CHAPTER 7

Constructors and Other Tools

LEARNING OBJECTIVES

- Constructors
  - Definitions
  - Calling
- More Tools
  - const parameter modifier
  - Inline functions
  - Static member data
- Vectors
  - Introduction to vector class

CONSTRUCTORS

- Initialization of objects
  - Initialize some or all member variables
  - Other actions possible as well
- A special kind of member function
  - Automatically called when object declared
- Very useful tool
  - Key principle of OOP

CONSTRUCTOR DEFINITIONS

- Constructors defined like any member function
  - Except:

1. Must have **same name** as class

2. **Cannot** **return** a value; **not** even **void**!

CONSTRUCTOR DEFINITION EXAMPLE

- Class definition with constructor:
  - class DayOfYear
    {
    public:
        DayOfYear(int monthValue, int dayValue);
                //Constructor initializes month and day
        void input();
        void output();
        …
    private:
        int month;
        int day;
    }

CONSTRUCTOR NOTES

- Notice name of constructor: DayOfYear
  - **Same name** as class itself!
- Constructor declaration has **no** return-type
  - Not even void!
- Constructor in **public** section
  - It's called when objects are declared
  - If private, could never declare objects!

CALLING CONSTRUCTORS

- Declare objects:

  DayOfYear date1(7, 4), date2(5, 5);

- Objects are created here
  - Constructor is called
  - Values in parens passed as arguments to constructor
  - Member variables month, day initialized:
    date1.month → 7 date2.month → 5
    date1.dat → 4 date2.day → 5

```
DayOfYear::DayOfYear(int monthValue, int dayValue)
{
      month = monthValue;
      day = dayValue;
}
```

CONSTRUCTOR EQUIVALENCY

- Consider:
  - DayOfYear date1, date2

    date1.DayOfYear(7, 4); // ILLEGAL!

    date2.DayOfYear(5, 5); // ILLEGAL!
- Seemingly OK…
  - **CANNOT** call constructors like other member functions!

CONSTRUCTOR CODE

- Constructor definition is like all other member functions:

```
DayOfYear::DayOfYear(int monthValue, int dayValue)
{
        month = monthValue;
        day = dayValue;
}
```

- Note **same name** around ::
  - Clearly identifies a constructor
- Note **no return** type
  - Just as in class definition

ALTERNATIVE DEFINITION

- Previous definition equivalent to:

```
DayOfYear::DayOfYear( int monthValue,
                      int dayValue)
                    : month(monthValue), day(dayValue)
{…}
```

- Third line called "**Initialization Section**"
- Body left empty
- **Preferable** definition version

CONSTRUCTOR ADDITIONAL PURPOSE

- Not just initialize data
- Body doesn't have to be empty
  - In initializer version
- Validate the data!
  - Ensure only appropriate data is assigned to class private member variables
  - Powerful OOP principle

