Introduction to the C programming language

From C to C++: Stack and Queue

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

February 24, 2010

Outline

Outline

Abstract data types

- An important concept in programming is the *Abstract Data Type* (ADT)
- An abstract data type is a user-defined type, that can be used similarly to built-in data types
- An ADT defines
  - What kind of values the data type can assume (*domain*)
  - What operations we can perform on the data type
- How the data and the operations are implemented is *hidden* to the user, and it is part of the implementation

ADT in C

- ADT are a general concept that can be supported in any language, including Assembler, Basic, C
- Example of ADT in C

```
struct point {
    double x, y;
    int z;
};

void point_read(ifstream &in, point *p);
void point_save(ofstream &out, point *p);
void point_print(point *p);
```

  - The structure defines the *domain* (i.e. how the data is composed by three components)
    The three functions define the *operations* we can do on the data

- It is not very nice to program ADT in C, because there is little support from the language. ADT are well supported in *Object Oriented* (OO) languages

Example in C++

- C++ is the OO version of C
- It maintains a similar syntax, adding new keywords and constructs
- How the previous class can be expressed in C++?

```
class Point {
  double x, y;
  int z;
public:
  Point(double x1, double y1);
  Point();
  void read(ifstream &in);
  void save(ofstream &out);
  void print();
  double get_x();
};
```

Example in C++

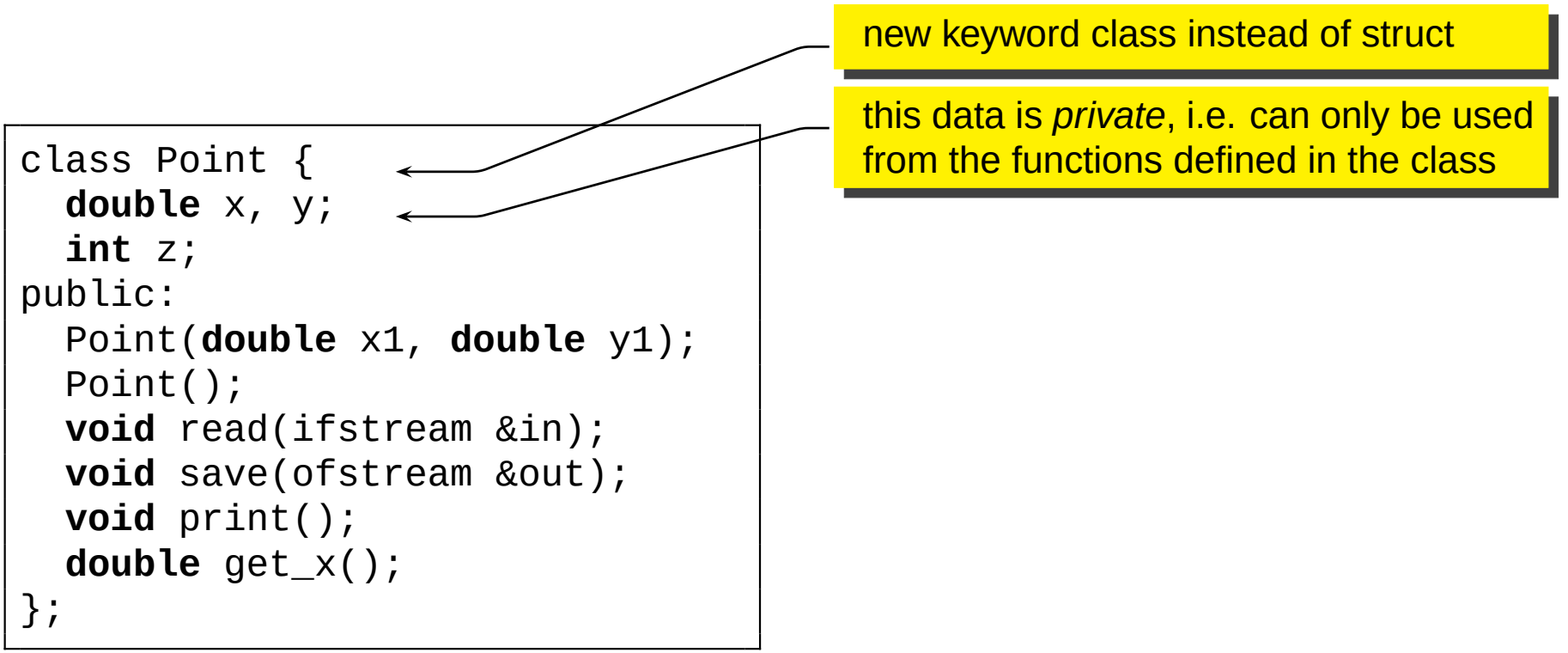
- C++ is the OO version of C
- It maintains a similar syntax, adding new keywords and constructs
- How the previous class can be expressed in C++?

