EE 441 – CH1

CLASSES

Instructor: UĞUR HALICI

ABSTRACT DATA TYPES

ADT: Implementation independent data description that specifies the contents, structure and legal operations on the data.

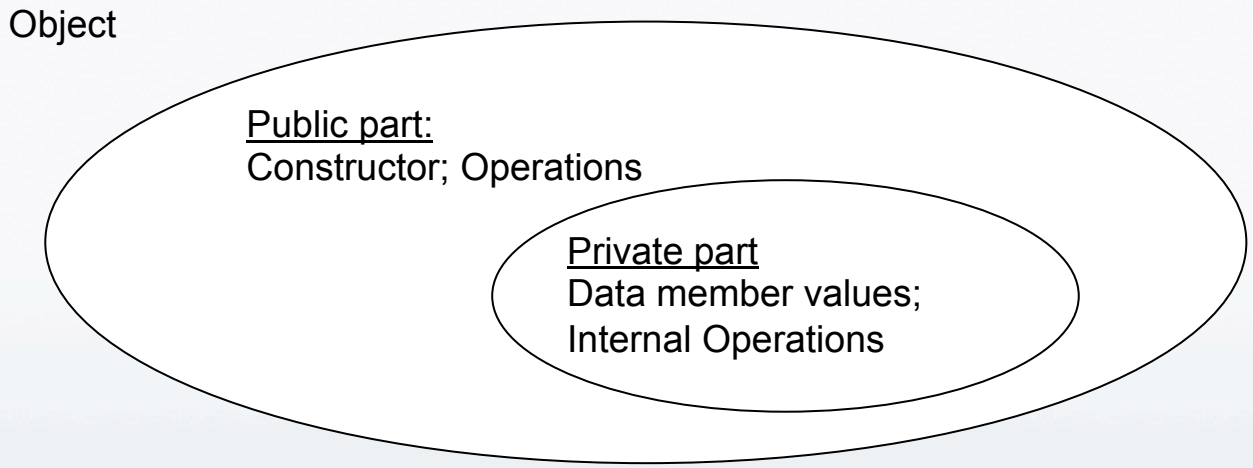
An ADT has a
Name:
Description of the data structure:
Operations:
- Construction operations:
  - Initial values;
  - Initialization processes;
- Other Operations:

For each operation:
- Name of the operation;
- Input: external data that comes from the user of this data (client)
- Preconditions: Necesssary state of the system before executing this operation;
- Process: Actions performed by the data
- Output: Data that is output to the client
- Post Conditions: state of the system after executing this operation

Example:

ADT circle
Data
    r≥0 (radius);
Operations
    Constructor:
        Input:radius of circle;
        Preconditions: none;
        Process: Assign initial value of r;
        Output: none;
        Postcondition: none;
    Area:
        Input: none;
        Preconditions: none;
        Process: A ← π*r*r
        Output: A
        Postcondition: none
    Circumference:
        Input: None;
        Preconditions: none;
        Process: C ← 2*π*r
        Output: C
        Postconditions: none;
End ADT circle;

Representation of ADT's in C++



Private: Data and internal operations necessary to implement the class

Public: Operations available to clients (who do not need to know anything about the private parts)

Example:

Class Circle

Private: radius (if it is not to be used anywhere outside the class)

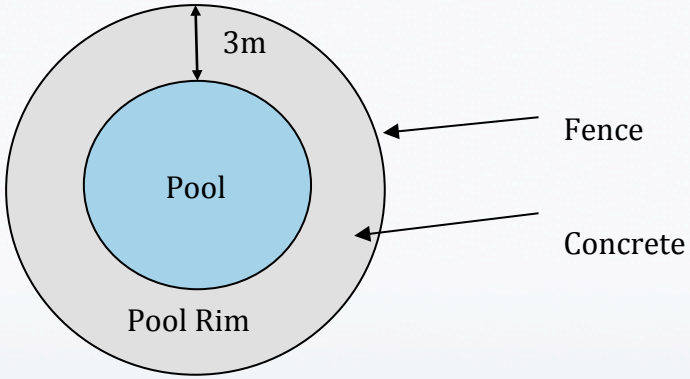
Public: constructor; area; circumference

Clients can only access the public part, i.e., the three methods in this case

Private data members and operations can be acessed only by the methods in the class

An object is an instance of a class type.