OVERLOADED CONSTRUCTORS

- Can overload constructors just like other functions
- Recall: a signature consists of:
  - Name of function
  - Parameter list
- Provide constructors for all possible argument-lists
  - Particularly "how many"

```
1)  class DayOfYear
2)  {
3)  public:
4)      DayOfYear(int monthValue, int dayValue);
5)      //Initializes the month and day to arguments.
6)      DayOfYear(int monthValue);
7)      //Initializes the date to the first of the given month.
8)      DayOfYear( );
9)      //Initializes the date to January 1.
10)     void input( );
11)     void output( );
12)     int getMonthNumber( );
13)     //Returns 1 for January, 2 for February, etc.
14)     int getDay( );
15) private:
16)     int month;
17)     int day;
18)     void testDate( );
19) };
```

```
1)  DayOfYear::DayOfYear(int monthValue, int dayValue)
2)                     :month(monthValue), day(dayValue)
3)  {
4)      testDate( );
5)  }
```

```
6)  DayOfYear::DayOfYear(int monthValue) :
                                  month(monthValue), day(1)
7)  {
8)      testDate( );
9)  }
```

```
10) DayOfYear::DayOfYear( ) :month(1), day(1)
11) { /*Body intentionally empty.*/ }
```

```
1)    void DayOfYear::testDate( )
2)    {
3)       if ((month < 1) || (month > 12))
4)       {
5)          cout << "Illegal month value!\n";
6)          exit(1);
7)       }
8)       if ((day < 1) || (day > 31))
9)       {
10)         cout << "Illegal day value!\n";
11)         exit(1);
12)      }
13)   }

14)   int DayOfYear::getMonthNumber( )
15)   {
16)      return month;
17)   }

18)   int DayOfYear::getDay( )
19)   {
20)      return day;
21)   }
```

```
1)  //Uses iostream and cstdlib:
2)  void DayOfYear::input( )
3)  {
4)      cout << "Enter the month as a number: ";
5)      cin >> month;
6)      cout << "Enter the day of the month: ";
7)      cin >> day;
8)      if ((month < 1) || (month > 12) || (day < 1) || (day > 31))
9)      {
10)         cout << "Illegal date! Program aborted.\n";
11)         exit(1);
12)     }
13) }
```

```
1)  void DayOfYear::output( )
2)  {
3)     switch (month)
4)     {
5)        case 1:
6)           cout << "January "; break;
7)        case 2:
8)           cout << "February "; break;
9)        case 3:
10)          cout << "March "; break;
11)       case 4:
12)          cout << "April "; break;
13)       case 5:
14)          cout << "May "; break;
15)       case 6:
16)          cout << "June "; break;
17)       case 7:
18)          cout << "July "; break;
19)       case 8:
20)          cout << "August "; break;
21)       case 9:
22)          cout << "September "; break;
23)       case 10:
24)          cout << "October "; break;
25)       case 11:
26)          cout << "November "; break;
27)       case 12:
28)          cout << "December "; break;
29)       default:
30)          cout << "Error in DayOfYear::output. Contact software vendor.";
31)    }
32)    cout << day;
33) }
```

```
1)  #include <iostream>
2)  #include <cstdlib> //for exit
3)  using namespace std;
4)  int main( )
5)  {
6)      DayOfYear date1(2, 21), date2(5), date3;
7)      cout << "Initialized dates:\n";
8)      date1.output( ); cout << endl;
9)      date2.output( ); cout << endl;
10)     date3.output( ); cout << endl;

11)     date1 = DayOfYear(10, 31);
12)     cout << "date1 reset to the following:\n";
13)     date1.output( ); cout << endl;
14)     return 0;
15)}
```

CONSTRUCTOR WITH NO ARGUMENTS

- Can be confusing
- Standard functions with no arguments:
  - Called with syntax: callMyFunction();
    - Including empty parentheses
- Object declarations with no "initializers":
  - DayOfYear date1; // This way!
  - DayOfYear date(); // NO!
    - What is this really?
    - Compiler sees a **function declaration/prototype**!
    - Yes! Look closely!

EXPLICIT CONSTRUCTOR CALLS

- Can also call constructor AGAIN
  - After object declared
    - Recall: constructor was automatically called then
  - Can call via object's name; standard member function call
- Convenient method of setting member variables
- Method quite different from standard member function call

EXPLICIT CONSTRUCTOR CALL EXAMPLE

- Such a call returns "anonymous object"
  - Which can then be assigned
  - **In Action**:
    DayOfYear holiday(7, 4);
    - Constructor called at object's declaration
    - Now to "re-initialize":

      holiday = DayOfYear(5, 5);

      - Explicit constructor call
      - Returns new "**anonymous** object"
      - Assigned back to current object

Default CONSTRUCTOR

- Defined as: constructor w/ no arguments
- One should always be defined
- Auto-Generated?
  - Yes & No
  - If no constructors AT ALL are defined → Yes
  - If any constructors are defined → No
- If no default constructor:
  - Cannot declare: MyClass myObject;
    - With no initializers

CLASS TYPE MEMBER VARIABLES

- Class member variables can be any type
  - Including **objects** of other classes!
  - Type of class relationship
    - Powerful OOP principle
- Need special notation for constructors
  - So they can call "back" to member object's constructor

CLASS MEMBER VARIABLE EXAMPLE:

Display 7.3 A Class Member Variable

```
1   #include <iostream>
2   #include<cstdlib>
3   using namespace std;

4   class DayOfYear
5   {
6   public:
7       DayOfYear(int monthValue, int dayValue);
8       DayOfYear(int monthValue);
9       DayOfYear( );
10      void input( );
11      void output( );
12      int getMonthNumber( );
13      int getDay( );
14  private:
15      int month;
16      int day;
17      void testDate( );
18  };
```

