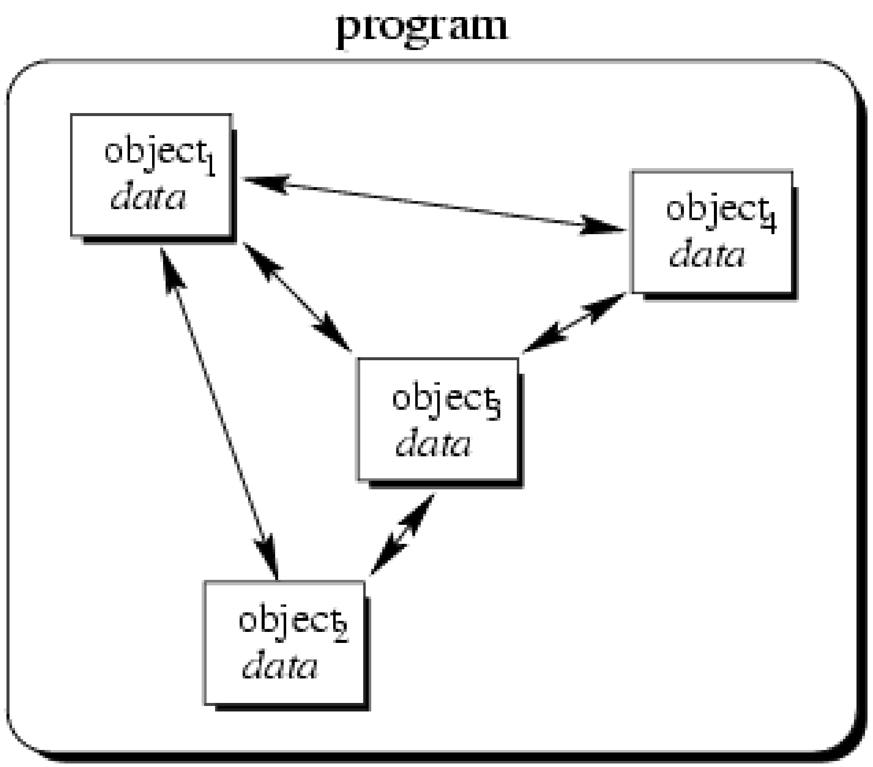
Introduction
C++ Review

CMPE226- Data Structures

Week 1 (Lecture-1)

Adapted from Textbook (D. S. MALIK), Lecture notes (C. TURHAN) and Lecture Notes (H. QU), 2020

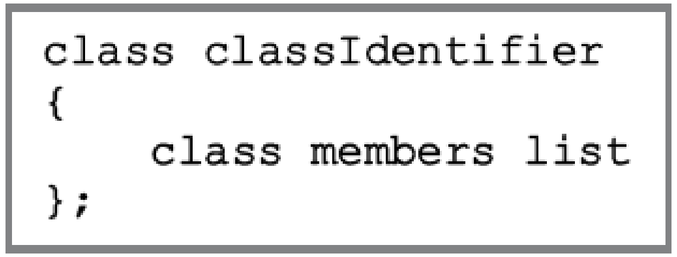
Object-Oriented Concept (C++)



