(9) C++ Abstractions

Nico Ludwig (@ersatzteilchen)

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

TOC

- (9) C++ Abstractions
  - Compile Time and Link Time Dependencies with UDTs
  - Modularization
  - Using Types in Size and using Types in Name
  - How to reduce Compile Time Dependencies
  - Internal and external Linkage
  - Name Mangling and Application Binary Interfaces (ABIs)
  - Static and dynamic linked Libraries

- Sources:
  - Bjarne Stroustrup, The C++ Programming Language
  - John Lakos, Large Scale C++ Software Design

Regulation of Modularization is required

- The C/C++ family of programming languages doesn't regulate the module system.
  - It's based on h-files (mind the standard library), but that's all we know!
- We have to define how we're going exploit h-files to get a useful module system:
  - John Lakos defines a set of rules in his book "Large Scale C++ Software Design".
  - He describes and proves how large modular software can be created in C++.
- Required knowledge to understand this lecture:
  - Preprocessor
  - #include guards
- Let's define our first rules:
  - R1: Each type is defined in a dedicated h-file.
  - R2: Each h-file is adorned with #include guards.

- We're discussing physical, not logical dependencies now! Alas we have to discuss this, because logical dependencies influence physical dependencies, esp. in C++.

A naïve Module

- To understand C++-modularization we've to discuss compilation and linking.
  - Let's analyze a naïvely implemented type *Car* in Car.h, obeying R1 and R2:

```
// Main.cpp
#include "Car.h"

void main() {
        Car car;
        car.StartEngine();
}
```

```
// Car.h
#ifndef CAR_H_INCLUDED
#define CAR_H_INCLUDED

#include "Engine.h"
#include "Tyre.h"

class Car {
        Engine* theEngine;
        Tyre spareTyre;
public:
        void StartEngine() { theEngine->Start(); }
        void SetSpareTyre(Tyre spareTyre) {
            this->spareTyre = spareTyre;
        }
        Tyre GetSpareTyre() { return spareTyre; }
};

#endif
```

```
// Engine.h
#ifndef ENGINE_H_INCLUDED
#define ENGINE_H_INCLUDED

class Engine {
public:
        void Start() {}
};

#endif
```

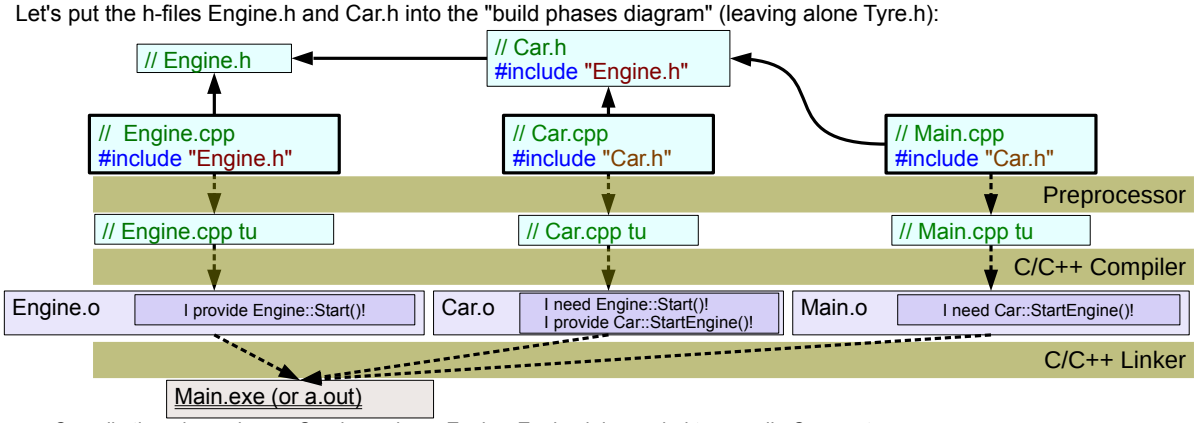
```
// Tyre.h
#ifndef TYRE_H_INCLUDED
#define TYRE_H_INCLUDED

class Tyre {
};

#endif
```

Review of the Build Phases of that Module

- Let's put the h-files Engine.h and Car.h into the "build phases diagram" (leaving alone Tyre.h):



  - Compile time dependency: *Car* depends on *Engine*, Engine.h is needed to compile Car.cpp tu.
  - Link time dependency: *Car* depends on *Engine*, Car.o uses undefined symbols (*Engine::Start()*) that get satisfied by Engine.o.
  - A compile time dependency almost always implies a link time dependency!