new keyword class instead of struct

```
class Point {
  double x, y;
  int z;
public:
  Point(double x1, double y1);
  Point();
  void read(ifstream &in);
  void save(ofstream &out);
  void print();
  double get_x();
};
```

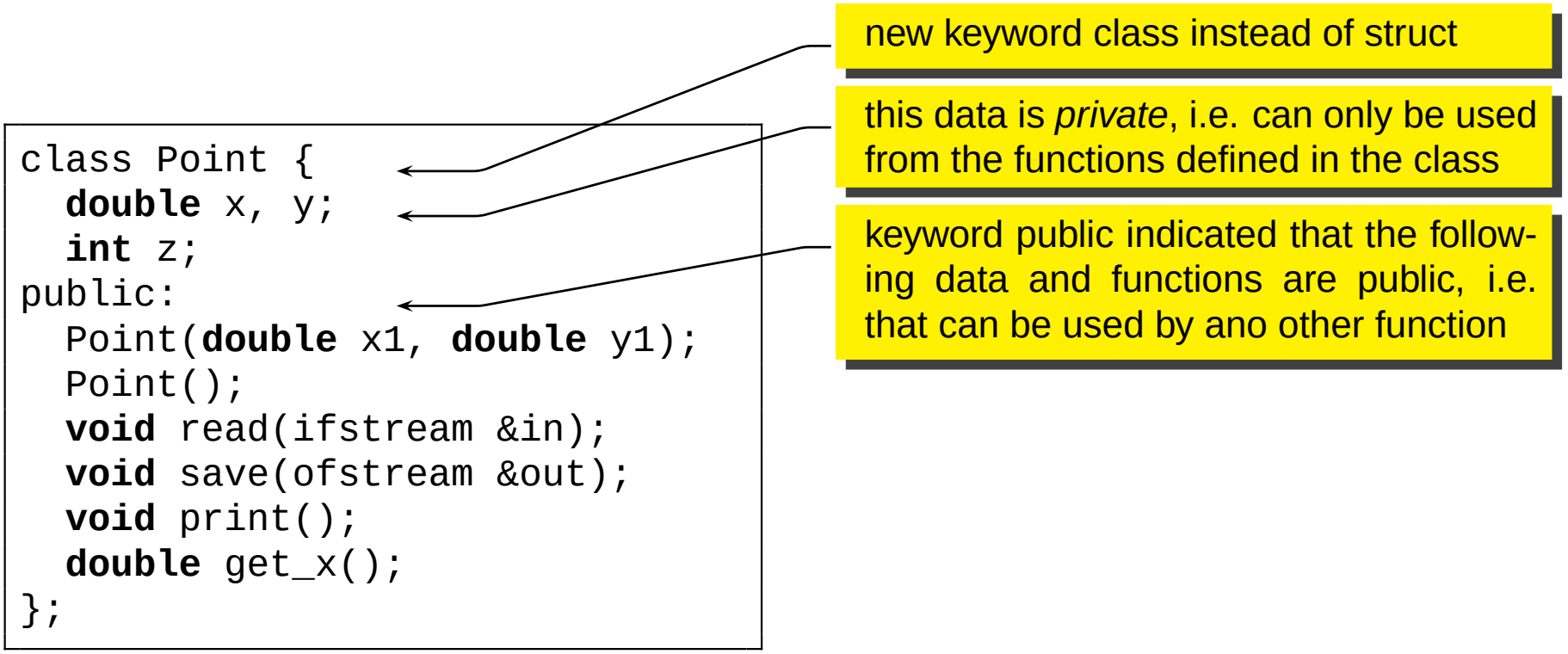
Example in C++

- C++ is the OO version of C
- It maintains a similar syntax, adding new keywords and constructs
- How the previous class can be expressed in C++?