Objects of the program interact by sending messages to each other

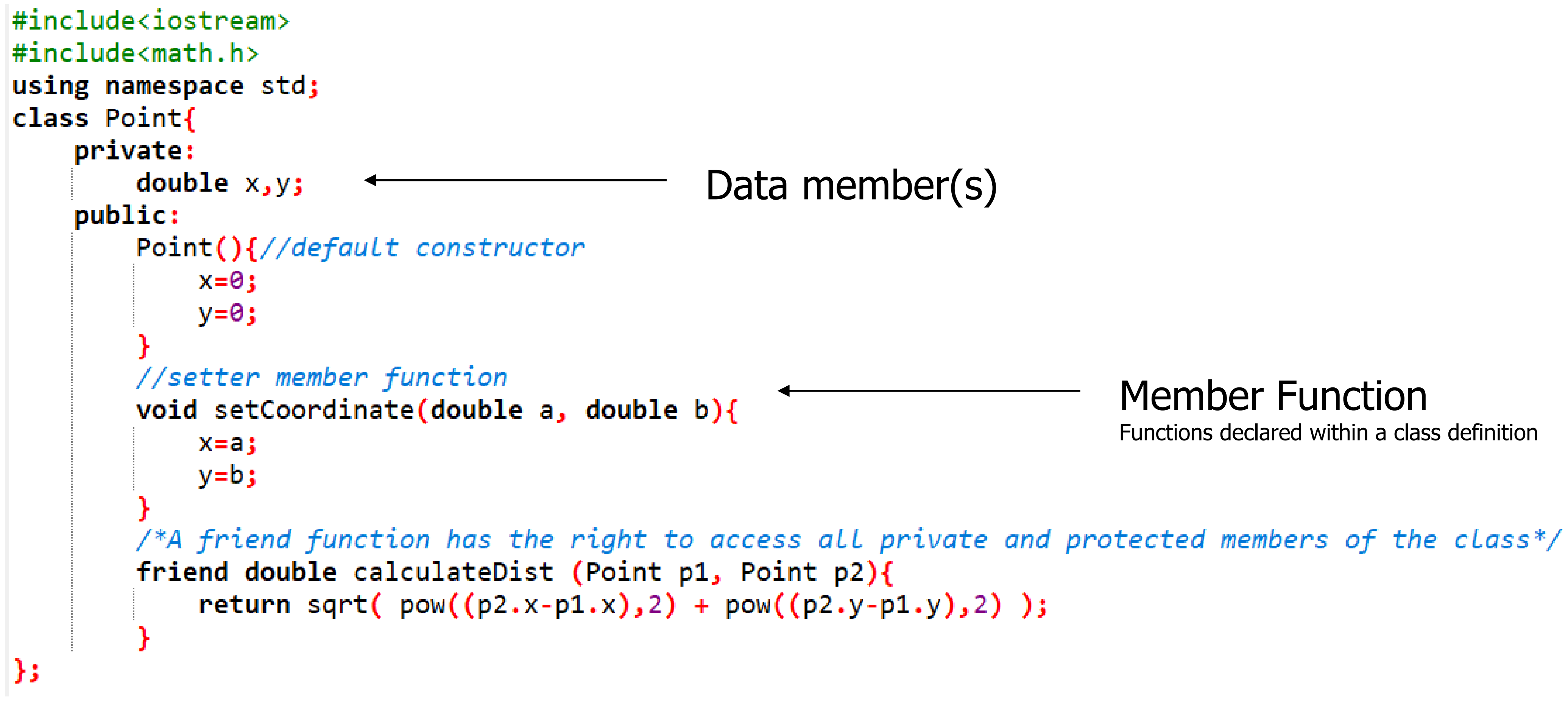
Classes

- OO-Design first step: identify components (objects)
  - Object is an instance of class.
- Class: collection of a fixed number of components
- A C++ class contains data members and methods (member functions)
  - Class members: class components
  - Class member categories
    - Private, public, protected
- Provides encapsulation and information hiding.



```
class classIdentifier
{
    class members list
};
```

Example: Class Definition



```
#include<iostream>
#include<math.h>
using namespace std;
class Point{
    private:
        double x,y;
    public:
        Point(){//default constructor
            x=0;
            y=0;
        }
        //setter member function
        void setCoordinate(double a, double b){
            x=a;
            y=b;
        }
        /*A friend function has the right to access all private and protected members of the class*/
        friend double calculateDist (Point p1, Point p2){
            return sqrt( pow((p2.x-p1.x),2) + pow((p2.y-p1.y),2) );
        }
};
```

Data member(s)

Member Function

Functions declared within a class definition

Information Hiding in C++

- Two labels: *public* and *private*
    - Determine visibility of class members
    - A member that is *public* may be accessed by any method in any class
    - A member that is *private* may only be accessed by methods in its class
- Information hiding
    - Data members are declared *private,* thus restricting access to internal details of the class
    - Methods intended for general use are made *public*

    - *Protected acts just like Private, except that it allow the inherited class to gain access.*

Pointers

- A *pointer* is a variable which contains addresses of other variables
- Declaring a pointer:

  int *p;

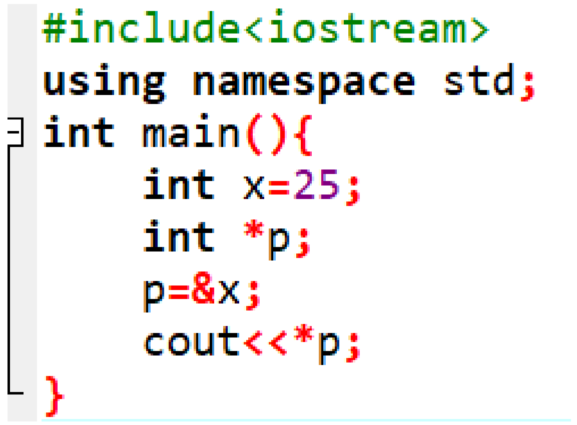
  char *c;

- Address of operator (&)

  int *x; *x is a variable that will contain the address of an integer variable*

  x= &num; address of the variable num is stored into x or x points to num.

