CH-230-A

# Programming in C and C++

C/C++

## Lecture 12

Dr. Kinga Lipskoch

Fall 2020

# 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

```
namespace CPPcourse {
  void f1() { ... }
  class class1 { ... };
}
```

- 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

  ```
  using CPPcourse::FirstExample;
  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