The class `DayOfYear` *is the same as in Display 7.1, but we have repeated all the details you need for this discussion.*

```
19  class Holiday
20  {
21  public:
22      Holiday( );//Initializes to January 1 with no parking enforcement
23      Holiday(int month, int day, bool theEnforcement);
24      void output( );
25  private:
26      DayOfYear date;
27      bool parkingEnforcement;//true if enforced
28  };

29  int main( )
30  {
31      Holiday h(2, 14, true);
32      cout << "Testing the class Holiday.\n";
33      h.output( );

34      return 0;
35  }
36
37  Holiday::Holiday( ) : date(1, 1), parkingEnforcement(false)
38  {/*Intentionally empty*/}

39  Holiday::Holiday(int month, int day, bool theEnforcement)
40                       : date(month, day), parkingEnforcement(theEnforcement)
41  {/*Intentionally empty*/}
```

member variable of a class type

Invocations of constructors from the class `DayOfYear`.

(continued)

Display 7.3 A Class Member Variable

```
42  void Holiday::output( )
43  {
44      date.output( );
45      cout << endl;
46      if (parkingEnforcement)
47          cout << "Parking laws will be enforced.\n";
48      else
49          cout << "Parking laws will not be enforced.\n";
50  }

51  DayOfYear::DayOfYear(int monthValue, int dayValue)
52                              : month(monthValue), day(dayValue)
53  {
54      testDate( );
55  }
```

```
56  //uses iostream and cstdlib:
57  void DayOfYear::testDate( )
58  {
59      if ((month < 1) || (month > 12))
60      {
61          cout << "Illegal month value!\n";
62          exit(1);
63      }
64      if ((day < 1) || (day > 31))
65      {
66          cout << "Illegal day value!\n";
67          exit(1);
68      }
69  }
70
71  //Uses iostream:
72  void DayOfYear::output( )
73  {
74      switch (month)
75      {
76          case 1:
77              cout << "January "; break;
78          case 2:
79              cout << "February "; break;
80          case 3:
81              cout << "March "; break;
              .
              .
              .
```

The omitted lines are in Display 6.3, but they are obvious enough that you should not have to look there.

Display 7.3 A Class Member Variable

```
82            case 11:
83                cout << "November "; break;
84            case 12:
85                cout << "December "; break;
86            default:
87                cout << "Error in DayOfYear::output. Contact software vendor.";
88        }

89        cout << day;
90    }
```

SAMPLE DIALOGUE

```
Testing the class Holiday.
February 14
Parking laws will be enforced.
```

PARAMETER PASSING METHODS

- Efficiency of parameter passing
  - Call-by-value
    - Requires copy be made → Overhead
  - Call-by-reference
    - Placeholder for actual argument
    - Most efficient method
  - Negligible difference for simple types
  - For class types → clear advantage
- Call-by-reference desirable
  - Especially for "large" data, like class types

THE CONST PARAMETER MODIFIER

- Large data types (typically classes)
  - Desirable to use pass-by-reference
  - Even if function will not make modifications
- Protect argument
  - Use constant parameter
    - Also called constant call-by-reference parameter
  - Place keyword ***const*** before type
  - Makes parameter "read-only"
  - Attempt to modify parameter results in compiler error

USE OF CONST

- All-or-nothing
- If no need for function modifications
  - Protect parameter with const
  - Protect ALL such parameters
- This includes class member function parameters

INLINE FUNCTIONS

- For **non-member** functions:
  - Use keyword ***inline*** in function declaration and function heading
- For class **member** functions:
  - Place implementation (code) for function **IN** class definition
    - automatically **inline**
- Use for very short functions only
- Code actually inserted in place of call
  - Eliminates overhead
  - More efficient, but only when short!

INLINE MEMBER FUNCTIONS

- Member function definitions
  - Typically defined separately, in different file
  - Can be defined IN class definition
    - Makes function "in-line"
- Again: use for very short functions only
- More **efficient**
  - If too long → actually less efficient!

```
1)   class Account
2)   {
3)   public:
4)      Account(double initial_balance) { balance = initial_balance; }
5)      double GetBalance();
6)      double Deposit( double Amount );
7)      double Withdraw( double Amount );
8)   private:
9)      double balance;
10)  };

11)  inline double Account::GetBalance() {    return balance; }

12)  inline double Account::Deposit( double Amount )
13)  {
14)     return ( balance += Amount );
15)  }

16)  inline double Account::Withdraw( double Amount )
17)  {
18)     return ( balance -= Amount );
19)  }
20)  int main() { … }
```

WHICH ONES ARE INLINE FUNCTIONS?