- Dereferencing (Indirection) operator (*)
  - Exp. *p refers pointed by p, which is the value of x.



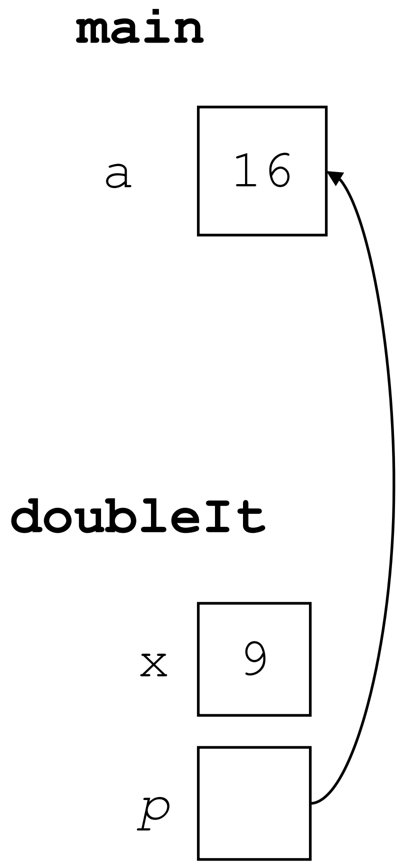
```
#include<iostream>
using namespace std;
int main(){
    int x=25;
    int *p;
    p=&x;
    cout<<*p;
}
```

A Pointer Example

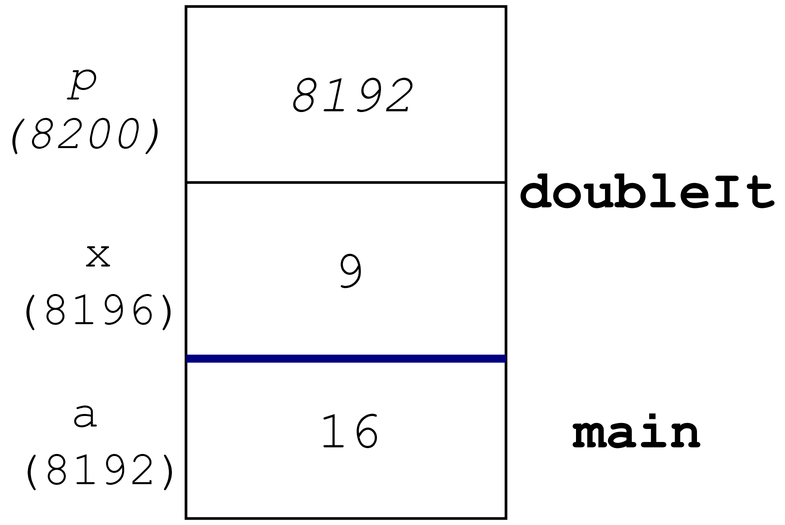
The code

```
void doubleIt(int x,  int * p)
{
  *p = 2 * x;
}
int main()
{
  int a = 16;
  doubleIt(9, &a);
  return 0;
}
```

Box diagram



Memory Layout



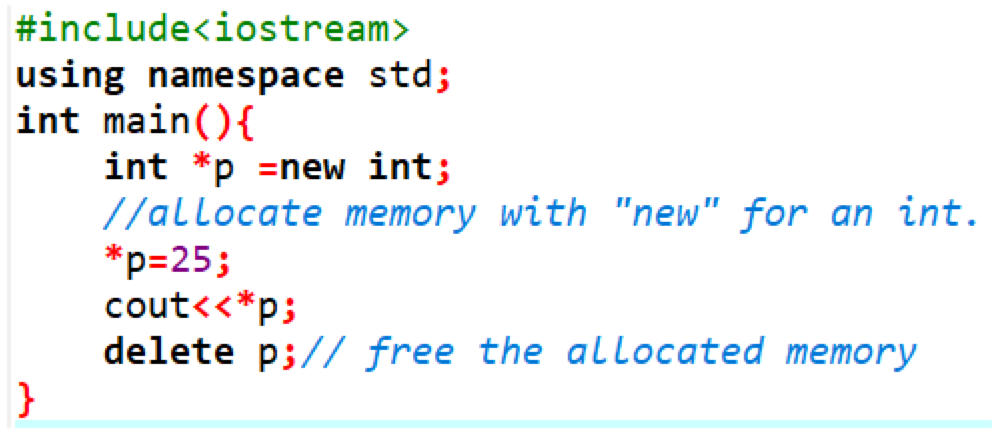
a gets 18

Delete a Pointer

- Dynamic object Creation:

```
Exp. IntCell *p; //defines a pointer to an object of class IntCell
     p = new IntCell;
```