Compilation: The Usage Perspective of UDTs

- Next, let's analyze, how the type *Car* is composed:
  - There are two fields: an *Engine** and a *Tyre*.
  - There are three member functions:
    - *Car::SetSpareTyre()* and *Car::GetSpareTyre()*, which accept and return *Tyre* objects and
    - *Car::StartEngine()*, which neither accepts nor returns objects.

- Then let's analyze how the UDT *Car* uses the types *Engine* and *Tyre*:
  - *Car* has fields of type *Engine** and *Tyre*.

```
class Car { // (members hidden)
      Engine* theEngine;
      Tyre spareTyre;
      /* pass */
};
```

```
class Car { // (members hidden)
public:
      void StartEngine() { theEngine->Start(); }
      void SetSpareTyre(Tyre spareTyre) {
            this->spareTyre = spareTyre;
      }
      Tyre GetSpareTyre() { return spareTyre; }
};
```

  - *Engine*'s member function *Start()* is called.
  - The cctor and copy-assignment of *Tyre* are used.

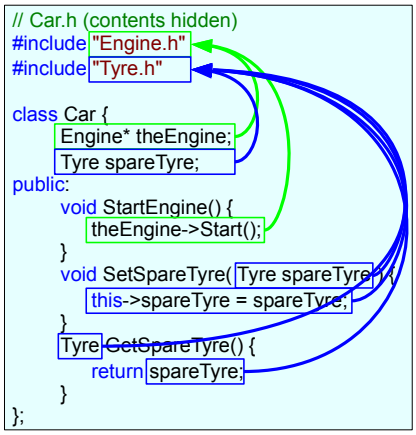
Compilation: The Usage Perspective of h-Files – Using Types in Size

- How does the compiler "see" types?
  - We're discussing physical, not logical dependencies now!
  - And we've to change the perspective: how does Car.h use the contributed types?

- Car.h "needs to see" the type definitions of *Engine* and *Tyre* in order to compile.
  - This is because members of these types are used in Car.h!

- In C++ this dependency is called "using in size":
  Car.h uses *Engine* and *Tyre* in size.

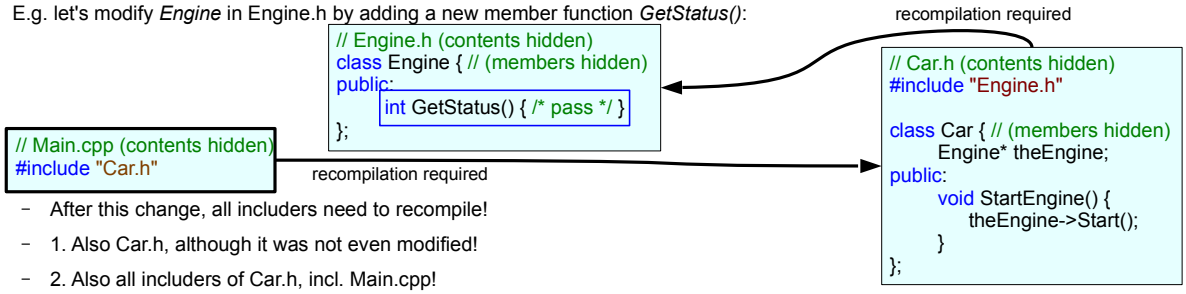
- A code file that uses types in size has to #include the definitions of these types directly or indirectly.



- What does that mean "a type definition has to be #included directly or indirectly"?