INLINE MEMBER FUNCTION EXAMPLES

- A function defined in the body of a class declaration is an inline function.
- Example
  - the Account constructor is an inline function.
  - The member functions GetBalance, Deposit, and Withdraw are not specified as inline but can be implemented as inline functions.

INLINE FUNCTIONS VS. MACROS

- inline functions are similar to macros
  - the function code is expanded at the point of the call at compile time
    - inline functions are parsed by the compiler
  - macros are expanded by the preprocessor
- Important differences:
  1. Inline functions follow all the protocols of type safety enforced on normal functions.
  2. Inline functions are specified using the same syntax as any other function except that they include the **inline** keyword in the function declaration.
  3. Expressions passed as arguments to inline functions are evaluated once. In some cases, expressions passed as arguments to macros can be evaluated more than once.

EXAMPLE- MACRO

```
1)  // inline_functions_macro.c
2)  #include <stdio.h>
3)  #include <conio.h>

4)  #define toupper(a) ((a) >= 'a' && ((a) <= 'z') ? ((a)-('a'-'A')):(a))

5)  int main() {
6)     char ch;
7)     printf_s("Enter a character: ");
8)     ch = toupper( getc(stdin) );
9)     printf_s( "%c", ch );
10) }
11) //  Sample Input:  xyz
12) // Sample Output:  Z
```

getc is executed to determine whether the character is

1. >= "a," and
2. <= "z.“
3. converted to uppercase.

EXAMPLE- INLINE

```
1)  // inline_functions_inline.cpp
2)  #include <stdio.h>
3)  #include <conio.h>

4)  inline char toupper( char a ) {
5)     return ((a >= 'a' && a <= 'z') ? a-('a'-'A') : a );
6)  }

7)  int main() {
8)     printf_s("Enter a character: ");
9)     char ch = toupper( getc(stdin) );
10)    printf_s( "%c", ch );
11) }
// Sample Input: aSample,  Output: A
```

getc is executed once!

STATIC MEMBERS

- Static member variables
  - All objects of class "share" one copy
  - One object changes it → all see change
- Useful for "tracking"
  - How often a member function is called
  - How many objects exist at given time
- Place keyword *static* before type

STATIC FUNCTIONS

- Member functions can be static
  - If **no access** to object data needed
  - And still "must" be member of the class
  - Make it a static function
- Can then be called outside class
  - From non-class objects:
    - E.g., Server::getTurn();
  - As well as via class objects
    - Standard method: myObject.getTurn();
- Can **only use static data**, **functions**!

STATIC MEMBERS EXAMPLE: DISPLAY 7.6 STATIC MEMBERS (1 OF 4)

Display 7.6 Static Members

```
1   #include <iostream>
2   using namespace std;

3   class Server
4   {
5   public:
6       Server(char letterName);
7       static int getTurn( );
8       void serveOne( );
9       static bool stillOpen( );
10  private:
11      static int turn;
12      static int lastServed;
13      static bool nowOpen;
14      char name;
15  };

16  int Server:: turn = 0;
17  int Server:: lastServed = 0;
18  bool Server::nowOpen = true;
```

```
19  int main( )
20  {
21      Server s1('A'), s2('B');
22      int number, count;
23      do
24      {
25          cout << "How many in your group? ";
26          cin >> number;
27          cout << "Your turns are: ";
28          for (count = 0; count < number; count++)
29              cout << Server::getTurn( ) << ' ';
30          cout << endl;
31          s1.serveOne( );
32          s2.serveOne( );
33      } while (Server::stillOpen( ));
34      cout << "Now closing service.\n";
35      return 0;
36  }
37
38
```

Display 7.6 Static Members

```
39  Server::Server(char letterName) : name(letterName)
40  {/*Intentionally empty*/}

41  int Server::getTurn( )
42  {
43      turn++;
44      return turn;
45  }
46  bool Server::stillOpen( )
47  {
48      return nowOpen;
49  }

50  void Server::serveOne( )
51  {
52      if (nowOpen && lastServed < turn)
53      {
54          lastServed++;
55          cout << "Server " << name
56              << " now serving " << lastServed << endl;
57       }
```

Since `getTurn` *is static, only static members can be referenced in here.*

```
58        if (lastServed >= turn) //Everyone served
59            nowOpen = false;
60    }
```

Sample Dialogue

How many in your group? **3**
Your turns are: 1 2 3
Server A now serving 1
Server B now serving 2
How many in your group? **2**
Your turns are: 4 5
Server A now serving 3
Server B now serving 4
How many in your group? **0**
Your turns are:
Server A now serving 5
Now closing service.