- In C++ *new* returns a pointer to the newly created object.
- C++ does not have garbage collection
  - When an object that **is allocated by *new*** is no longer referenced, the *delete* operation must be applied to the object
    - Syntax: *delete p;*



```
#include<iostream>
using namespace std;
int main(){
    int *p =new int;
    //allocate memory with "new" for an int.
    *p=25;
    cout<<*p;
    delete p;// free the allocated memory
}
```

Templates in C++

- Problem: We wrote a function for adding two integer values. However, if we want other functions for adding two values of other types, do we need to write a function for each of them?
- Use a template:
    - the parameter is substituted *by the compiler*

Templates in C++

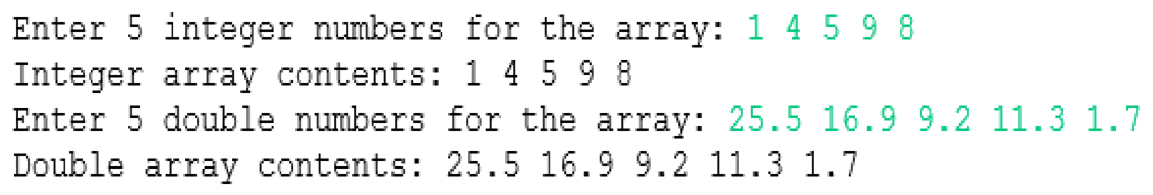
- Template is simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write same code for different data types.
- Function Template

  template<class Type>
  function declaration
- Class Template

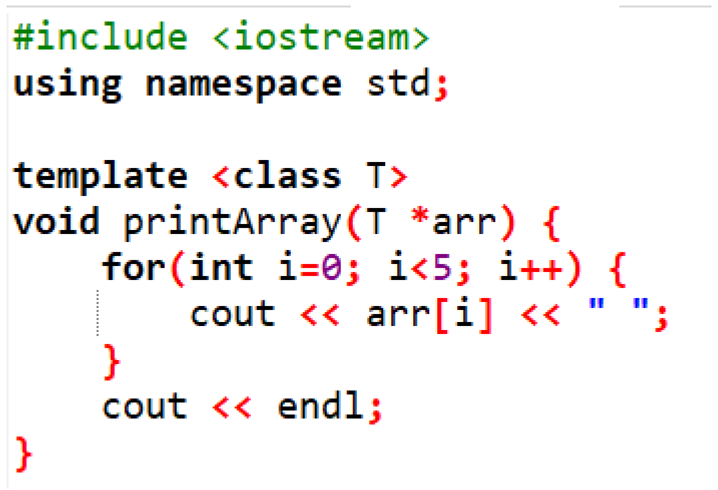
  template<class Type>
  class declaration

Example: Function Template

- Write a C++ program includes generic function named **printArray(array)** prints the value of an array that includes 5 elements of different data types (see Sample Run).

