CH08-320142

Programming in C++ I

C++ I

Lecture 3 & 4

Dr. Kinga Lipskoch

Fall 2018

Agenda Week 2

- More on classes
- Constructors
- Destructors
- Overloading
- Copy constructor
- Dynamic memory allocation
- The `const` keyword
- Conditional compilation
- Inheritance

More on Methods

- There is a strong correspondence between method calls and function calls
- A method:
  - Can accept parameters
  - Returns a typed value to the caller
  - Can call functions and other methods, including itself
  - Can include iterative cycles, local variables declaration, etc.
- A method is allowed to access data members and other methods in its class

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)
- `constructorsexample.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 $\sim$ 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

C++ References (1)

A reference is a constant pointer which is automatically dereferenced and that has to be initialized when it is created

- Constant means that it cannot be modified to reference a different entity
  - But you can of course modify what it is pointing to
- Cannot reference NULL

```
1   int a = 3;
2   int &reference = a;
3   reference++;
4   cout << a;          // prints 4
```

C++ References (2)

- References create a synonym (alias)
  - Previous example: acting on a reference is the same that acting on the variable a
- The first use of references is for creating functions and methods having out parameters
  - Indeed C++ references can be used even if you do not exploit the object-oriented capabilities of the language
- `outparameters.cpp`

outparameters.cpp

```
1  #include <iostream>
2  using namespace std;
3  /* This function takes two parameters. Only modifications
4     done on the first one are visible outside. */
5  void oldStyle(int *outval, int inval)
6  {
7    cout << "Inside oldStyle" << endl;
8    inval++;
9    (*outval)++;                      // need to dereference
10 }
11 /* Also this function takes two parameters. Again, only
12   modifications done on the first one are visible outside. */
13 void newStyle(int &outval, int inval)
14 {
15   cout << "Inside newStyle" << endl;
16   inval++;
17   outval++;                         // no need to dereference
18 }
19 int main(int argc, char** argv)
20 {
21   int a = 0, b = 0;
22   cout << a << " " << b << endl;
23   oldStyle(&a, b);                  // needs to take the address
24   cout << a << " " << b << endl;
25   a = b = 0;                        // reset to initial values
26   newStyle(a, b);                   // no specific syntax to pass the parameter
27   cout << a << " " << b << endl;
28   return 0;
29 }
```

Passing `const` Object References

The usual way to pass an input parameter to a function or method, is to pass it as a `const` reference

- Improved efficiency + cannot modify
  - No need to create a temporary copy of the object
- No need to define a copy constructor (more soon)

```
1  void method(const string& byvaluepar) {
2    // use it as a constant object
3  }
```

- All previous examples should be rewritten according to this indication

Passing Objects by Reference

- Another case: use a reference when you wish to pass an object by reference, i.e., if modifications have to be seen outside

```
1 void modifyString(string& tomod) {
2   tomod.assign("new value");
3   // non const as modification has to be seen
4 }
```

- The use is consistent with basic data types

The Copy Constructor

To correctly manage by value argument passing for objects it is necessary to define a copy constructor:

- For class X a copy constructor has the form:

```
1   X::X(const X&);
```

- If defined, this will replace bit-copy (exact, bit by bit copy of an object) when passing by value object parameters
- Its goal is to correctly create a copy of an object starting from an existing one
- `copyconstructor.cpp`

Compiler Generated Constructors

To summarize, the compiler can generate two types of constructors:

- The default constructor, taking no arguments
  - This is generated only if you do not provide any constructor
- The copy constructor, which performs bit-copy initialization from an existing object
  - This is not generated if you either provide an `X::X(const X&)` implementation or you declare a `private X::X(const X&)` constructor
  - The `private X::X(const X&)` constructor does not need to be implemented

Dynamic Memory Allocation

- C++ has an operator for dynamic memory allocation
  - It replaces the use of the C `malloc` function
  - Easier and safer
- The operator is called `new`
  - It can be applied both to user defined types (classes) and to native types

Using `new` for Predefined Data Types

- The operator returns a pointer to a specified type
- It automatically calculates the amount of memory necessary
  - It only requires the type and the number of "objects" to hold
- `newarrays.cpp`
- Note: same syntax for pointers to different types (no casting needed)

