Cairo University, Faculty of Computers and AI

CS213 – 2022 / 2023

Programming II

Lecture 2: OOP - I

By

Dr. Mohammad El-Ramly

Lecture Objective / Content

1. Object-orientation in C++
2. A first example
   - C-style structures vs. C++ classes
3. History of OOP

The C++ Language

- Bjarne Stroupstrup, the language's creator

  C++ was designed to provide Simula's facilities for program organization together with C's efficiency and flexibility for systems programming.

The C++ Language

“C makes it easy to shoot
yourself in the foot;
C++ makes it harder,
but when you do,
It blows your whole leg off.”

- Bjarne Stroustrup

The C++ Language

- C++ is an **object-oriented** extension of C
- Stroustrup original interest at Bell Labs was research on simulation
- Early extensions to C are based primarily on Simula
- Called “C with classes” in early 1980s
- Features were added incrementally
    - Classes, templates, exceptions, multiple inheritance, type tests...

How Successful?

- Many users, tremendous popular success
- Given the design goals and constraints, very well-designed language
- Very complicated design, however
  - Many features with complex interactions
  - Difficult to predict from basic principles
  - Most serious users chose subset of language
    - Full language is complex and unpredictable
  - Many implementation-dependent properties
    - Many details were not specified in the standard

Non-Object-Oriented Additions

- Things we studied
  - Pass-by-reference
  - User-defined function / operator overloading
  - Boolean type
  - Templates (generic programming)
- Things we will study
  - Exceptions

1. C++ Object System

- Classes define new types
- Objects are instances of classes
- Inheritance
  - Single and multiple inheritance
- Encapsulation

What is OO ?

- ***Object-orientation*** is a way of thinking about problems using models built from real-world concepts.
- The fundamental unit is the ***Object***
- An object has ***data*** and ***behavior***
- ***OO Software*** means we write our program in terms of objects, each tightly integrates data and operations on the data
- In ***Structured programming***, data and operations on the data were separated or loosely related.

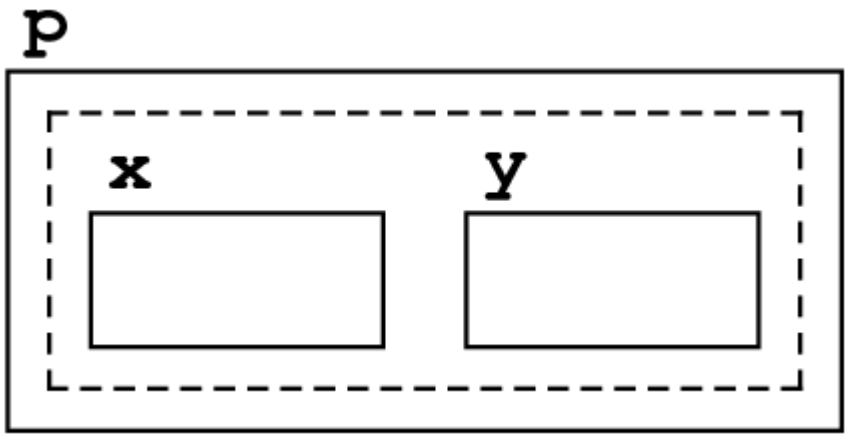
`struct` Point

- ```
  struct Point {
      int x;
      int y;
  ```
- ```
  };
  ```

- A point has two ***fields, attributes*** or ***data members***
- You need to define functions to operate on points and pass them copies of this structure

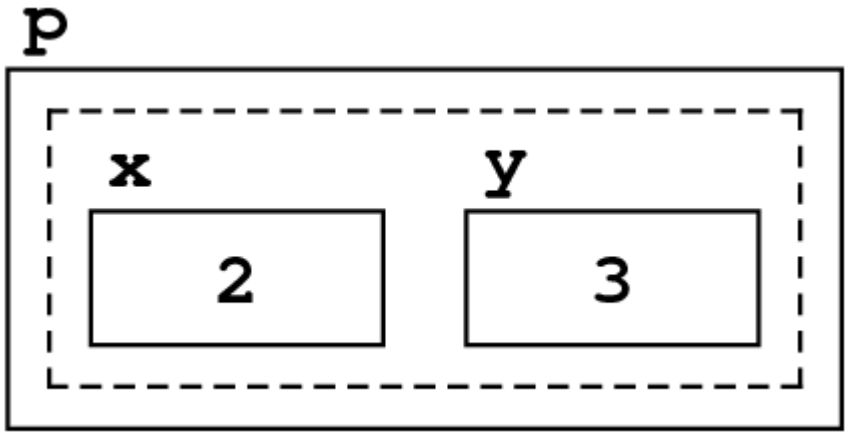
`struct` Point

- `Point p;`



- Use the ***dot operator*** to access members

- `p.x = 2;`
- `p.y = 3;`



`Struct` Point

- **`Point movePoint (int newX, int newY,`**
  `Point& point)`

```
{
    point.x = newX;
    point.Y = newY;
};
```

- We define **independent** or **free** functions that take copies of **`struct`** to operate on.