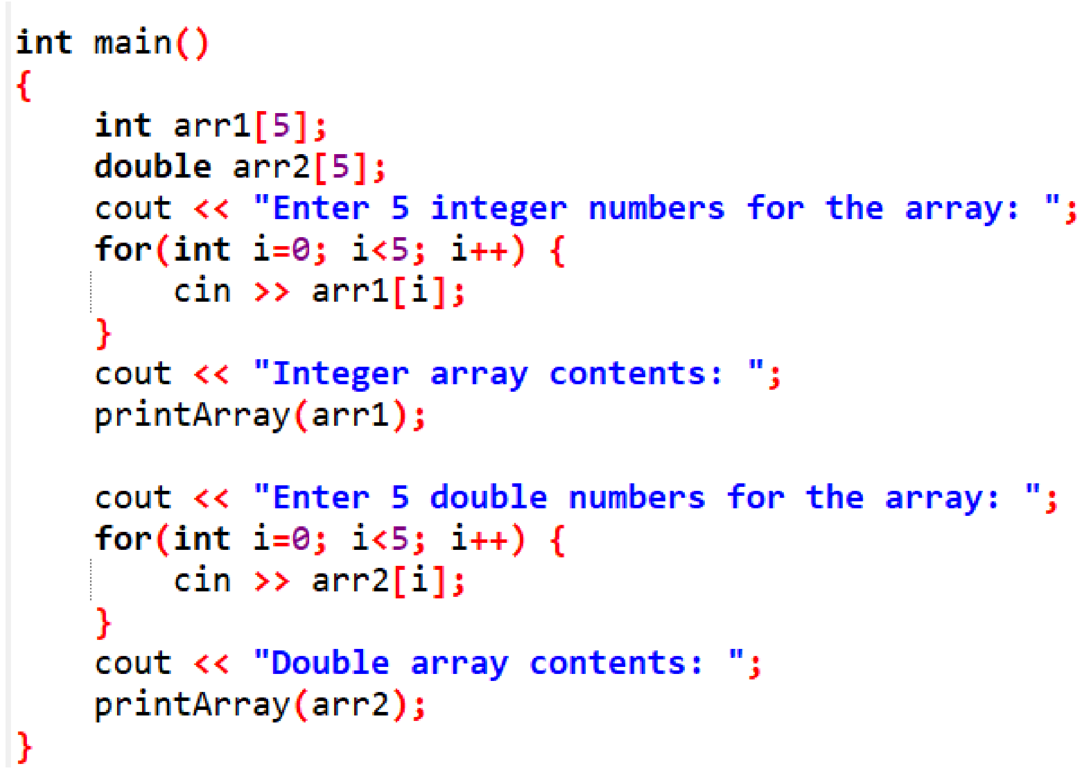


```
Enter 5 integer numbers for the array: 1 4 5 9 8
Integer array contents: 1 4 5 9 8
Enter 5 double numbers for the array: 25.5 16.9 9.2 11.3 1.7
Double array contents: 25.5 16.9 9.2 11.3 1.7
```



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

template <class T>
void printArray(T *arr) {
    for(int i=0; i<5; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
```



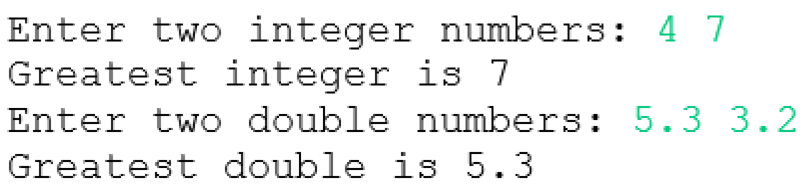
```cpp
int main()
{
    int arr1[5];
    double arr2[5];
    cout << "Enter 5 integer numbers for the array: ";
    for(int i=0; i<5; i++) {
        cin >> arr1[i];
    }
    cout << "Integer array contents: ";
    printArray(arr1);