Dynamic Memory Deallocation

- As `malloc` has the companion function `free`, the operator `new` is coupled with the operator `delete`, which removes an object from memory
- `delete` requires the address of the object(s) to be deleted from the memory

```
1   int *a, *b;
2   a = new int;
3   b = new int[40];
4   delete a;
5   delete []b;
```

More on `delete`

- When `delete` is called to remove an object, the destructor is invoked before removing it
- Calling `delete` twice (or more) on the same object will result in an undefined behavior
- Calling `delete` on a `NULL` pointer will do nothing
  - Thus it could be advisable to set a pointer to `NULL` after calling `delete` (further `delete` will have no effect)
- Do not mix calls to `new` / `delete` with `malloc` / `free` – similar purpose, but predictably very bad result

`new` and `delete` for Arrays of Objects

- It is possible to dynamically create arrays of objects (instances of classes)
  - There must be a default (empty) constructor for the class (which will be called for every element of the array)
  - An array of objects must be explicitly deleted, by using the following syntax
    `delete []ptr;`
  - In this way the compiler is able to call the destructor for every element before freeing memory
  - There is no need to specify how many elements
- `studentsrevised.cpp`

Even More on `delete`

- When you create objects via `new`, you should destroy them via `delete`
  - All the memory you get from the operating system should be returned
  - Again, your programs must avoid memory leaks
- Most of the bugs in early stage are due to bad / misplaced calls of `delete`
  - Many memory related errors cause severe problems

Constants (1)

- As in C, the keyword `const` is used to define values that do not change
- In C++ the use of constants is wider
  - Constants should be used instead of the preprocessor `#define` directive

```
1  // avoid #define SIZE 100
2  const int SIZE = 100;
3  // use this instead
```

    - Why? Preprocessor directives can hide bugs which are nasty to find
  - Constants can be inserted into header files, name clashes will be detected by the compiler

Constants (2)

- Methods or method parameters can be declared as `const`
  - Does not add information to the outside, but rather force the compiler to check that no modifications are attempted
  - Useful when dealing with temporarily generated objects
  - Useful for efficient parameter passing (more soon)
- `constantparameters.cpp`

`const` Objects

- Again, as classes are types, it is possible to declare `const` objects or to declare a method which accepts a `const` object
  - The syntax is the same
- For a `const` object it is not possible to modify its public data members

Constant Methods

- A constant method is a method which does not alter the object. Thus
  - It cannot modify data members
  - It may call only other `const` methods
- Constant methods are the only methods which can be called for constant objects
- `constclass.cpp`

Multiple Inclusions

- Class declarations go to header files
- Header files will be included in all the `.cpp` files that need their declarations
- What if a header file is included twice?
  - A repeated class declaration is an error
    - But a repeated function declaration is not (as long as the declarations are the same)
- Should the programmer take care of not including the file twice?
  - Almost impossible in big projects

Conditional Compilation

- The preprocessor can be used to avoid multiple inclusions
- The `#ifdef`, `#ifndef`, `#else`, `#endif` directives allow to exclude some parts of the code according to specified conditions
- They are to be used with the `#define` that you already know

The Structure of a Header File

```
1 /* Student.h */
2 #ifndef _STUDENT_H
3 #define _STUDENT_H
4 class student {
5   /* your class declaration */
6 };
7 #endif // this matches the initial #ifndef
```

How Does This Work?