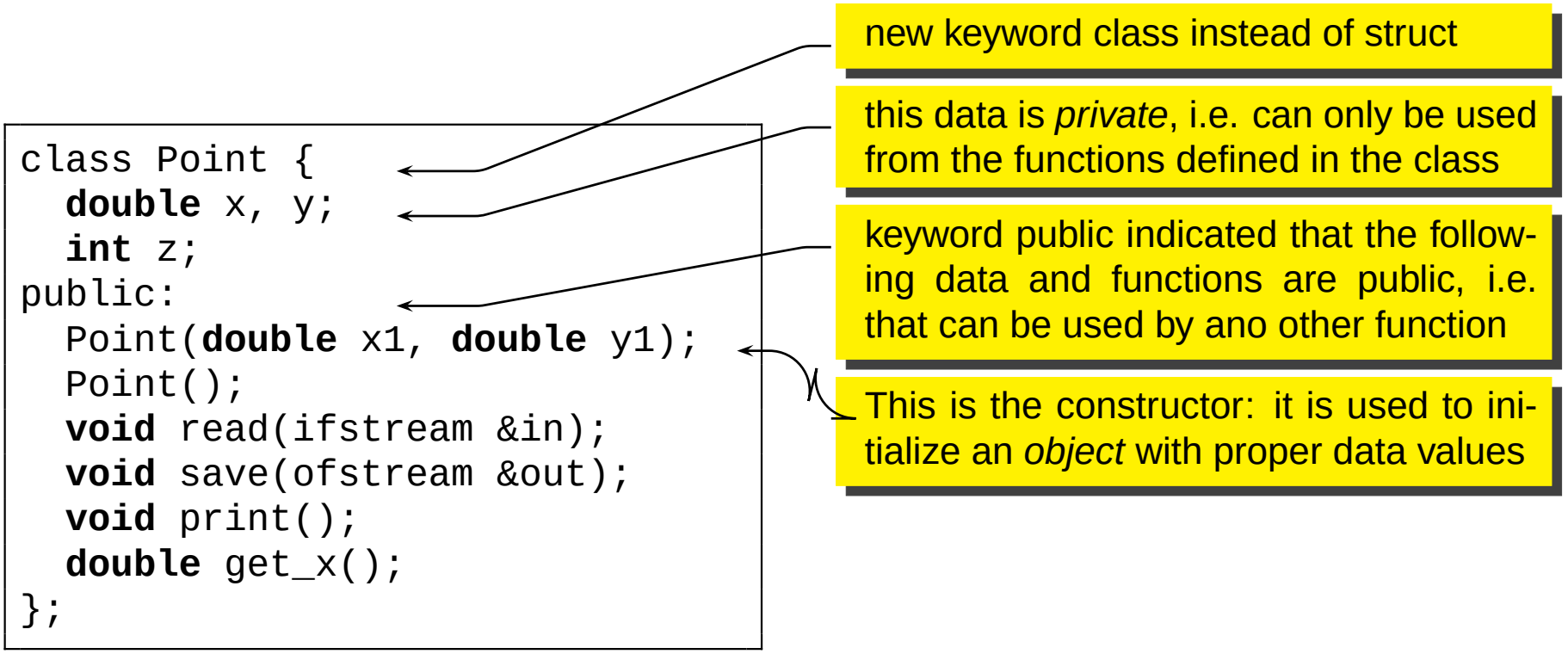
Example in C++

- C++ is the OO version of C
- It maintains a similar syntax, adding new keywords and constructs
- How the previous class can be expressed in C++?