Compilation: Compile Time Dependencies

- If an h-file needs to see types in size, including "their" h-files is needed after R1.
- There's a problem: if an #included h-file changes, all includers need recompilation!
  - This is also true, if the including files don't even make use of these changes!
- E.g. let's modify *Engine* in Engine.h by adding a new member function *GetStatus()*:



  - After this change, all includers need to recompile!
  - 1. Also Car.h, although it was not even modified!
  - 2. Also all includers of Car.h, incl. Main.cpp!
  - => This is because Engine.h and Car.h have changed. (Of course the executable needs to be relinked!)

- If only a comment was changed in an h-file, all includers would have to recompile as well!

Resolving Compile Time Dependencies: Forward Declarations

- To "fix" the "use in size"-dependency we'll move the usage of a type into a c-file.
  - 1. Moving the implementation of *Car::StartEngine()* into Car.cpp (non-inline):

```
// Car.h (contents hidden)
#include "Engine.h"

class Car { // (members hidden)
        Engine* theEngine;
public:
        void StartEngine();
};
```

```
// Car.cpp (contents hidden)
#include "Car.h"

void Car::StartEngine() {
        theEngine->Start();
}
```

  - 2. In Car.h we no longer have to #include Engine.h. We only forward declare *Engine*!
  - 3. But then we have to #include Engine.h into Car.cpp, where it is used in size now.

```
// Car.h (contents hidden)
class Engine; // Forward declaration of Engine.

class Car { // (members hidden)
        Engine* theEngine; // Engine used in name.
public:
        void StartEngine();
};
```

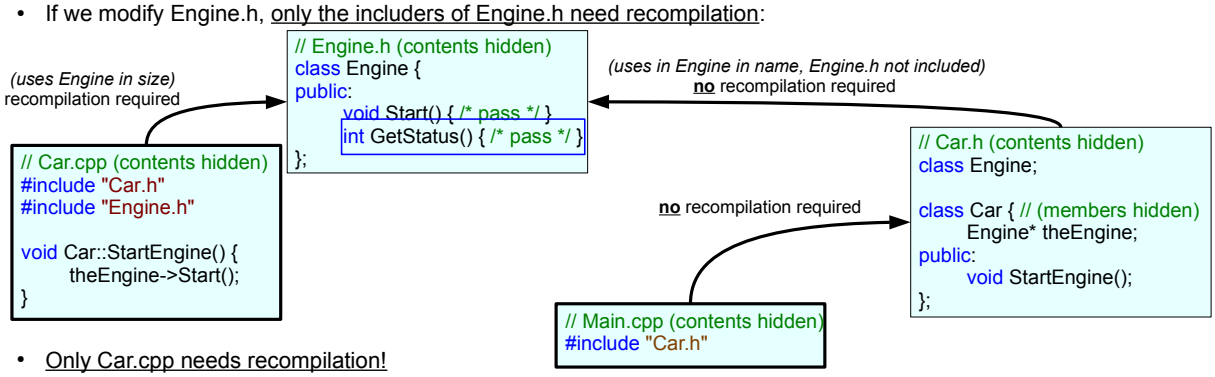
```
// Car.cpp (contents hidden)
#include "Car.h"
#include "Engine.h"

void Car::StartEngine() {
        theEngine->Start(); // Engine used in size
}
```


  - => Car.h does now use *Engine* in name and Car.cpp uses *Engine* in size!

Resolving Compile Time Dependencies: It works!

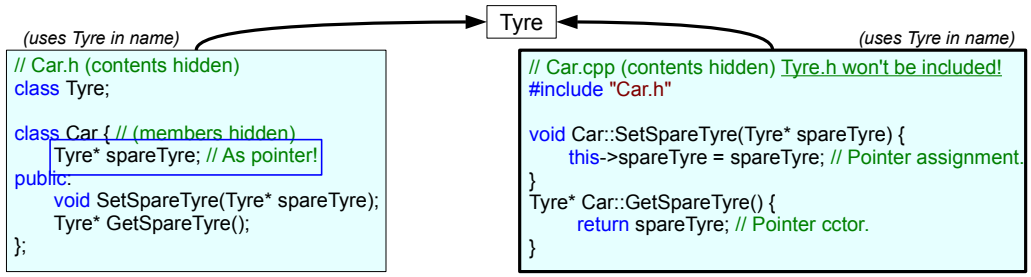
- If we modify Engine.h, only the includers of Engine.h need recompilation:



- Only Car.cpp needs recompilation!
  - Car.cpp uses *Engine* in size and needs to "see" *Engine*'s definition (In a modified tu!).
  - Car.h uses *Engine* in name, it only has to "know", that there'll exist the type *Engine*.
  - Of course the executable needs to be relinked!

What's behind "using Types in Size"?

- With this new tool (forward declaration), we'll change Car.h to use *Tyre* in name as well:



- Now we can better understand the term "use in size":
  - In former versions of *Car*, the sizes of *Engine* and *Tyre* needed to be known to make this work:

```
std::size_t formerCarsSize = sizeof(Car);
```

  - *formerCarsSize* will be composed of the sizes of all full-object-fields of *Car*!

```
std::size_t formerCarsSize = sizeof(Engine*) + sizeof(Tyre);
```

  - If *Car* is composed of pointer-fields, the fields' sizes are the same (e.g. 4 == sizeof(void*))!

```
std::size_t carsSize = 4 + /* = sizeof(Engine*) */ + 4 /* = sizeof(Tyre*) */; // 32b-system
```

Using Types in Size and in Name: Edge Cases

- Parameters and return types are never used in size.
  - If we retain the non-pointer *Tyre* parameters, a forward declaration is sufficient:

```
// Car.h (contents hidden)
class Tyre;

class Car { // (members hidden)
      Tyre* spareTyre; // Not used in size!
public:
      // Both functions use Tyre in name only!
      void SetSpareTyre(Tyre spareTyre);
      Tyre GetSpareTyre();
};
```

  - => Using types in parameters creates only a "using in name"-dependency.
  - On the declaration of functions the compiler doesn't need to know the type sizes!
  - The same is valid for return-types.

- STL types (e.g. *std::string*) are not allowed to be forward declared!
  - It's possible on many compilers, but it's not portable.

Summary

- Modularization is required to have reusable code, esp. UDTs.
  - But modularization introduces physical dependencies among the modules.
  - Regarding "using in name"- and "using in size"-dependencies helps controlling and reducing physical dependencies.
  - => C++ programmers have to understand these aspects.
- Compile time dependencies are independent of UDT accessors in C++ (private etc.)!
- Types are used in name
  - if they are used as pointer or reference types in fields of a UDT and
  - if they are "mentioned" in function declarations (return and/or parameter).
- Types are used in size
  - if they are used as full-objects in fields or as base type of a UDT or
  - if their interface is used in an implementation (in function code, esp. inline code).
- Mind and exploit the discussed edge cases (parameter and return types).

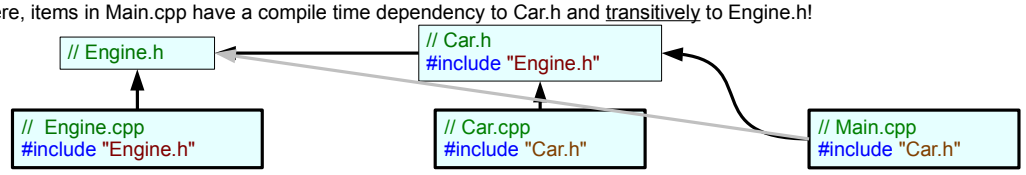
- References can be impractical as fields, as they need to be initialized in ctors.