    cout << "Enter 5 double numbers for the array: ";
    for(int i=0; i<5; i++) {
        cin >> arr2[i];
    }
    cout << "Double array contents: ";
    printArray(arr2);
}
```

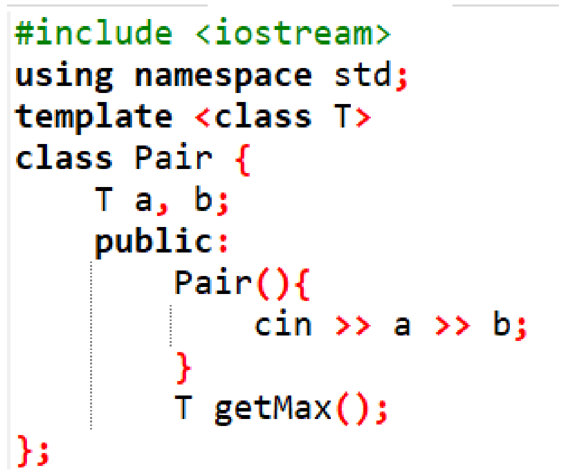
Example: Class Template

Write a program to explain class template by creating a template T for a class named **Pair** having two data members of type T which are inputted by a constructor and a member function **getMax()** return the greatest of two numbers to main.

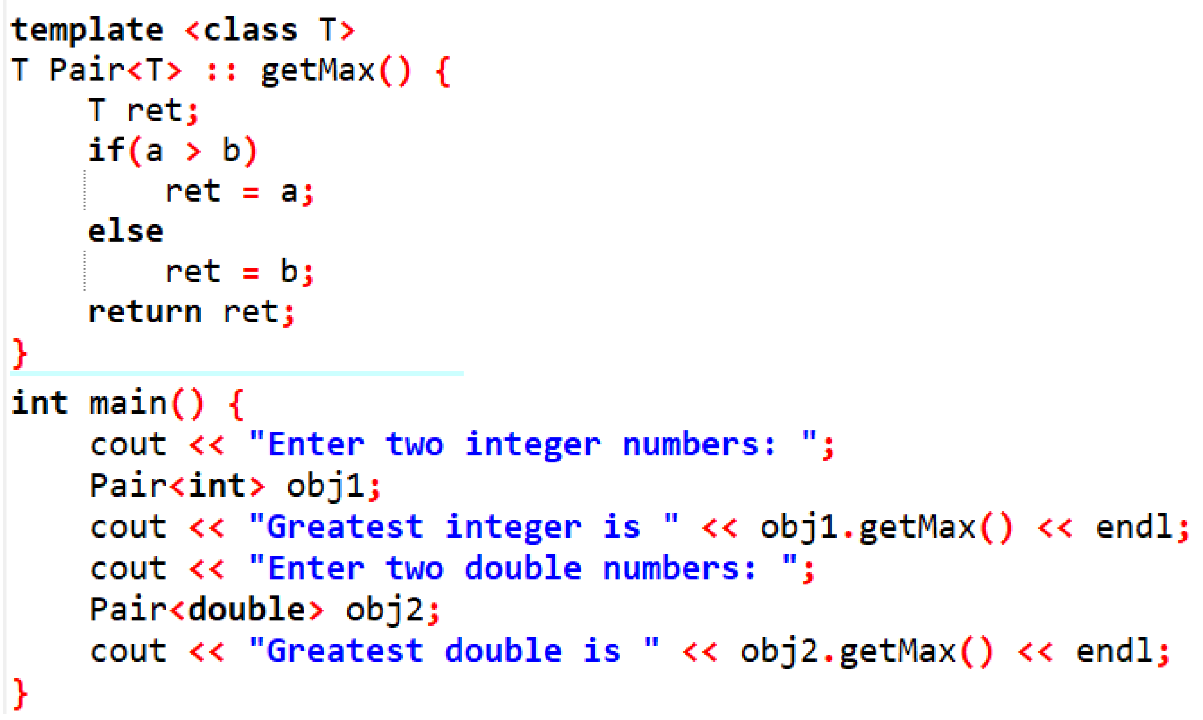
Note: the value of **T depends upon the data type specified during object creation in main.**



```
Enter two integer numbers: 4 7
Greatest integer is 7
Enter two double numbers: 5.3 3.2
Greatest double is 5.3
```



```cpp
#include <iostream>
using namespace std;
template <class T>
class Pair {
    T a, b;
    public:
        Pair(){
            cin >> a >> b;
        }
        T getMax();
};
```



```cpp
template <class T>
T Pair<T> :: getMax() {
    T ret;
    if(a > b)
        ret = a;
    else
        ret = b;
    return ret;
}
int main() {
    cout << "Enter two integer numbers: ";
    Pair<int> obj1;
    cout << "Greatest integer is " << obj1.getMax() << endl;
    cout << "Enter two double numbers: ";
    Pair<double> obj2;
    cout << "Greatest double is " << obj2.getMax() << endl;
}
```

C++ Operator Overloading

- C++ allows programmers to extend the definition of operators (relational, arithmetic, data input, output, etc.). Syntax:

return type operator *operatorsymbol (arguments)*

```
class Complex {
    ...
  public:
    ...
    Complex operator +(const Complex &op) {
      ...
    }
    ...
  };
```

In this case, we have made operator + a member of class Complex. An expression of the form

```
c = a + b;
```

is translated into a method call

```
c = a.operator +(a, b);
```

C++ Operator Overloading

- Exp. Adding complex numbers

```
#include<iostream>
using namespace std;
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0) {real = r; imag = i;}
    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print() { cout << real << "+"<< imag << "i" ; }
};
int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

C++ Operator Overloading

Overloading Input and Output Stream Operators “<<“ and “>>”

- C++ is able to input and output the built-in data types using:
  - stream extraction operator >>
  - stream insertion operator <<
- The stream insertion and stream extraction operators also **can be overloaded to perform input and output for user-defined <u>types like an object.</u>**
- Exp. Point p(2,3);

```
cout<<p; //compile time error if << operator is not defined in Point class
```

Should be defined as a friend function in class to use private data members.

