assignment operators **should return** a reference to the left-hand operand
- The prefix operators **should return** a reference to the object that changed, similar to built-in operators
- The postfix operators **should return** as value the old object, same as before

# SUMMARY

```
class intCell {
public:                                 // --> PUBLIC SECTION
    intCell(                            // --> constructor
            const int val = 0)          // --> constant parameter
    : member{val} {                     // --> initialization list
                                        // --> constructor's body
    }

    int read(void)                      // --> definition of accessor
    const {                             // --> constant member method
        return member;                  // --> accessor's body
    }

    void write(const int val) {         // --> mutator method
        member = val;                   // --> mutator's body
    }

private:                                // --> PRIVATE SECTION
    int member;                         // --> instance variable
};
```

```
// clang++ -std=c++14 -Wall -Werror explicit.cc
#include <iostream>
using namespace std;

int main(void) {
    intCell obj;                      // --> empty constructor
    intCell bjo{45};                  // --> one-param constructor
    intCell job{};                    // --> zero-param constructor
    obj = 32;                         // --> implicit type convert
//  obj = intCell(32);                // --> if explicit is used
    cout << obj.read() << endl;
    return 0;
}
```