Finally let's complete our Rules for C++ Modularization

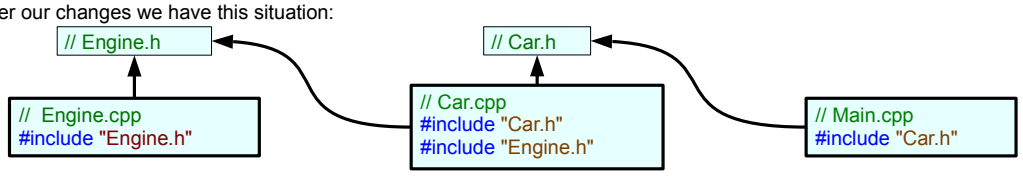
- Believe in these simple rules! Don't be too clever!
- R1: Each type is defined in a dedicated h-file.
- R2: Each h-file is adorned with #include guards.
- R3: Employ strict separated compilation.
  - Use h-files and c-files and avoid inline member functions!
  - A c-file #includes "its" h-files as 1st #include!
- R4: Strive to using types in name only.
  - Avoid using full objects as fields; prefer pointers (and references (secondary)).
- R5: Only use UDTs, void*, char*, const char*, bool, int and double in interfaces!
  - R5a: UDTs should be used as (const) references or (const) pointers preferably.
  - Other types should only be used with a good reason! Here some more rules:
    - R5b: Avoid using float, char, short or long! Don't blindly believe in what teachers tell you, those types are exotic in professional C++ programming!
    - R5c: Never use unsigned types!

Ok – But what have we archived?

- We learned that a compile time dependency is a dependency an item *A* has to an h-file!
  - Here, items in Main.cpp have a compile time dependency to Car.h and transitively to Engine.h!



- When we use less types in size in h-files, less #includes are required.
  - After our changes we have this situation:



  - Using types in name moved the compile time dependency from Main.cpp to Engine.h here!
  - Using types in name is the way to reduce compile time dependencies and compilation time!

- The physical reusability of Car.h and the logical reusability of *Car* has been improved enormously!

Reducing Dependencies means reducing Costs!

- Esp. in large C++ projects, compile time dependencies should be minimized!
  - Changes in h-files have a bigger impact to compilation than changes in c-files!
  - The compile time costs can be reduced.
  - The risk of compile time errors can also be reduced.

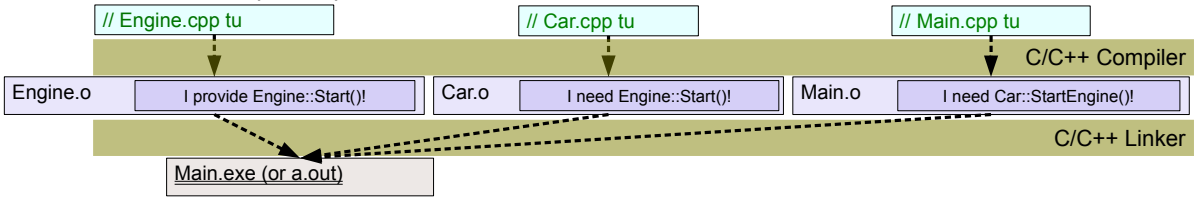
- Reduced dependencies leads to improved reusability, it makes modules/UDTs versatile, valuable and user-friendly.
  - The documentation of such modules/UDTs is usually "a piece of cake"!

- The difficulty to find errors in a C/C++ program increases in each build phase:
  - Phase 1: preprocessing, phase 2: compile time and phase 3: link time.
  - After each phase we've a greater "logical distance" to the original source code.
  - (During run time the distance to the original source code reaches its maximum.)

- With this perspective we can state that compile time errors are significantly easier to find that link time errors!
  - After we discussed compile time dependencies, let's discuss link time dependencies. – Then we can better understand and fix link time errors.

Linking: How does the Linker "link"?

- Symbols defined in a tu (e.g. *Engine::Start()*) can be used in other tus (e.g. Car.cpp's tu).
- After compilation: symbols defined in an o-file can be used in another o-file.
  - The connection is finally done by the linker. Let's have another look:



- When we talk about symbols we should ask: How does the linker "link"?
  - The linker knows nothing about code, it knows about external and exported symbols!
  - But which symbols of a tu are finally exported by the resulting o-file?
- Now we are going to discuss and understand linkage.