struct Point

```
#include <bits/stdc++.h>
using namespace std;

struct Point {
      int x = 0;
      int y = 0;
      Point (int newX, int newY);
};

Point::Point (int newX, int newY){
      x = newX;
      y = newY;
}

int getX (Point pnt) {
      return pnt.x;
}

int getY (Point pnt) {
      return pnt.y;
}
```

Data

Separate from

Functions
that work on them

struct Point

```
string toString(Point pnt) {
      return "(" + to_string(pnt.x) + "," +
                   to_string(pnt.y) + ")";
}

ostream & operator<<(ostream & os,  Point & pnt) {
      os << toString(pnt);
}

int main () {
      Point p1(9,4);
      Point p2(9,4);
      Point p3(3,6);

      cout << "\nP1 = " << p1;
      cout << "\nP2 = " << p2;
      cout << "\nP3 = " << p3;
}
```

`Class` Point

- ```
  class Point {
      int x;
      int y;
      ......
  };
  ```

- A point **object** has two ***fields, attributes*** or ***data members***
- And we can define inside ***functions*** or ***methods*** or ***function members***.
  ```

class Point

```
/*
 * Class: Point
 * ------------
 * This class represents an x-y coordinate point on a two-dimensional
 * integer grid.
 */

#include <string>
#include "strlib.h"
using namespace std;

class Point {

public:

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

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

   int getX() {
      return x;
   }

   int getY() {
      return y;
   }

   string toString() {
      return "(" + integerToString(x) + ","
                 + integerToString(y) + ")";
   }

private:
   int x;
   int y;

};
```

Constructors

Getter methods

Public section

Instance variables

Private section

`class` Point

- ```
  Point () {          // Default constructor
     x = y = 0;
  }
  ```
- ```
  Point (int newX, int newY){
     x = newX;
     y = newY;
  }
  ```
- *Constructor* is a function that is called automatically when a Point object is created.
- *Default constructor* takes no parameters.

`class` Point

- ```
  int getX() {
      return x;
  }
  ```
- ```
  int getY() {
      return y;
  }
  ```
- *Getter (accessors)* methods are used to access class's private data.

`class` Point

- `void setX(int newX) {`
  `x = newX;`
  `}`
- `void setY(int newY) {`
  `y = newY;`
- `}`
- *Setter (mutators)* are methods used to change class's private data (if needed).
- Classes with no setters are *immutable*.

2. Example: A `struct` C-style stack

```
const int SIZE = 20;

struct Stack {
  char data[SIZE];
  int  size;
};

Stack create() {
  Stack s;
  s.size = 0;
  return s;
}
```

Example: A Better C stack

```
char pop(Stack& s) {
  if (s.size = 0) error("Underflow");
  return s.data[--(s.size)];
}

void push(Stack& s, char v) {
  if (s.size == SIZE) error("Overflow");
  s.data[s.size++] = v;
}

void error (string message) {
  cout << "\n" << message << "\n";
  exit (1);
}
```

C++ Solution: Class

```
class Stack {
  private:
      char data[SIZE];
      int  size;

  public:
      Stack () {size = 0;}

      char pop() {
            if (size == 0) error("Underflow");
            return data[--size];
      }

      void push(char v) {
            if (size == SIZE) error("Overflow");
            data[size++] = v;
      }
};
```

Definition of both representation and operations

Public: visible outside the class

Constructor: initializes

Member functions see object fields like local variables

C++ Stack Class

- Natural to use

```
Stack st;
st.push('a');
st.push('b');
char d = st.pop();

Stack *stk = new Stack;
stk->push('a');
stk->push('b');
char d = stk->pop();
```

Another C++ Solution – Same Interface

```
class Stack {
  private:
      vector<char> data;

  public:
      Stack () {
      }

      char pop() {
            if (v.empty()) error("Underflow");
            char c = data[data.size()-1];
            data.pop_back();
            return c;
      }

      void push(char v) {
            data.push_back(v);
      }
};
```

`vector` Operations

Member functions

(constructor)	Construct vector (public member function)
(destructor)	Vector destructor (public member function)
operator=	Copy vector content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)

Capacity:

size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)

`vector` Operations

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)

Modifiers:

assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)

Allocator:

get_allocator	Get allocator (public member function)

Class Implementation

- C++ compiler translates to C-style implementation

C++

```
class Stack {
  char s[SIZE];
  int sp;

public:
  Stack()
  void push(char);
  char pop();
};
```

Equivalent C implementation