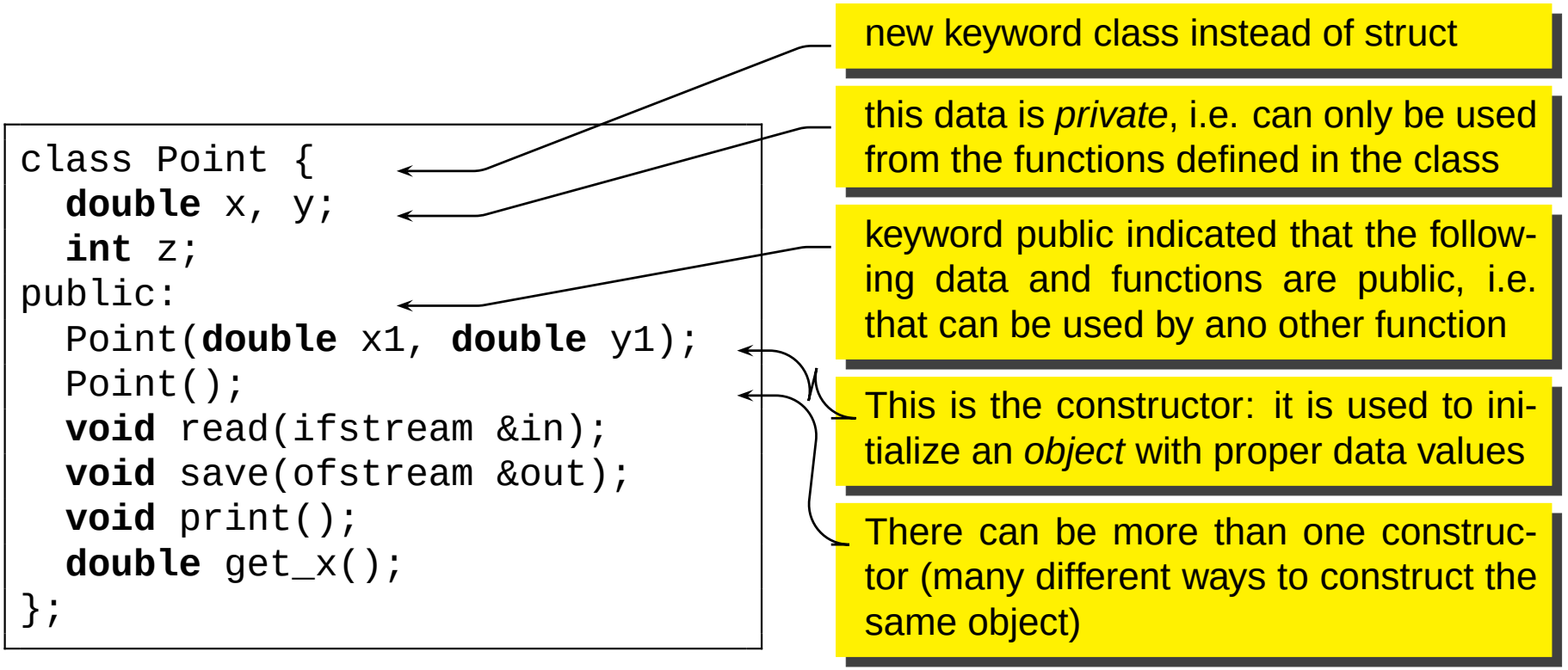
Example in C++

- C++ is the OO version of C
- It maintains a similar syntax, adding new keywords and constructs
- How the previous class can be expressed in C++?