Linking: How are Symbols resolved during Link Time?

- Each o-file has a table of the symbols it exports:

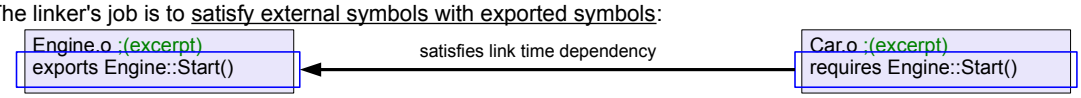
```
Engine.o ;(excerpt)
exports Engine::Start()
```

```
Car.o ;(excerpt)
exports Car::StartEngine()
exports Car::GetSpareTyre()
```

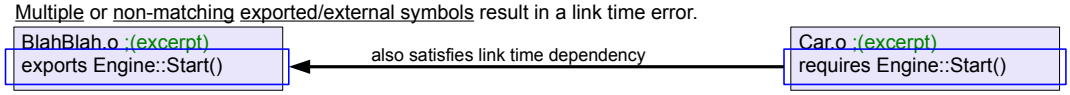
- Additionally do o-files record, which external symbols they need:

```
Car.o ;(excerpt)
requires Engine::Start()
```

- The linker's job is to satisfy external symbols with exported symbols:

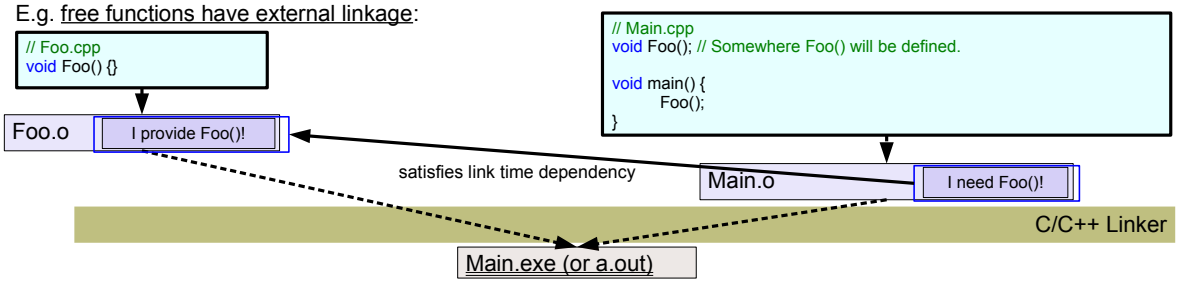


- The linker only processes a list of o-files!
  - It is not interested in the names of the files, it only tries to satisfy link time dependencies.
  - Multiple or non-matching exported/external symbols result in a link time error.



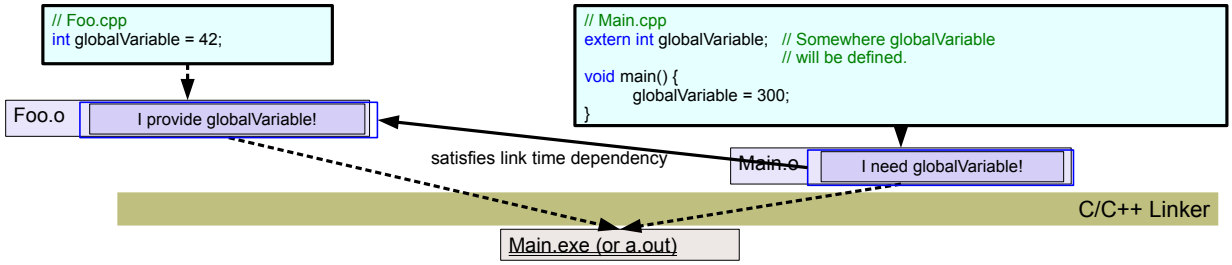
Linking: Finally – What's linkage?