Example: Program to compute fence cost and concrete cost for swimming pool




```
# include <iostream.h>  /* necessary for input,outputs */
const float PI=3.141592;
const float FencePrice=5; /*TL per meter */
const float ConcretePrice=8 /*TL per squaremeter */
/ * Class decleration */
class Circle
{ private:
      float radius;
  public:
      Circle(float r); /* constructor function */
      float Circumference(void) const; // this is  a constant function since the object
                                       // is not changed when it is called
      float Area(void) const;
}
```

Example: (continues)

```
/ * Class decleration */
class Circle
{ private:
        float radius;
  public:
     Circle(float r);
     float Circumference(void) const;
     float Area(void) const;
}
```

```
/* Class imlementation */
/* constructer */
Circle::Circle(float r)
{
     radius=r
}
/* return circumference */
float Circle::Circumference (void) const
{
     return  2*PI*radius
}
/* return area */
float Circle::Area(void) const
{
     return PI*radius*radius
}
```

Example: (continues)

```
void main()  /* Now let's use this data
     structure*/
{ float radius;  /* pool radius */
     /* this has the same name with the
     encapsulated private data, but
     does not cause any problem */
  float FenceCost, ConcreteCost;
 /* arrange output format to 2 decimal
 place */
        cout.setf(ios::fixed);
        cout.setf(ios::showpoint);
        cout.precision (2);
 /* get radius from user */
        cout << "Enter pool radius>";
        cin >> radius;
```

```
/* create two different instances off class
     Circle, i.e. Circle objects */
     Circle Pool(radius);
     Circle PoolRim(radius+3);
/* Note: we did not have to say anything
     about centers*/

/* compute fence cost and output
     FenceCost=PoolRim.Circumference()*Fenc
     ePrice;
     Cout<<"Fence cost is:"
     <<FenceCost<<"TL"<<endl;

/* ..and compute concretecost and output it */
     ConcreteCost=(PoolRim.Area()-
     Pool.Area())*ConcretePrice;
     Cout<<"Concrete cost is:"
     <<ConcreteCost<<"TL"<<endl
}
/* sample run:
Enter pool radius> 5
Fence cost is: 50.27  TL
ConcreteCost is: 122.52 TL
*/
```

Example:

```
class Rectangle
{
private:
float length, with;
// Note: length and width are data members of ADT ; by placing data members
// in private part, we ensure that only defined methods may change them.

public:
Rectangle (float l=0, float w=0); // constructor
// Note: constructor has the same name as the class

// methods to retrieve and modify private data GetLength, PutLength,
//GetWidth, Putwidth
float GetLength(void) const;
// Note:const indicates that the method is  constant, therefore no class data
// item may be modified, i.e. GetLength does not change the state of
// the Rectangle object
void PutLength (float l);
float GetWidth(void) const;
void PutWidth(float w);

float Perimeter(void) const;
float Area(void) const;
};
```

OBJECT DECLARATION:

Object declaration creates an instance of a class:

<classname> <object(parameters)>;

Rectangle room(12,10);
Rectangle t; // omitting parameters means using default values, i.e. l=0, w=0
Rectangle square(10,10), yard=square, S;
// yard is initially identical to square, S is created with default length, with=0.

Public members of an object can be accessed :

x=room.Area(); // assigns Area=12*10=120 to x
t.PutLength(20); //assigns 20 as length of Rectangle t

cout<<t.GetWidth(); // outputs current value of width of t, which is 0
cin>>x;
t.PutLength(x); // inputs x and assigns it to length of t.

Constructor and other methods can be defined inline or outside the class body. If a method is not defined inside the class declaration (i.e.if not defined inline), it must be defined outside

<ReturnValueType> <ClassName>::<FuctionName(parameters)>

```
float Rectangle::GetLength(void) const
// using :: makes function belong to class, so it can access private members
{ return Length; // access private member length
}
```

the same result may be achieved as:

```
class Rectangle
{
private:
     float length, width;
public:
     ….
float GetLength(void) const  // code is given inline
{
    return Length;
}
….
}
```

ALTERNATIVE CONSTRUCTORS

More than one constructer can be defined for a class

Example:

```
#include  <string.h>
class Date
{
private:
    int month, day, year;
public:
    Date (int m=1, int d=1, int y=2000);
    Date(char *dstr);
// Note: two different constructors are defined;
// The compiler will select the appropriate constructor according
// to the call parameters during object creation
    void PrintDate(void); // the only method other than the constructor
};
```

```
// Now the methods will be defined externally:
Date::Date(int m=1, int d=1, int y=2000): month(m), day(d), year(y)
{ }
// Alternative constructor
Date::Date(char *dstr)
{       ……
       // Here some operations are needed to read string in the form
        // “dd/mm/yyyy” from input stream, and then convert it to integer
        // month, day, year
}

// the print method
void Date::PrintDate(void)
{    //static table of months
static char *Months[ ]={"",”January", …,"December"};
cout<<Months[month]<<" "<<day<<", "<<year;
}
```

Assuming the class date is contained in the file "date.h"

```
#include <iostream.h>
#include "date.h"
void main(void)
{
        Date day1(10,29,1923);
        Date day2;
        Date day3("04/10/2007");
        cout<<"The Turkish Republic was founded on";
        day1.printDate(); cout<<endl;
        cout<<"The first day in the 21st century was";
        day2.Printdate();cout<<endl;
        cout<<"Today is";
        day3.PrintDate();cout<<endl;
}
```

TEMPLATES

We may want to use the same class or function definition for different types of items,

i.e. we may want to create different objects within the same class, but whose datatypes are different

or we may want to define a general function that works on different datatypes.