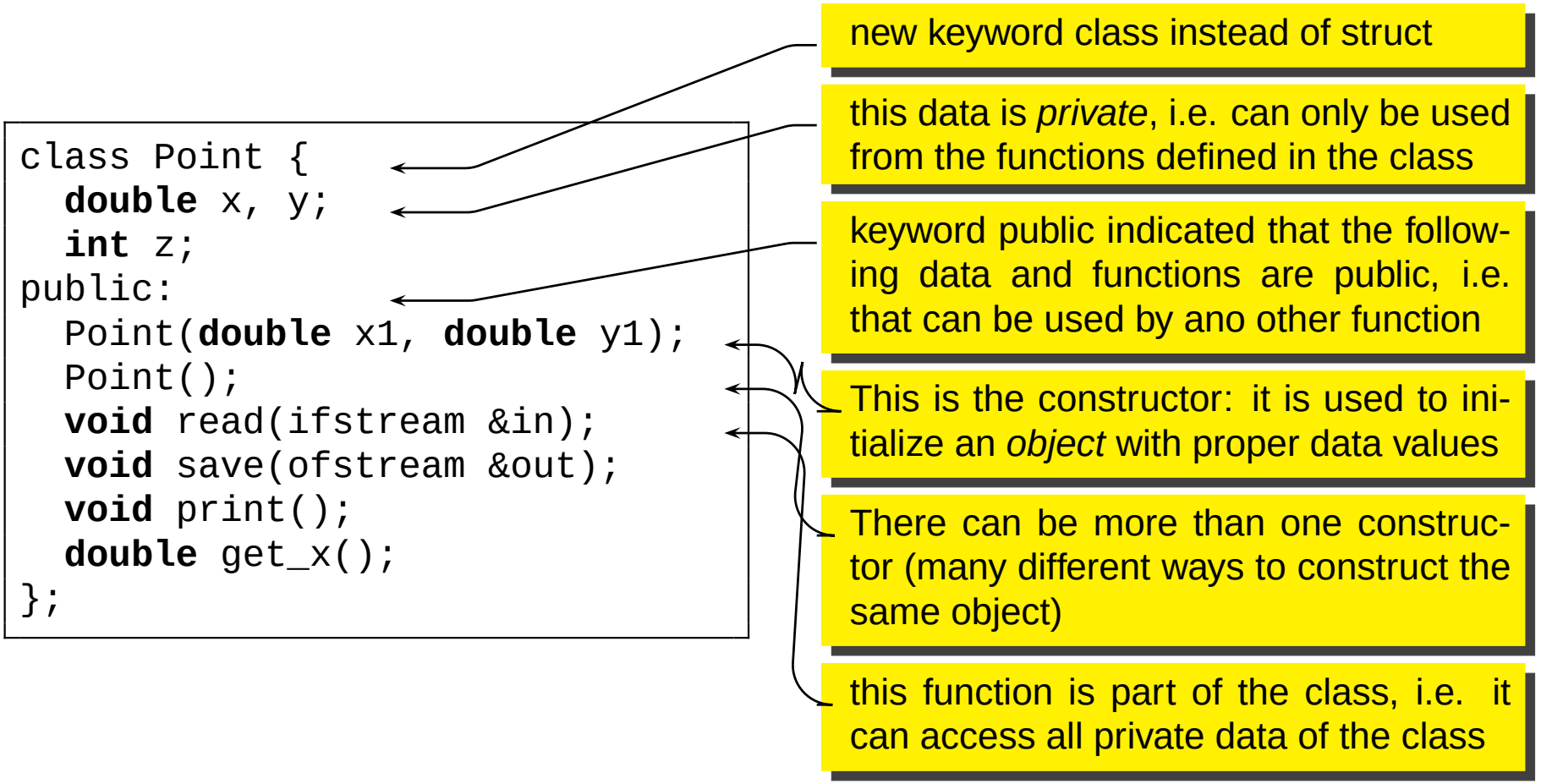
Example in C++

- C++ is the OO version of C
- It maintains a similar syntax, adding new keywords and constructs
- How the previous class can be expressed in C++?



Example in C++

- C++ is the OO version of C
- It maintains a similar syntax, adding new keywords and constructs
- How the previous class can be expressed in C++?



Usage

- This is an example of how the class Point can be used in a program.

