

CH08-320142

# Programming in C++ I

C++ I

## Lecture 5 & 6

Dr. Kinga Lipskoch

Fall 2018

## Agenda Week 3

- ▶ The `bool` data type
- ▶ More on namespaces
- ▶ The `static` keyword
- ▶ Inline methods
- ▶ The implicit pointer `this`
- ▶ The `friend` keyword
- ▶ Operator overloading
- ▶ Polymorphism
- ▶ Abstract classes
- ▶ The `virtual` keyword

## C++ bool Data Type

- ▶ C++ introduces a basic data type for dealing with boolean variables
- ▶ Its name is `bool` and the constants `true` and `false` can be used to assign values to a `bool` variable  
`bool a = true;`
- ▶ Usual C conventions still hold: `false` is converted to 0 and `true` to 1
- ▶ Every `int` not equal to 0 is converted to `true`, and 0 is converted to `false`

# C++ Boolean Operators

- ▶ As not all the keyboards easily provide the keys for the C boolean operators (&, |, ^, etc) in C++ the following operators are introduced (in the header `<ciso646>`)
  - ▶ and, or, not, not\_eq, bitand, and\_eq, bitor, or\_eq, xor, xor\_eq, compl

# Namespaces (1)

- ▶ While developing large projects, the risk of running into a name clash is high
  - ▶ Multiple programmers could use the same names for their classes, functions, etc. At the linking stage, name collisions can arise
  - ▶ You can have the same problem when using third party developed libraries
- ▶ Solutions found in the past, consisting on appending specific prefixes, are not appealing

## Namespaces (2)

- ▶ A namespace introduces a further level of code protection
- ▶ Elements belonging to the same namespace can refer to each other without any special syntax
- ▶ Elements in different namespaces can refer to each other just by using a designed syntax
  - ▶ They have to explicitly declare that they are referring to a different namespace

## Creating a Namespace

- ▶ A namespace is created using the namespace keyword at the file level

```
1 namespace CPPcourse {  
2     void f1() { ... }  
3     class class1 { ... };  
4 }
```

- ▶ Namespace declaration can be split over multiple files without creating redefinition problems
- ▶ namespace.h
- ▶ namespace.cpp

## Using Names from a Namespace

Three ways:

- ▶ Import the whole namespace  
`using namespace CPPcourse;`
- ▶ Import a specific name from a namespace

```
1 using CPPcourse::FirstExample;  
2 FirstExample a("Try this");
```

- ▶ Using complete name specification  
`CPPcourse::FirstExample a("Try this");`



## Examples Revised

- ▶ Then in all former examples  
`using namespace std;`  
was introduced to use standard C++ classes, which are declared in the `std` namespace
- ▶ In header files we have used full name specification  
`std::string name;`
- ▶ Never use the `using` directive in a header file, use full name qualification instead
  - ▶ While writing a header file you do not know what your potential client will need in terms of namespaces

## Final Remarks on Namespaces

- ▶ If a namespace's name is too "awkward" to use, it is possible to create an alias

```
namespace shortName = AliasForANameTooLongToBeUsed;
```

- ▶ From now on we can use shortName instead of the alias it points to
- ▶ Namespaces can be nested
- ▶ More details in Eckel's book (chapter 10)

## static Data Members

- ▶ A `static` data member is shared among all the instances of a class
  - ▶ It creates a sort of class variable
- ▶ It exists even if no instances are created
- ▶ Storage must be explicitly allocated outside of class definition
- ▶ Can be useful to define class constants
  - ▶ Using `const` as modifier for a data member does not yield the desired results (as class constant)
- ▶ `staticexample.cpp`

## static Methods

- ▶ Also methods can be declared as `static`
- ▶ Static methods can access only static data members and can call only static methods
- ▶ Static methods can be called referring to an instance or to the class
  - ▶ Like `static` data members they are class methods
- ▶ `staticshapes.cpp`

## When Should we Use `static`?

No general rules, but some generic indications:

- ▶ When creating class level constants
- ▶ When you devise some information which belongs to the class rather than to instances
- ▶ When a method needs to access data members but it is not logically tied to a specific instance

## Inline Methods (1)

- ▶ C is well appreciated as it is an efficient language
  - ▶ The UNIX operating system relies on C
- ▶ C++ cannot give up C efficiency
- ▶ Inline methods are designed to improve the performance of C++ programs
  - ▶ No semantic alterations w.r.t. non-inline methods

## Inline Methods (2)

- ▶ A method call is equivalent to a procedure call
  - ▶ Push arguments onto the stack (or register)
  - ▶ Execute a CALL-like instruction
  - ▶ Execute function/method code and then return
  - ▶ Stack cleanup
- ▶ For small methods the overhead of the call could take more time than code execution
  - ▶ Think for example of getter or setter methods, where you have just one instruction as body
  - ▶ Moreover those methods are likely to be called frequently

## Inline Methods (3)

- ▶ An `inline` function is expanded in place, rather than called
  - ▶ Instead of a regular call, function code is directly inserted
- ▶ You trade off speed for size
  - ▶ No call overhead, but your code could grow as the body of the function will be copied many times
- ▶ Good candidates for being `inline` are short methods that are frequently called



## Inline Methods (4)

How to create inline methods - two possibilities:

- ▶ Put the definition of the method inside the class declaration

`inlineinside.h`      `inlineinside.cpp`

- ▶ Use the keyword `inline` and write the definition outside the class declaration

`inlineoutside.h`    `inlineoutside.cpp`

Put the `inline` function definition in the same header file where the class is declared

## Inline Methods: How Do they Work?

- ▶ When the compiler finds the definition of an inline method it stores its signature and its code in its symbol table
- ▶ When it finds a call to an inline method it checks type correctness and replaces/copies the code
  - ▶ C preprocessor macros offered similar advantages, but no type checking was enforced
    - ▶ Nasty to find bugs which could be generated
    - ▶ Preprocessor macros have no concept of scoping

## Inline Methods: Final Remarks

- ▶ Not everything declared as `inline` by the programmer will necessarily be inlined by the compiler (`inline` is just a hint)
  - ▶ If a method includes loops it is unlikely that it will be expanded
- ▶ Defining inline methods outside class declaration increases code readability
- ▶ Multiple inclusions of headers with inline methods will not result in redefinition problems

## The Implicit Pointer `this`

- ▶ The reserved keyword `this` is a pointer to the current instance of a class
- ▶ `this` is silently passed by the compiler as an argument to every method call
  - ▶ Except of course to static methods. Why?
- ▶ `thisexample.cpp`
- ▶ Will be very useful when implementing overloaded operators

## friend Functions

It is possible to "break" the protection mechanism, i.e., to let a class or a function access non-public data members of a class

- ▶ Is this needed? Sometimes yes, if getting through the getter and setter methods becomes difficult to manage
- ▶ We will see that this will be very important while redefining operators
- ▶ Be aware: when using `friend` elements, you break the information hiding mechanism
- ▶ Do not misuse it

## friend: How to Create

- ▶ In the class declaration, declare a "method" with the `friend` modifier
  - ▶ That indicates a function which can access class data members
  - ▶ The function has to be defined later, but remember that it is not a method
  - ▶ `friendexample.cpp`
- ▶ It is also possible to create friend classes, i.e., classes which can access private data of other classes

## Linking (1)

- ▶ While developing non-trivial projects, it is useful to split the code into multiple files
  - ▶ For example, each class could be coded in a separate couple of files (i.e., the header and the implementation file)
- ▶ The compiler takes an implementation source file and produces an object file
  - ▶ An object file is something "almost ready to be executed", but with dangling external references