It would be nice if we could define the data type with the object or with the function call.

Example

```
Stack <float> A;
Stack<char> B;
SeqList<int> C;
SeqList<char> D; etc.
```

C++ allows this through the usage of templates

template <class T1, Class T2, …ClassTn>

indicates that T1,T2, ..Tn are classes that will be used with a specific class upon creation of an object

Example:

```
template <class T>
// Note here that any operation in the template class or function must be
//defined for any possible data types in the template
int SeqSearch(T list[ ], int n, T key)
// T is a type that will be specified when SeqSearch is called
{
     for (int i=0; i<n; i++)
        if list[i]==key)
                    return i;
     return -1;
}
```

Now call SeqSearch with different datatypes:

```
int A[10], Aindex, Mindex
float M[10], fkey=4.5;
Aindex=SeqSearch(A,10,25); //search for int 25 in A
Mindex=SeqSearch(M,100,fkey); //search for float 4.5 in M
```

C++ OPERATION ON ARRAYS

Decleration examples:

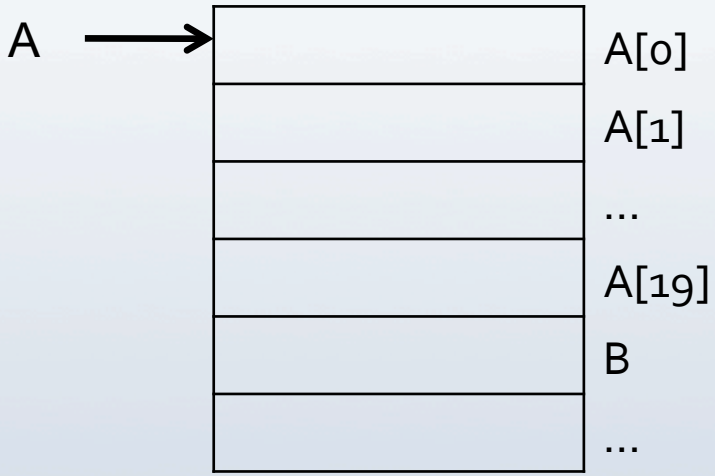
```
Const int ArraySize=50;
    float        A[ArraySize];
    long         X[ArraySize+10];
```

assignment examples:

```
A[i]=z;
t=X[i]
X[i]=X[i+1]=t
/* this is equivalent to X[i+1]=t;  X[i]=X[i+1] , i.e. right to left */
```

Example:

```
int V=20;
int A[20]; /*  index range is 0-19  */
// A is a pointer  contains the starting address of the array,
// same as the address of A[0]
int B;
```




```
A[V]=0; /* index is out of range, but most C++ compilers don't check this */
```

Effect is B=0, since the first location after A is reserved for B by above decleration

TWO DIMENSIONAL ARRAYS

Example:

int T[3] [4]={{20,5,-3,0},{-85,35,-1,2},{15,3,-2,-6}};

T:

20	5	-3	0	T[0]
-85	35	-1	2	T[1]
15	3	-2	-6	T[2]

a=T[2] [3]; T[0][4]=a;

Example:

int T[ ][5] // list of 5 element arrays

ARRAYS OF OBJECTS

Rectangle room[100]; // constructor is called for room[0] .. room[99];

This declaration creates an array of 100 objects. Each have different members, all initialized in this case to the default values.

Note: For declaring large array objects, a constructor with default values or with no parameters is preferred.

Rectangle room[3]={rectangle(10,15), Rectangle(5,8),Rectangle(2,30)}

may be practical, but

Rectangle room[100];

simply initializes all length and with values to the default value of 0.

Notes:

```
void f(char X[ ])
{… }
```

is equivalent to

```
void f(char *X)
{… }
```

when calling

```
void main ()
{
    Char A[100]
    ….
    f(A)  // here A is pointer to array
    ….
}
```

Character Sequences:

As you may already know, the C++ Standard Library implements a **string** class, which is useful to handle and manipulate strings of characters. However, because strings are in fact sequences of characters, we can represent them also as plain arrays of char elements.