VECTORS

- Vector Introduction
  - Recall: arrays are fixed size
  - Vectors: "arrays that grow and shrink"
    - During program execution
  - Formed from Standard Template Library (STL)
    - Using template class

VECTOR BASICS

- Similar to array:
  - Has base type
  - Stores collection of base type values
- Declared differently:
  - Syntax: vector<Base_Type>
    - Indicates template class
    - Any type can be "plugged in" to Base_Type
    - Produces "new" class for vectors with that type
  - Example declaration:
    vector<int> v;

VECTOR USE

- vector<int> v;
  - "v is vector of type int"
  - Calls class default constructor
    - Empty vector object created
- Indexed like arrays for access
- But to add elements:
  - Must call member function push_back
- Member function size()
  - Returns current number of elements

VECTOR EXAMPLE: DISPLAY 7.7 USING A VECTOR

```
1)   #include <iostream>
2)   #include <vector>
3)   using namespace std;

4)   int main( )
5)   {
6)      vector<int> v;
7)      cout << "Enter a list of positive numbers.\n"
8)           << "Place a negative number at the end.\n";

9)      int next;
10)     cin >> next;
11)     while (next > 0)
12)     {
13)        v.push_back(next);
14)        cout << next << " added. ";
15)        cout << "v.size( ) = " << v.size( ) << endl;
16)        cin >> next;
17)     }

18)     cout << "You entered:\n";
19)     for (unsigned int i = 0; i < v.size( ); i++)
20)        cout << v[i] << " ";
21)     cout << endl;

22)     return 0;
23)  }
```

SAMPLE DIALOGUE

```
Enter a list of positive numbers.
Place a negative number at the end.
2 4 6 8 -1
2 added. v.size = 1
4 added. v.size = 2
6 added. v.size = 3
8 added. v.size = 4
You entered:
2 4 6 8
```

VECTOR EFFICIENCY

- Member function capacity()
  - Returns memory currently allocated
  - Not same as size()
  - Capacity typically > size
    - Automatically increased as needed
- If efficiency critical:
  - Can set behaviors manually
    - v.reserve(32); //sets capacity to 32
    - v.reserve(v.size()+10); //sets capacity to 10 more than size

SUMMARY 1

- Constructors: automatic initialization of class data
  - Called when objects are declared
  - Constructor has same name as class
- Default constructor has no parameters
  - Should always be defined
- Class member variables
  - Can be objects of other classes
    - Require initialization-section

SUMMARY 2

- Constant call-by-reference parameters
  - More efficient than call-by-value
- Can *inline* very short function definitions
  - Can improve efficiency
- Static member variables
  - Shared by all objects of a class
- Vector classes
  - Like: "arrays that grow and shrink"

UNIFORM INITIALIZATION: OUR WAYS TO CONSTRUCT OBJECTS OF A CLASS

```
1)  class Circle
    {
2)      double radius;
3)  public:
4)      Circle(double r) { radius = r; }
5)      double circum() { return 2 * radius*3.14159265; }
6)  };

7)  int main()
    {
8)      Circle foo(10.0);   // functional form
9)      Circle bar = 20.0;   // assignment initialization.
10)     Circle baz { 30.0 };   // uniform initialization.
11)     Circle qux = { 40.0 }; // POD-like

12)     cout << "foo's circumference: " << foo.circum() << '\n';
13)     return 0;
14) }
```