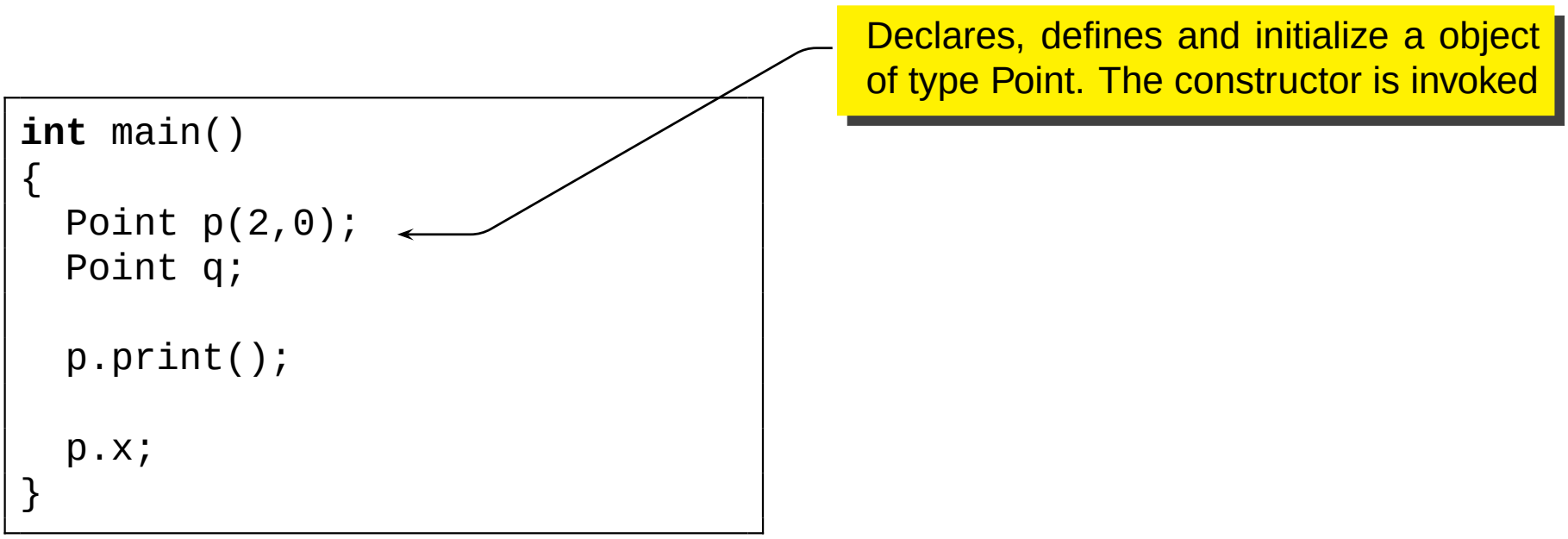
```
int main()
{
  Point p(2,0);
  Point q;

  p.print();

  p.x;
}
```

Usage

- This is an example of how the class Point can be used in a program.



Usage

- This is an example of how the class Point can be used in a program.

```
int main()
{
  Point p(2,0);
  Point q;

  p.print();

  p.x;
}
```

Declares, defines and initialize a object of type Point. The constructor is invoked

The default constructor is invoked

Usage

- This is an example of how the class Point can be used in a program.

```
int main()
{
  Point p(2,0);
  Point q;

  p.print();

  p.x;
}
```

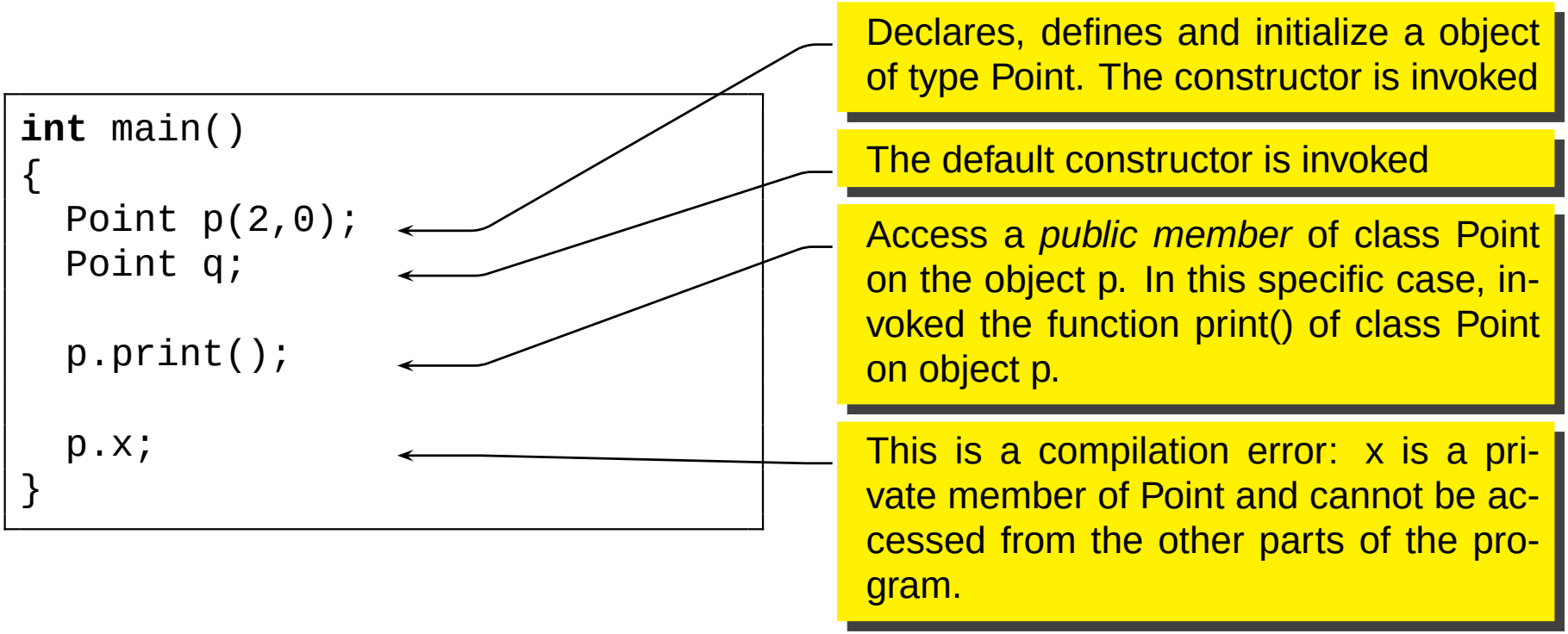
Declares, defines and initialize a object of type Point. The constructor is invoked

