

CH-230-A

# **Programming in C and C++**

C/C++

## **Tutorial 10**

Dr. Kinga Lipskoch

Fall 2020

## Instances of the Same Class

- ▶ Instances of the same class have the same set of data, but they are replicated so that they do not overlap

```
1  Critter a, b;  
2  a.setHunger(1);  
3  b.setHunger(4);  
4  cout << a.getHunger() << " "  
5      << b.getHunger();
```

will print

1 4

- ▶ a and b have a different memory space, so their modifications are independent

## When Should Data Members be `public`?

- ▶ The interface of a class should be *minimal*
  - ▶ This gives least commitments in what you should keep untouched in order to avoid modifying client code
- ▶ Exceptions: if you need to access a data very frequently, the use of setter and getter methods may result in a bottleneck (after all it is a function call)
- ▶ In those cases you could consider to make a data member `public` (but you can also declare the method as `inline`)

# Initialization of an Object

- ▶ When you declare an instance of a class its data members are **not** initialized
- ▶ It is possible to define a piece of code to be executed when the instance is created such that it brings the class into a "consistent" state
  - ▶ Remember the problem that in C (older standards) variables are not initialized
- ▶ This piece of code is called **constructor**
- ▶ A **constructor** is a special function, which is automatically called at object creation

## Constructors (1)

- ▶ A constructor can be declared as "a method" with *no return type information and with the same name as the name of the class*
  - ▶ No return type is different from `void`
- ▶ The definition of a constructor is like that of a method
- ▶ A constructor can take parameters to allow parametric initialization
  - ▶ Provides a way to guarantee that the object is initialized with appropriate values
- ▶ There can be more constructors, provided that they take a different parameter list (overloading, to be covered later)
- ▶ `Complex.h`    `Complex.cpp`    `testcomplex.cpp`

## Constructors (2)

- ▶ In the case of constructors with parameters they have to be specified at declaration time
- ▶ The choice among overloaded constructors is done on the basis of the effective parameter list given at object creation
- ▶ It is possible to specify default values for parameters
  - ▶ But they must be at the end of the parameter list

## Default Constructors

- ▶ If a constructor is not defined, the compiler will create one taking no arguments ([default constructor](#))
- ▶ If at least a constructor is defined by the programmer, the compiler will not generate the [default constructor](#)
- ▶ [Default constructors](#) do not initialize properties to 0
- ▶ *Good programming practice*: always define your own constructors (also the default one)
- ▶ `constructorexample.cpp`

## Constructors of Sub-Objects

- ▶ A class can have objects as data members (like name in the student class)
- ▶ These objects need to be initialized during constructor execution
- ▶ Let us revise the class `Critter` and add a constructor which initializes all data members
- ▶ `Critter2.h`    `Critter2.cpp`    `testcritter2.cpp`



## Syntax Details

```
1 A::A(B par1, B par2, int par3) : member1(par1),  
2   member2(par2) {  
3     // do something with par3  
4 }
```

- ▶ `subobject.cpp`
- ▶ The order of the constructor calls is determined by how data members are declared in the class declaration and not by the order of calls in the constructor definition
- ▶ `callsequence.cpp`

## callsequence.cpp

```
1  #include <string>
2  using namespace std;
3  // simple class just to make an
4  // example of how constructors of internal objects are called
5  class ToTest {
6  private:
7      string First;  // dummy data
8      string Second;
9      string Third;
10     int anint;
11 public:
12     ToTest(string, string, string, int);
13     // constructor: one parameter for each data member
14 };
15 ToTest::ToTest(string a, string b, string c, int d)
16 : Second(b) , Third(c) , First(a) {
17     // no matter the order here indicated, the first object to be initialized
18     // will be First, the second will be Second, the Third will be third,
19     // according to how they were declared in the class definition
20     // (an disregarding the order in the constructor definition
21     anint = d;
22 }
23 int main(int argc, char** argv) {
24     ToTest aninstance("Jacobs", "EECS", "320142", 1);
25     /*    aninstance.First will be the string "Jacobs"
26          aninstance.Second will be the string "EECS"
27          aninstance.Third will be the string "320142" */
28     return 0;
29 }
```