For example, declare Jenny as an array that can store up to 20 elements of type char.

char Jenny [20];

It can be represented as:

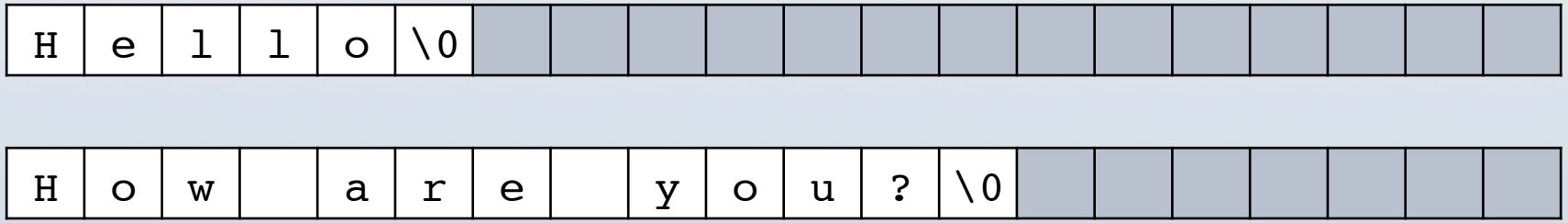
Jenny:

In the array Jenny, in theory, we can store sequences of characters up to 20 characters long. But we can also store shorter sequences. For example, Jenny could store at some point in a program either the sequence "Hello" or the sequence ”How are you?", since both are shorter than 20 characters.

Therefore, since the array of characters can store shorter sequences than its total length, a special character is used to signal the end of the valid sequence: the *null character*, whose literal constant can be written as '\0' (backslash, zero).

Our array of 20 elements of type char, called jenny, can be represented storing the characters sequences "Hello" and ”How are you?" as:

Jennny:



Notice how after the valid content a null character ('\0') has been included in order to indicate the end of the sequence. The panels in gray color represent char elements with undetermined values.

Initialization of null-terminated character sequences

Because arrays of characters are ordinary arrays they follow all their same rules. For example, if we want to initialize an array of characters with some predetermined sequence of characters we can do it just like any other array:

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

In this case we would have declared an array of 6 elements of type char initialized with the characters that form the word "Hello" plus a null character '\0' at the end.

But arrays of char elements have an additional method to initialize their values: using string literals.

Double quoted strings (") are literal constants whose type is in fact a null-terminated array of characters. So string literals enclosed between double quotes always have a null character ('\0') automatically appended at the end.

Therefore we can initialize the array of char elements called myword with a null-terminated sequence of characters by either one of these two methods:

```
char myword [ ] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char myword [ ] = "Hello";
```

In both cases the array of characters myword is declared with a size of 6 elements of type char: the 5 characters that compose the word "Hello" plus a final null character ('\0') which specifies the end of the sequence and that, in the second case, when using double quotes (") it is appended automatically.

Notice that we are talking about initializing an array of characters in the moment it is being declared, and not about assigning values to them once they have already been declared. In fact because this type of null-terminated arrays of characters are regular arrays we have the same restrictions that we have with any other array, so we are not able to copy blocks of data with an assignment operation.

Assuming mystext is a char[ ] variable declared previously. Assignment operations within a source code like:

```
mystext = "Hello";
```

```
mystext[ ] = "Hello";
```

would **not** be valid, like neither would be:

```
mystext = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

The reason for this may become more comprehensible once you know a bit more about pointers, since then it will be clarified that an array is in fact a constant pointer pointing to a block of memory.

- **Using null-terminated sequences of characters**

Null-terminated sequences of characters are the natural way of treating strings in C++, so they can be used as such in many procedures. In fact, regular string literals have this type (char[ ]) and can also be used in most cases.

For example, cin and cout support null-terminated sequences as valid containers for sequences of characters, so they can be used directly to extract strings of characters from cin or to insert them into cout. For example:

```
// null-terminated sequences of characters
#include <iostream>;
voidmain ()
{ char question[] = "Please, enter your first name: "; char
greeting[] = "Hello, ";
char yourname [80];
 cout << question; cin >> yourname;
cout << greeting << yourname << "!";}
```

```
a sample run:
Please, enter your first name: John
Hello, John!
```

As you can see, we have declared three arrays of char elements. The first two were initialized with string literal constants, while the third one was left uninitialized. In any case, we have to specify the size of the array: in the first two (question and greeting) the size was implicitly defined by the length of the literal constant they were initialized to. While for yourname we have explicitly specified that it has a size of 80 chars.

Finally, sequences of characters stored in char arrays can easily be converted into string objects just by using the assignment operator:

```
string mystring;

char myntcs[ ]="some text";

mystring = myntcs;
```