The default constructor is invoked

Access a *public member* of class Point on the object p. In this specific case, invoked the function print() of class Point on object p.

Usage

- This is an example of how the class Point can be used in a program.



Implementation

- Implementation usually goes into a separate class point.cpp

point.cpp

```
#include <iostream>
#include "point.hpp"

Point::Point() : x(0), y(0), z(0)
{}

Point::Point(double x1, double y1) :
    x(x1), y(y1), z(0)
{}

void Point::print()
{
   cout << "(" << x << "," << y << ")";
}
```

point.cpp

```
void Point::read(ifstream &if)
{
    in >> x >> y >> z;
}


void Point::save(ofstream &out)
{
    out << x << " " << y << " "
        << z << endl;
}

double Point::get_x()
{
    return x;
}
```

- Notice how we specify the functions, and how we access the member variables (i.e. variables defined inside the class).

Dynamic memory allocation

C language

```
int *p = (int *)malloc(sizeof(int));

int *a = (int *)malloc(10*sizeof(int));

...

free(p);
free(a);
```

C++ language

```
int *p = new int(0);

int *a = new [10] int;

...

delete p;
delete a;
```

- C++ uses the special keyword new, and a more automatic syntax (you can specify the type, and there is no need to speficy the size)

Is that all?

- C++ is a complex language, and we have just seen a few very basic concepts
- We have no time to present C++ in details. However, these very basic things should be necessary to start reasoning on data structures
- We will see more features as we go on.

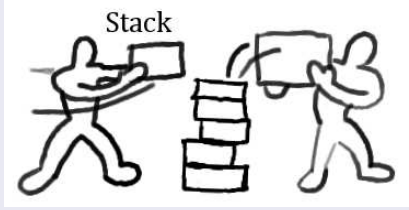
Outline

Stack

- A stack is a very simple data structure.
- A stack can hold a set of uniform data, like an array (for example, integers)
- Data is ordered according to the LIFO (Last-In-First-Out) strategy

Two main operations are defined on the data structure:

- Push: a new data in inserted in the stack
- Pop: data is extracted from the stack



Usually, we can also read the element at the top of the stack with a Top operation

Stack interface

- Let's start by defining a stack of integers of fixed size
  - Initially, we will allow only a maximum number of elements in the stack

stack.hpp

```
#ifndef __STACK_HPP__
#define __STACK_HPP__

class Stack {
    int array[10];
    int top;
public:
    Stack();
    void push(int elem);
    int pop();
    int query();
    void print();
};

#endif
```

Stack implementation

- Here is the implementation

stack.cpp

```
#include <iostream>
#include "stack.hpp"

using namespace std;

Stack::Stack() : top(0)
{
}

void Stack::push(int elem)
{
    if (top < 10) array[top++] = elem;
    else cerr << "Stack is full, push operation failed" << endl;
}

int Stack::pop()
{
    if (top > 0) return array[--top];
    else cerr << "Stack::pop() : is empty" << endl;
}
```