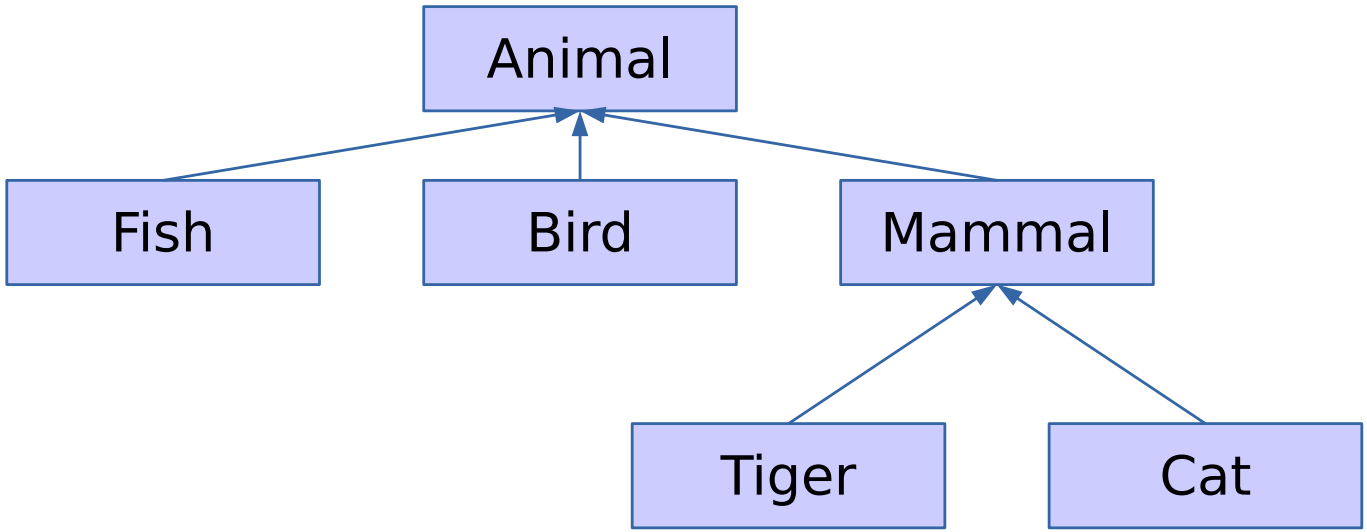
- The first time the header is included the symbol `_STUDENT_H` is not defined
  - Then the class declaration is compiled and then the symbol is defined
- In all the subsequent inclusions the symbol is already defined and then the class declaration is skipped
- You must always protect (or guard) your header files with this mechanism

Inheritance (1)

- Inheritance is used to create a new class from an existing one
  - The new (derived) class inherits what has been defined in the class (base) from which it derives
  - Great advantage in safely reusing existing code and increased abstraction capability
- Code reuse
  - Create new class as type of existing class
  - Take form of existing class and add code to it
  - Inherit from base class

Inheritance (2)

- Relationships between classes are usually graphically illustrated with trees
- Generally it is not always a tree since a node may have more than one parent, but for the moment we will consider a tree



Inheritance (3)

- Inheritance models the IS-A logic relationship between objects
  - A mammal IS-A(n) animal. Thus everything holding for an animal holds also for a mammal
    - A tiger IS-A mammal, and by transitive law, also an animal
    - Also a cat IS-A mammal, and then an animal, but different from a tiger
- To inherit means that properties in the base class are transferred into the derived class

Inheritance (4)

The IS-A relationship means that the interface of the base class is inherited by the derived class

- Thus every message which can be sent to base class can be sent to a derived class (method call)
- What is not visible in the base class will be not usable in the derived class, but will be there
  - Otherwise some information would be lost, and the interface methods could also not provide what they should

Inheritance in C++

- To create a derived class, use the following syntax:

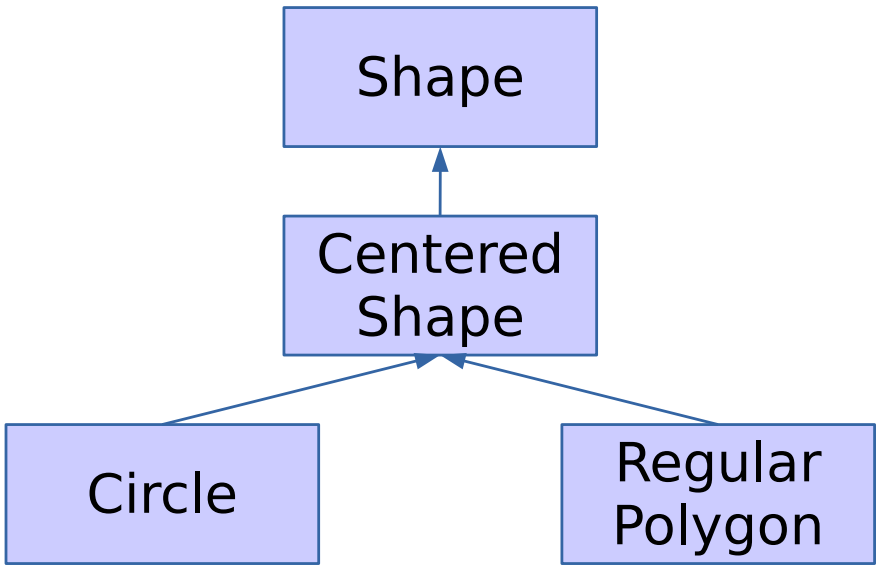
```
class A {
  // whatever you need
};

class B : public A {
  // B specific methods and data members
};
```

- Do not forget the `public` modifier, otherwise the compiler will make some not frequently used assumptions

Inheritance: An Example

- The classic shape example: devise a hierarchy of classes to model a rough drawing system
- `Shapes.h`



Initialization and Inheritance

- When initializing an object, constructors are called in sequence top to down starting from the root
- You can specify parameters in the constructor list Example:

```
1   B(int na, double nb): A(na) {
2      b = nb;
3   }
```

- Shapes.cpp

A Game Example

- `creature.cpp`

Inherited Interface

- `testshapes.cpp`
- On one hand, it is possible to call the method `printName` also on instances of `Circle` or `RegularPolygon`
  - Because it is part of the interface
- On the other hand, inside of `Circle` it is not possible to access `name`, because it is not part of the interface
  - The data however is there, otherwise `printName` would not work

Upcasting (1)

- As the interface of the base class is a subset of the interface of the derived class, it is possible to use a derived class where a base class is required
  - If a shape is required, then only its interface will be used. But its methods and data members are also in the interface of circle, so it is possible to use an instance of `Circle` as "`Shape`"
- Upcasting = objects are "pushed up" on the derivation tree

Upcasting (2)

- An example:

```
1   Shape *collection[2];
2   collection[0] = new Circle("A", 1, 1, 3);
3   collection[1] =
4       new RegularPolygon("B", 1, 1, 6);
5   collection[0]->printName();
6   collection[1]->printName();
```

- It works, as every message that can be sent to the base class can be sent to the derived class
- This will show its enormous power when we will deal with polymorphism

Inheritance & Destructors

- When a derived object is removed from memory, destructors are called from the leaf up to the root in the derivation tree
  - i.e., the order is reversed compared to the constructors
  - This prevents dependency problems between objects
- `simpleinheritance.cpp`

`protected`: A Third Level of Information Hiding

- We have seen that class elements can be either `public` or `private`, and what this means in terms of accessibility
- There exists a third access modifier: `protected`
- Protected means that the element is not accessible outside the class, but it is accessible in derived classes
  - It is somehow between `private` and `public`

Information Hiding: Scenario with `public` Inheritance

	Accessible outside the class	Accessible in derived class
private	no	no
protected	no	yes
public	yes	yes

There is More about Inheritance

- We have seen single inheritance. C++ allows also multiple inheritance, i.e., a class with two or more parents
  - Not always easy to use. In most cases you can do without it. Some OOP languages (like Java) do not even provide multiple inheritance
    - We will discuss multiple inheritance later (example)
- We have seen `public` inheritance: it is also possible to specify `private` or `protected` inheritance
- See 13.6 in C++ Annotations or Eckel's book for details

`protected` and `private` Inheritance

Derivation	Access rights in base class	Access rights in derived class
public	public	public
	protected	protected
	private	not accessible
private	public	private
	protected	private
	private	not accessible
protected	public	protected
	protected	protected
	private	not accessible

Summary

- A derived class is created by
  `class derivedclass : public baseclass`
- A derived class has all data of its direct and indirect base classes
- A derived class can directly access data and methods which are declared as `public` or `protected`
- If the base class does not have a default constructor, a derived class must call a constructor of the base class explicitly