# Destructors (1)

- ▶ A **destructor** is the companion concept of the constructor
  - ▶ It provides operations to be done when an object instance is removed from memory
    - ▶ Typical use: releasing resources acquired during object lifecycle
- ▶ **Destructors** do not take parameters, do not return any type and their name is that of the class preceded by a `~` character
- ▶ `destructors.cpp`

## destructors.cpp

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  /* This example illustrates that destructor call are handled by the
5     compiler. Please compile this code and observe the output */
6  class ToTest {
7  private:
8     string name;
9  public:
10     ToTest(char*);
11     ~ToTest();
12     void doSomething();
13 };
14 ToTest::ToTest(char* n) : name(n) {
15     cout << "Executing " << name << "'s constructor" << endl;
16 }
17 ToTest::~ToTest() {
18     cout << "Executing " << name << "'s destructor" << endl;
19 }
20 void ToTest::doSomething() {
21     cout << "Doing something with " << name << endl;
22 }
23 int main(int argc, char** argv) {
24     ToTest a("FIRST"); {
25         ToTest b("SECOND");
26         a.doSomething();
27         b.doSomething();
28         // b's destructor will be called here, as it is going out of scope
29     } // a's destructor will be called here
30     return 0;
31 }
```

## Destructors (2)

- ▶ Destructor calls are managed by the compiler
  - ▶ The destructor is called when the object goes out of scope or when it is removed from the heap (more on this later)
- ▶ *Good programming practice*: if your class allocates memory on the heap, it should de-allocate it while executing the destructor
  - ▶ Your programs must avoid *memory leaks* (i.e., incorrect management of memory allocations)

## Overloading (1)

While programming it is often useful to indicate with the same name more entities with related functionality

- ▶ Example: to print on the screen it would be nice to have a family of functions, say called `screen_print`, which are used to print data of different types

```
1  screen_print("prints a string");  
2  int a;  
3  screen_print(a);  
4  float b;  
5  screen_print(b);
```

- ▶ Their implementation is however different, thus requiring different calls

## Overloading (2)

- ▶ From the argument list the compiler can infer which version of the function needs to be called
- ▶ Every function or method is associated with a signature:
  - ▶ The signature is the concatenation of the *name* and of the *parameters type* (the order is relevant)
    - ▶ The name of the parameters and the return type do not appear in the signature. Why?
- ▶ Overloading is possible if this yields different signatures
- ▶ `overloading.cpp`

## overloading1.cpp

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Student {
6  private:
7      string name;
8      float grade;
9  public:
10     Student(const string& n, int grade) {
11         name = n ;
12         cout << n << " initialized with an integer grade." << endl;
13     }
14     Student(const string& n, double grade) {
15         name = n ;
16         cout << n << " initialized with a float grade." << endl;
17     }
18 };
19 int main() {
20     Student first("Anni Friesinger", 2);
21     // parameters determine which constructor will be called
22     Student second("Claudia Pechstein", 2.3);
23     return 0;
24 }
```



## Overloading: When?

- ▶ If a class offers the same service (i.e., method) for different data types and just the implementation changes, this is a good candidate for overloading
- ▶ Class users need to remember just one usage policy
- ▶ Overloading increases the capacity to abstract during coding

## Pointers to Classes

- ▶ Being types, it is possible to declare pointers to a class as for basic data types. The syntax is the usual

```
1  string a("this is one string");  
2  string *ptr;  
3  ptr = &a;
```

- ▶ The special operator `->` allows to access data members and call methods through pointers

```
1  cout << ptr->substr(1, 3) << endl;  
2  cout << (*ptr).substr(1, 3) << endl;
```

- ▶ Access to data members and methods through pointers is subject to the data hiding restrictions