Example: C++ Operator Overloading

Point.h

```
#ifndef POINT_H
#define POINT_H
// Point class declaration
#include<iostream>
using namespace std;
class Point{
        private:
                int x,y;
        public:
                Point();
                Point(int x=0, int y=0);
                void setPoint(int, int);
                friend ostream& operator<<(ostream& out, Point& p);
                friend istream& operator>>(istream& in, Point& p);
};
#endif /* POINT_H_ */
```

Example: C++ Operator Overloading

Point.cpp

```
#include <iostream>
#include "Point.h"
using namespace std;
ostream& operator<<(ostream& out, Point& p){
        out<<"("<<p.x<<","<<p.y<<")"; //output members of the object
        return out; //return the ostream object
}
istream& operator>>(istream& in, Point& p){
        in>>p.x>>p.y; //read data into object
        return in; //return the istream object
}
Point::Point(int x, int y){
        setPoint(x, y); }
void Point::setPoint(int x, int y){
        this->x=x;
        this->y=y; }
```

Example: C++ Operator Overloading

main.cpp

```
#include <iostream>
#include "Point.h"

int main() {
        Point p(2,3);
        cout<<p<<endl;
        cout<<"Enter a point";
        cin>>p; // invokes operator>> function by calling operator>> (in, p)
        cout<<p<<endl; // invokes operator<< function by calling operator<< (out, p)
        return 0;
}
```

Data Structures in C++

CMPE226- Data Structures

Adapted from Textbook (D. S. MALIK), Lecture notes (C. TURHAN) and Lecture Notes (H. QU), 2020

Introduction

Programs are comprised of data and algorithms.

Data structures: Organized collection of data

Why to study Data Structures?

- The more sophisticated the data structure, the simpler the algorithm.
- Simple algorithms are less expensive to develop.
- There is less code to read and comprehend, less likely to introduce errors.
- It's usually much easier to repair defects, make modifications, or add enhancements.

http://users.csc.calpoly.edu/~jdalbey/103/Lectures/Whystudydatastructures.html

Introduction

- C++ Data Structures: Arrays, Linked Lists, Stacks, Queues, Trees
- A certain data structure might be better suited for a problem in terms of efficiency.

  Examples:

  - *Queue:* An example of FIFO data structure (Data inserted first, will leave the queue first.)

    Exp: printer – first come first served
  - *Stack:* An example of LIFO data structure (Data inserted last will leave the stack first)

    Exp: Web History

Abstract Data Type (ADT)

- Data type
  - a collection of values + a set of operations
  - Example: integer
    - set of whole numbers: ……, -2, -1, 0, 1, 2, ……
    - operations: +, -, x, /
- Can this be generalized?
  - (e.g. procedures generalize the notion of an operator)
  - Yes!

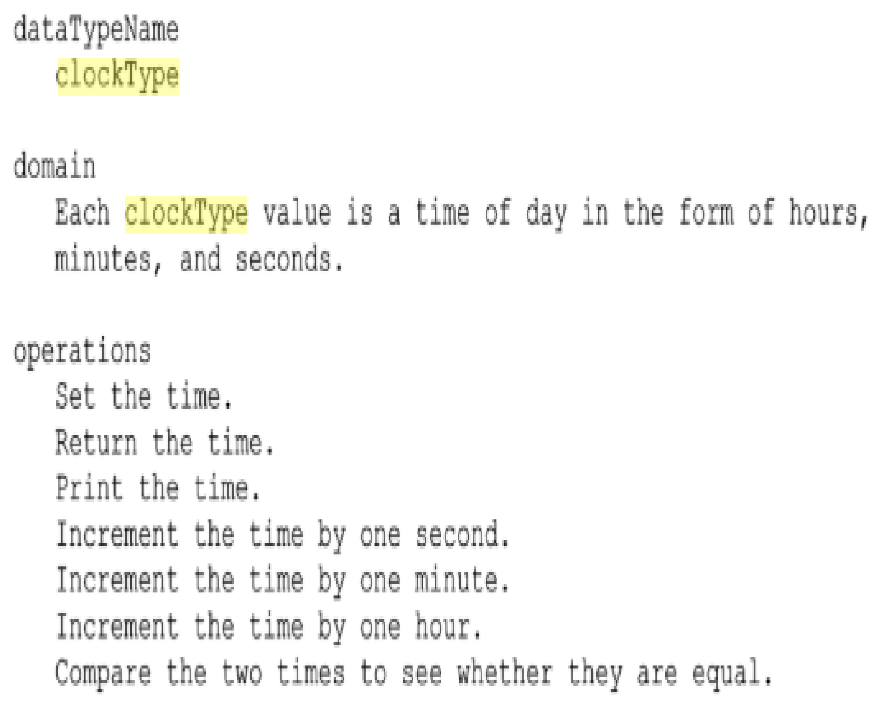
➢ **Abstract** data type
  - A datatype that specifies the logical properties without the implementation details

Abstract Data Type (ADT)

- An ADT defines a data type and a set of operations on that type.
  - Details of implementation is hidden (encapsulation).
  - Only know what operations are allowed on that datatype (abstraction).
- ADT has three things associated with it:
- The name of ADT: type name
- The set of values belonging to ADT: domain
- The set of operations on the data.