## Linking (2)

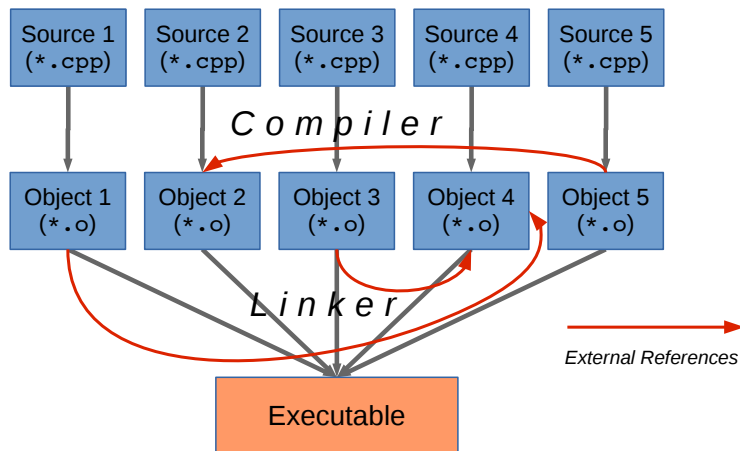
- ▶ To compile a source file, the compiler needs only the declaration of the classes used
  - ▶ Thus you include only header files
- ▶ This needs only the names and the types because of checking the correctness of calls, declaration, etc.
- ▶ If a source file refers to something declared but not defined, a dangling link is introduced



## Linking (3)

- ▶ When building the executable, the linker puts together multiple object files and produces one single executable
- ▶ While putting them together, it tries to resolve dangling references
- ▶ If it finds multiple definitions of the same entity, a linker error is raised
  - ▶ Which one should it use?

## Linking (4)



# Function Overloading

```
1  #include <iostream>
2  using namespace std;
3  int division(int dividend, int divisor)
4  {
5      return dividend/divisor;
6  }
7  float division(float dividend, float divisor)
8  {
9      return dividend/divisor;
10 }
11 int main()
12 {
13     int ia = 10;
14     int ib = 3;
15     float fa = 10.0;
16     float fb = 3.0;
17     cout << division(ia, ib) << endl;
18     cout << division(fa, fb) << endl;
19     return 0;
20 }
```

# Operator Overloading (1)

- ▶ Overloaded operators allow a different syntax for function/method calls
- ▶ Operator overloading allows the programmer to use language operators for user defined data type. For example

```
1   Car first, second, third;  
2   ...  
3   third = first + second;
```

what this could mean is left to the programmer

- ▶ New operators cannot be introduced

## Operator Overloading (2)

- ▶ It is possible to overload operators only for user defined data types
  - ▶ Thus it is not possible to alter the meaning of the `+` operator between `ints`, `floats`, and so on
    - ▶ C code should compile with C++ compilers without changing its functionality
- ▶ Overloading an operator should help the reader of the program and not the programmer
  - ▶ You can define an operator `+` between two instances of a salary class which subtracts them
  - ▶ Whether this makes sense or not is up to your design choices and skills

## Sorting Students by Grade

### With overloaded operators

```
1 Student *list;  
2 ...  
3 for (i = ...)   
4     if (list[i] > list[j]) {  
5         tmp = list[i];  
6         list[i] = list[j];  
7         list[j] = tmp;  
8     }  
9 ...
```

### Without overloaded operators

```
1 Student *list;  
2 ...  
3 for (i = ...) {  
4     if (list[i].getGrade() >  
5         list[j].getGrade()) {  
6         ...  
7     }  
8 }  
9 ...
```

Which one is “cleaner”?

## Using/Calling Operators

- ▶ From the previous example it should be clear that no magic things are happening, but just a compact (and clear) syntax for method calls can be used
- ▶ Whenever the compiler finds an operator involving user defined types, it verifies if it is possible to find a proper definition
- ▶ Operators are **overloaded** as there can be more than just one version: + between two instances of car, + between an instance of car and an instance of bike, and so on
- ▶ Types help in determining the version which should be called

## Operators, Methods or friend Functions

- ▶ To carry out their task operators need to access class data
- ▶ Then, they have to be either methods or friend functions
  - ▶ There are some guidelines  
`Student3.h`   `Student3.cpp`   `studentoperator.cpp`
- ▶ It is possible to overload both unary and binary operators