Most existing code currently uses functional form

```
1)  #include <iostream>
2)  #include <string>
3)  #include <vector>
4)  #define NUMBER_OF_SAME_TYPE_CHARS 3;
5)  class FlyweightCharacter; /* Actual flyweight objects class (declaration) */
6)  //  FlyweightCharacterAbstractBuilder is a class holding the properties which are shared by many objects.
7)  //  So instead of keeping these properties in those objects we keep them externally, making objects flyweight.
8)  class FlyweightCharacterAbstractBuilder
9)  {
10)     FlyweightCharacterAbstractBuilder() {}
11)     ~FlyweightCharacterAbstractBuilder() {}
12) public:
13)     static std::vector<float> fontSizes; // lets imagine that sizes be may of floating point type
14)     static std::vector<std::string> fontNames; // font name may be of variable length
15)     static void setFontsAndNames();
16)     static FlyweightCharacter createFlyweightCharacter(unsigned short fontSizeIndex,
17)                 unsigned short fontNameIndex,
18)                 unsigned short positionInStream);
19) };

20) std::vector<float> FlyweightCharacterAbstractBuilder::fontSizes(3);
21) std::vector<std::string> FlyweightCharacterAbstractBuilder::fontNames(3);
22) void FlyweightCharacterAbstractBuilder::setFontsAndNames()
23) {
24)     fontSizes[0] = 1.0;   fontSizes[1] = 1.5;   fontSizes[2] = 2.0;
25)     fontNames[0] = "first_font"; fontNames[1] = "second_font"; fontNames[2] = "third_font";
26) }
```

```
1)  class FlyweightCharacter
2)  {
3)          unsigned short fontSizeIndex; // index instead of actual font size
4)          unsigned short fontNameIndex; // index instead of font name
5)          unsigned positionInStream;
6)  public:

7)          FlyweightCharacter(unsigned short fontSizeIndex,
                    unsigned short fontNameIndex,
                    unsigned short positionInStream): fontSizeIndex(fontSizeIndex),
                                fontNameIndex(fontNameIndex), positionInStream(positionInStream) {}
8)          void print()
9)          {
10)           std::cout << "Font Size: " << FlyweightCharacterAbstractBuilder::fontSizes[fontSizeIndex]
11)                 << ", font Name: " << FlyweightCharacterAbstractBuilder::fontNames[fontNameIndex]
12)                 << ", character stream position: " << positionInStream << std::endl;
13)         }
14)         ~FlyweightCharacter() {}
15)};

16) FlyweightCharacter FlyweightCharacterAbstractBuilder::createFlyweightCharacter(
            unsigned short fontSizeIndex, unsigned short fontNameIndex,
            unsigned short positionInStream)
17){
18)         FlyweightCharacter fc(fontSizeIndex, fontNameIndex, positionInStream);
19)         return fc;
20)}
```

```
1)  int main(int argc, char** argv)
2)  {
3)     std::vector<FlyweightCharacter> chars;

4)     FlyweightCharacterAbstractBuilder::setFontsAndNames();

5)     unsigned short limit = NUMBER_OF_SAME_TYPE_CHARS;

6)     for (unsigned short i = 0; i < limit; i++)
7)     {
8)       chars.push_back(
             FlyweightCharacterAbstractBuilder::createFlyweightCharacter(0, 0, i));
9)       chars.push_back(
             FlyweightCharacterAbstractBuilder::createFlyweightCharacter(1,1,i+1*limit));
10)      chars.push_back(
             FlyweightCharacterAbstractBuilder::createFlyweightCharacter(2,2,i+2*limit));
11)  }

12)  for (unsigned short i = 0; i < chars.size(); i++)
13)  {
14)          chars[i].print();
15)  }

16)  return 0;
17)}
```

- Each char stores links to it's fontName and fontSize so what we get is:

- each object instead of allocating 6 bytes (assumed avg len) for string
- and 4 bytes for float allocates 2 bytes for fontNameIndex and fontSizeIndex.

- That means for each char we save 6 + 4 - 2 - 2 = 6 bytes.
- Now imagine we have NUMBER_OF_SAME_TYPE_CHARS = 1000 i.e. with our code
- we will have 3 groups of chars with 1000 chars in each group which will save us

- 3 * 1000 * 6 - (3 * 6 + 3 * 4) = 17970 saved bytes.

- 3 * 6 + 3 * 4 is a number of bytes allocated by FlyweightCharacterAbstractBuilder.

- So the idea of the pattern is to move properties shared by many objects to some
- external container. The objects in that case don't store the data themselves they
- store only links to the data which saves memory and make the objects lighter.
- The data size of properties stored externally may be significant which will save REALLY huge amount of memory and will make each object super light in comparison to it's counterpart.
- That's where the name of the **pattern** comes from: flyweight (i.e. very light).