Stack implementation - 2

stack.cpp

```
int Stack::query()
{
    if (top > 0) return array[top-1];
    else cerr << "Stack::query() : is empty" << endl;
}

void Stack::print()
{
    int i;
    cout << "[";
    for (i=0; i<top; i++) cout << array[i] << ",";
    cout << "]" << endl;
}
```

Usage

- This is only an example of how to use the Stack class.

stackmain.cpp

```
#include "stack.hpp"
#include <iostream>

using namespace std;

int main()
{
    Stack s;
    int i;

    s.push(5);
    for (i=0; i<12; i++) s.push(2*i);

    s.print();

    for (i=0; i<5; i++) cout << s.pop() << endl;
}
```

Stack: unlimited size

- Let's remove the limitation on the size
  - We want a stack that enlarges itself dynamically

stackdyn.hpp

```
#ifndef __STACKDYN_HPP__
#define __STACKDYN_HPP__

class Stack {
    int *array;
    int cursize;
    int top;
public:
    Stack();
    ~Stack();
    void push(int elem);
    int pop();
    int query();
    void print();
};

#endif
```

Destructor

- The function `~Stack()` is called destructor
- It is automatically called when an object of type stack is destroyed
- As we will see in the implementation, in our case we need to deallocate the memory allocated by new with a corresponding delete.

Constructor and Destructor in stackdyn

stackdyn.cpp

```
#include <iostream>
#include "stackdyn.hpp"

using namespace std;

#define INC_SIZE 5

Stack::Stack() : top(0), cursize(INC_SIZE)
{
    array = new int[INC_SIZE];
}

Stack::~Stack()
{
    delete array;
}
```

Dynamic size stack implementation

stackdyn.cpp

```
void Stack::push(int elem)
{
    if (top >= cursize) {
        int i;
        int *temp = new int[cursize + INC_SIZE];
        for (i=0; i<top; i++) temp[i] = array[i];
        delete array;
        array = temp;
        cursize += INC_SIZE;
    }
    array[top++] = elem;
}

int Stack::pop()
{
    if (top > 0) return array[--top];
    else cerr << "Stack::pop() : is empty" << endl;
}
```

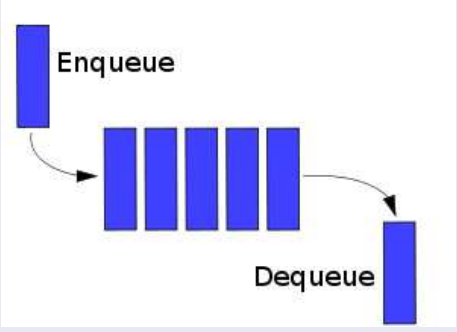
Outline

Queue

- Let us now implement a queue of integers
- The policy for inserting / extracting elements is FIFO (First-In-First-Out)

Two operations:

- enqueue inserts a new element in the queue
- dequeue extracts an element from the queue



Circular array

- Let's start again from a fixed size array

queue.hpp

```
#ifndef __QUEUE_HPP__
#define __QUEUE_HPP__

class Queue {
    int array[10];
    int head;
    int tail;
    int num;
public:
    Queue();
    void enqueue(int elem);
    int dequeue();
    void print();
};


#endif
```

Queue implementation

queue.cpp

```
#include "queue.hpp"
#include <iostream>

using namespace std;

Queue::Queue() : head(0), tail(0), num(0)
{
}

void Queue::enqueue(int elem)
{
    if (num < 10) {
        array[head] = elem;
        head = (head + 1) % 10;
        num++;
    }
    else cerr << "Queue::enqueue() : queue is full" << endl;
}
```

Queue implementation - 2

queue.cpp

```
int Queue::dequeue()
{
    int ret = 0;
    if (num > 0) {
        ret = array[tail];
        tail = (tail + 1) % 10;
        num--;
    }
    else cerr << "Queue::dequeue() : queue is empty" << endl;

    return ret;
}
```

Queue implementation - 3

queue.cpp

```
33  void Queue::print()
    {
35      int i;
        cout << "[";
37      for (i=0; i<num; i++) cout << array[(tail + i)%10] << ",";
        cout << "]" << endl;
39  }
```