```
struct Stack {
  char s[SIZE];
  int sp;
};

void st_Stack(Stack*);
void st_push(Stack*, char);
char st_pop(Stack*);
```

3. History of OOP

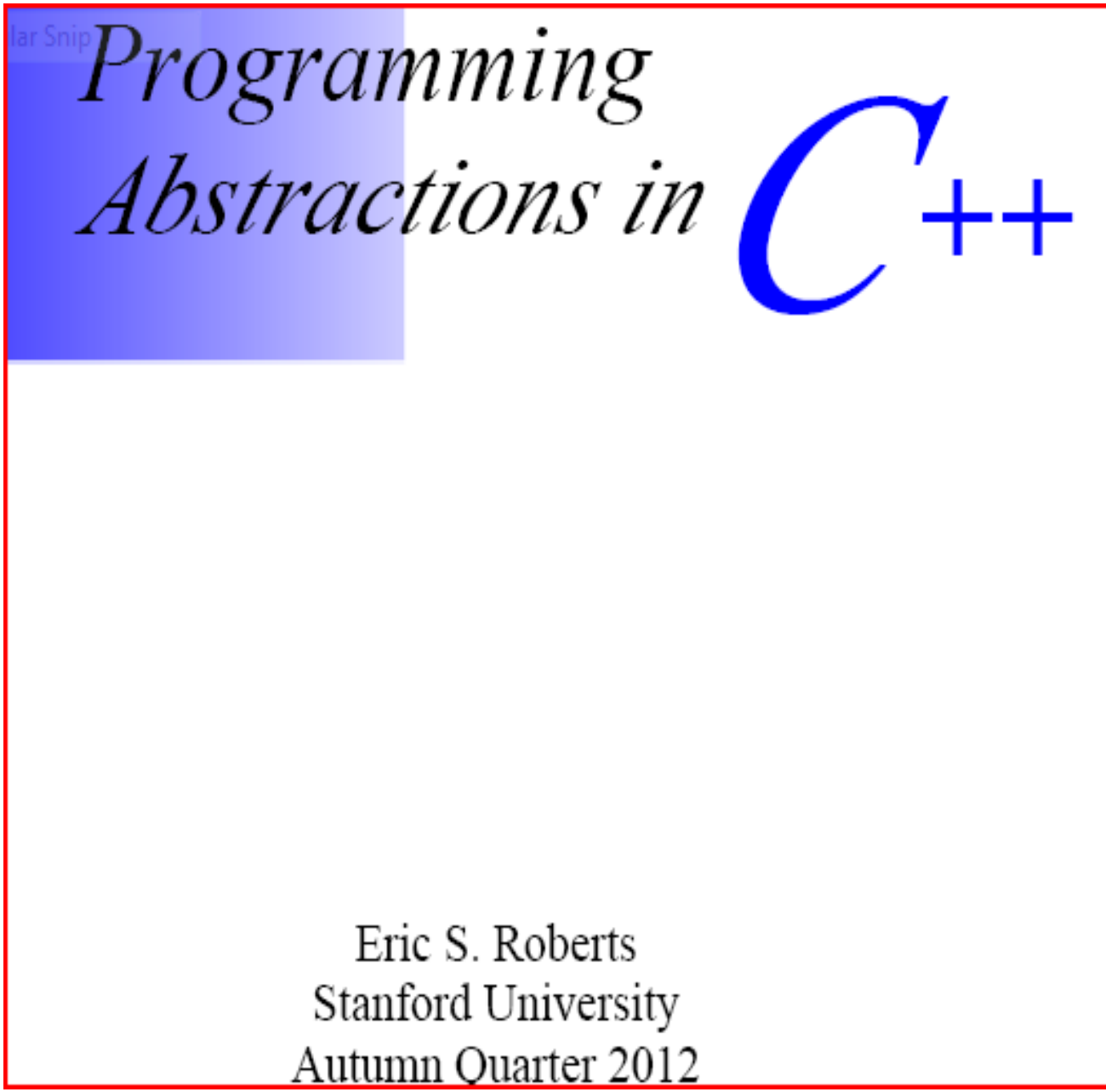
- The first OOP language was Simula-67
  - For writing simulation programs
- In the early 1980's, Smalltalk by Xerox
  - New syntax, large library of reusable code, bytecode, platform independence, garbage collection.
- Late 1980's, C++ was developed
  - Advantages of OO + tremendous numbers of C programmers
- In 1991, Sun Microsystems started a project on a language for consumer 'smart devices': Oak
  - When the Internet gained popularity, Sun saw an opportunity to exploit the technology.
  - The new language, renamed Java, was formally presented in 1995 at the SunWorld '95 conference.

3. History of OOP

- The first OOP language was Simula-67
  - For writing simulation programs
- In the early 1980's, Smalltalk by Xerox
  - New syntax, large library of reusable code, bytecode, platform independence, garbage collection.
- Late 1980's, C++ was developed
  - Advantages of OO + tremendous numbers of C programmers
- In 1991, Sun Microsystems started a project on a language for consumer 'smart devices': Oak
  - When the Internet gained popularity, Sun saw an opportunity to exploit the technology.
  - The new language, renamed Java, was formally presented in 1995 at the SunWorld '95 conference.

Readings of Week 1

- Chapters 1, 2, 3, 4, 6, 11
- You must read and try the examples in these chapters.
- Solve the exercises at chapters' ends.



- Watch https://www.youtube.com/watch?v=uMR6SX266F4&list=PLLhe0ZInsJiV_fTY_68CtTT-QwyfwG2GL&index=4
- You must read it from the book:
  - Problem Solving in C++, Chap 10
  - Prog. Abstractions in C++, Chap 6.1 to 6.3