## Overloading the = Operator

- ▶ Overloading the = operator is fundamental if your class deals with pointers as properties
- ▶ Language provided = operator performs field to field copy
- ▶ If the class has pointers, different instances end up with sharing the same memory
  - ▶ Also, there could be memory leaks  
`charbuffer.h`    `charbuffer.cpp`
- ▶ The operator = must be a method and must return a reference to the class
  - ▶ This allows iterated assignments (`a=b=c;`)

# Overloadable Operators

Unary Operators	Binary Operators
<code>+ - &amp; ! ~</code> <code>++ --</code> (both prefix and postfix) <code>[ ] -&gt; ( )</code>	<code>+ - * / % ^ &amp;   &lt;&lt;</code> <code>&gt;&gt; += -= *= /= %= ^=</code> <code>&amp;=  = &gt;&gt;= &lt;&lt;= == !=</code> <code>&lt; &gt; &lt;= &gt;= &amp;&amp;    -&gt;*</code>

new and delete can be overloaded as well

## What Should Operators Return?

- ▶ A reference, if they modify the involved argument(s) (like `=`, `+=`, etc.)
- ▶ Return a reference to the modified object, usually by using `this`
- ▶ A new instance if they do not modify arguments, but rather use them to produce new information (like most binary operators: `+`, `-`, etc.)
- ▶ A `bool` if it is a boolean operator

## Member or friend?

The following table proposed in the textbook can be taken as a general guideline

Operator	Use
Unary operators	Member
= ( ) [ ] -> ->*	Must be member
+= -= /= *= ^= &=  = %= >>= <<=	Member
Other binary operators	Friend or member

`complex.h``complex.cpp``testcomplex2.cpp`

## Overloading << and >>

- For example for the Complex class:

```
1      friend ostream& operator<<(ostream& out,  
    const Complex &z)  
2  
3      friend istream& operator>>(istream& in,  
    Complex &z)
```

# Polymorphism

- ▶ The last cornerstone of OOP has no correspondence in non-OOP languages
- ▶ It deals with writing code which can correctly operate even with data types unknown at compile time
- ▶ Just one additional keyword, `virtual`, which offers a wide range of applications

## Starting from Upcasting

- ▶ Upcasting means that a derived class can be used wherever an ancestor class is expected
  - ▶ This because the interface of the parent is inherited
- ▶ A derived class can redefine a method, i.e., it can provide a new implementation
- ▶ `randomnonvirtual.cpp`
- ▶ `randomvirtual.cpp`

## What Happens?

- ▶ In the first example the redefinition of the method was not perceived, while in the second the expected behavior was observed
- ▶ It should be evident that at compile time there is not enough information to bind the method call to the right method to execute
  - ▶ This is called **late binding**



## The virtual Keyword

- ▶ When a method is declared as `virtual`, late binding is requested
- ▶ `virtual` should be specified just in the declaration of the base class
- ▶ Late binding is inherited from there on
  - ▶ The redefinition of virtual methods is called **overriding**
- ▶ When dealing with virtual methods, each object carries some sort of info to make its identification possible at runtime

## virtual and Pointers

- ▶ Be aware: the late binding feature works only if you deal with pointers to instances, and not if you directly work with instances
  - ▶ And of course, also with C++ references
- ▶ When passing objects to methods/functions, using C++ references speeds up parameters passing and enables polymorphism

# The Added Value of Polymorphism

- ▶ By mean of polymorphism it is possible to write general purpose code
  - ▶ When possible, general code always deals with base classes, and calls virtual methods
- ▶ Then, the code will work even with later introduced data types (new classes)

# What Should be virtual?

- ▶ The decision is up to the designer
  - ▶ Some languages (like Java) make all methods virtual
- ▶ Virtual method calls introduce overhead
  - ▶ Run time binding
  - ▶ Making always every method virtual is poor design
- ▶ The choice is done at the base class level
  - ▶ If a method is not virtual in the base class, it cannot be made virtual in a derived class