ADT Example:

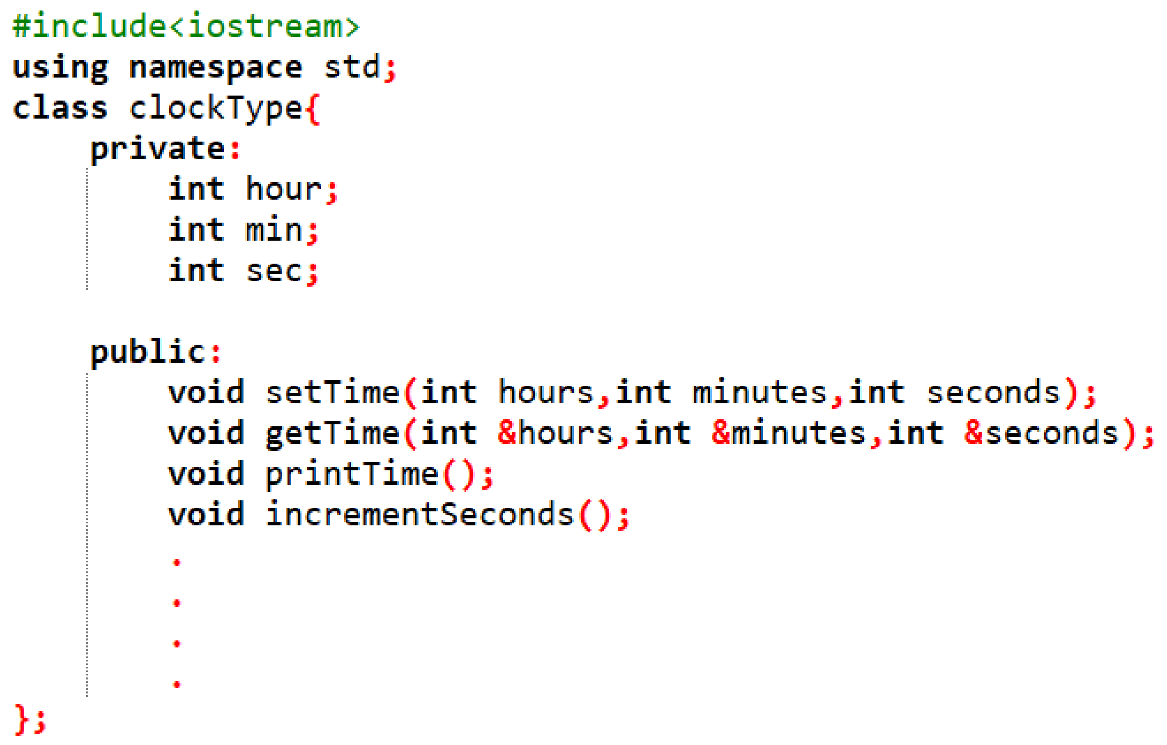
C++ Code



```
dataTypeName
   clockType

domain
   Each clockType value is a time of day in the form of hours,
   minutes, and seconds.

operations
   Set the time.
   Return the time.
   Print the time.
   Increment the time by one second.
   Increment the time by one minute.
   Increment the time by one hour.
   Compare the two times to see whether they are equal.
```



```cpp
#include<iostream>
using namespace std;
class clockType{
    private:
        int hour;
        int min;
        int sec;

    public:
        void setTime(int hours,int minutes,int seconds);
        void getTime(int &hours,int &minutes,int &seconds);
        void printTime();
        void incrementSeconds();
        .
        .
        .
        .
};
```

Malik, D. S. (2009). *Data structures using C++*. Cengage Learning.

References

- CMPE226- Lecture Notes by Cigdem Turhan
- Data Structures Using C++, D.S. Malik, Thomson Course Technology, 2nd Edition.
- Lecture Slides by Huamin Qu, The Hong Kong University of Science and Technology (2005)