## Abstract Classes (1)

- ▶ It should be evident that classes near to the root of the hierarchy are seldom instantiated
  - ▶ Very general but also very unspecialized
- ▶ Some classes are introduced just to define common behaviors, but are not self sufficient
  - ▶ Think of the class shape in one of the former examples
- ▶ Those classes are useful only for abstraction

## Abstract Classes (2)

- ▶ Abstract classes define a set of methods to be shared by a derived class but are not yet implemented
  - ▶ Implementation will be defined in a derived class
    - ▶ Virtual mechanism plays a fundamental role
- ▶ A pure virtual method is a method declared as:  
`virtual void something() = 0;`
- ▶ A class having one or more pure virtual methods is **abstract**

## Abstract Classes (3)

- ▶ Abstract classes cannot be instantiated
- ▶ Abstract classes can also include non-pure virtual methods
- ▶ Methods and functions can accept pointers to abstract classes
  - ▶ This is their main use: through virtual calls generic code is developed

## Shapes Example Revised

- ▶ In the shape example the shape class has not actually represented a shape (instance), but rather collected some data common to all shapes
- ▶ Therefore, Shape is a good candidate to be an abstract class
  - ▶ `shapesrevised.h`
  - ▶ `shapesrevised.cpp`
  - ▶ `testshapesrevised.cpp`



# Virtual Destructors?

Destructors are almost always `virtual`

- ▶ If you are manipulating objects via pointers to the base class, then the base class should define its destructor as `virtual`
- ▶ Otherwise just the base class destructor is called
- ▶ Recall that destructors are called from bottom to up
- ▶ Destructors can be pure `virtual`
  - ▶ There are some subtle details concerning this aspect (see Eckel's book, chapter 15)

## Virtual Constructors?

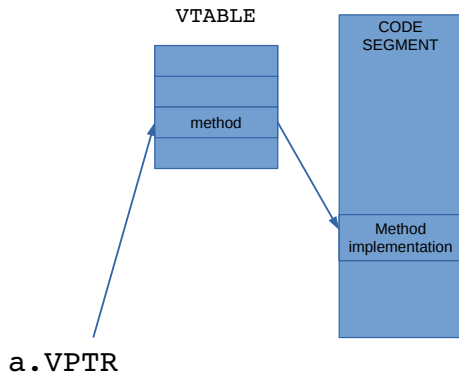
- ▶ You cannot have virtual constructors
  - ▶ Remember that constructors are called from the base to the leaves of the derivation tree
- ▶ Inside a constructor you can call a virtual method, but this will execute the local version
  - ▶ No downsearch is performed, as the assembly of the object is still being performed and elements belonging to derived classes are not guaranteed to be properly initialized

## How Does Polymorphism Work?

- ▶ In order to correctly manage it, it is useful to know how polymorphism is implemented
  - ▶ For every class with virtual methods, a table is created; the table holds the address of the virtual methods (VTABLE)
  - ▶ In addition, a pointer to this table is stored inside the class; this pointer is invisible (VPTR)
  - ▶ When a virtual method is called, the pointer is used to access the table, and then from the table the address of the function is read
  - ▶ As each class with virtual elements is shaped this way, the compiler can insert the code to resolve the calls without knowing the type
  - ▶ The intermediate indirection through VTABLE is the reason for the slower performance of virtual method calls

# Calls Through the Virtual Table

```
1 class Base {
2     public:
3         virtual void
4             method()=0;
5         // VPTR INSERTED
6 };
7 class Derived
8     : public Base {
9     public:
10         void method() { };
11         // VPTR INSERTED
12 };
13 ...
14 Base *a= new Derived;
15 a->method();
```



## Final Exam: Details

- ▶ At the end of the semester, scheduled by the SRO
- ▶ Final tutorial before exam will be offered by the TAs
- ▶ Programming exercises to be solved on paper
  - ▶ You have two hours to solve exercises
  - ▶ Similar to the programming assignments
  - ▶ Practice to write your programs on paper
- ▶ You do not need paper, it will be provided
- ▶ You may not use books or other documentation while taking the exam
- ▶ You may not use mobile phones, calculators or any other electronic devices