- Linkage describes the visibility of symbols of an o-file to the linker.
- Important about symbols is that not all of them are being exported!
  - There exist symbols that have external linkage.
  - There exist symbols that have internal linkage.
  - There exist symbols that have no linkage at all.
- E.g. free functions have external linkage:

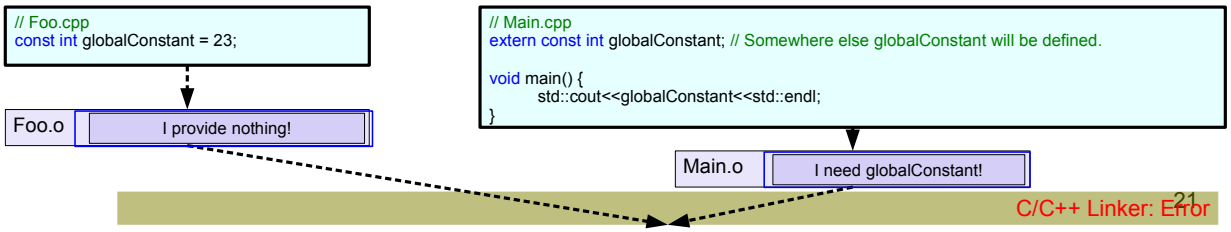


Linking: Example of Linkage

- Global/free variables do also have external linkage:
  - Notice the extern keyword to notate the variable declaration of *globalVariable*.



- Global/free constants have internal linkage:



C++ Symbols with external, internal and no Linkage

- External linkage:
  - Free functions.
  - Free variables.
  - Member functions (static and non-static).
  - (The linker can remove unused symbols with external linkage to minify executables.)

- Internal linkage:
  - Free constants (by default).
  - static free functions (deprecated in C++).
  - inline free and member functions.
  - (Items in anonymous namespaces.)

- No linkage:
  - UDTs
  - Preprocessor symbols

Name Mangling

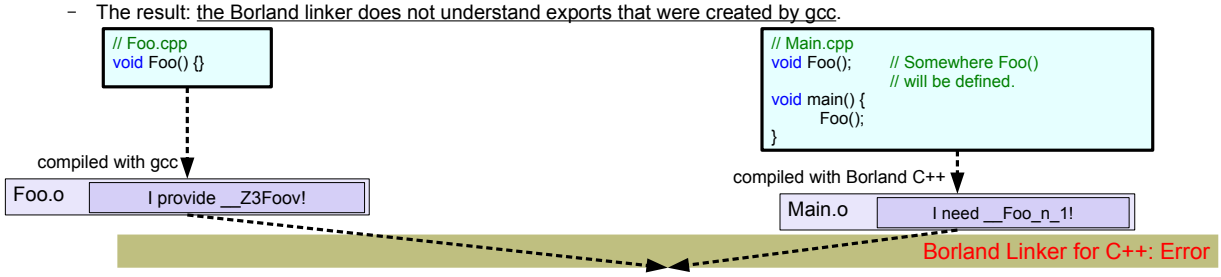
- In C++, functions can have overloads. How does it influence exported symbols?
  - The solution is simple: all exported symbols need to be unique for the linker!
  - The compiler automatically adorns exported symbols w/ prefixes and suffixes to make overloads of functions distinguishable for the linker.
  - This name adorning is called name mangling.
- E.g. gcc produces following mangled names for *Foo()* and its overload *Foo(int)*:

- Name mangling results in the Application Binary Interface (ABI) of an o-file.
- The ABI is generally influenced by
  - function's interfaces: name mangling, calling convention, exception declaration,
  - UDT's interfaces: padding and the layout of virtual function tables and
  - the platform (x86, x64, ARM etc.).

- An o-file demanding a prototype that is different from provided external symbol won't link. Name mangling demands not only the name but also the argument list to match exactly.
  - This is different in C, where calling a function from a different prototype is a disaster as the partial sf would not have the layout expected by the implementation.
- ARM: the architecture developed by ARM plc (earlier known as "Advanced RISC Machines Ldt").

Name Mangling and Link Time Compatibility

- Problem: different compilers of different vendors create different ABIs!
  - E.g. the name mangling of the Borland compiler is different from gcc's.
  - The result: the Borland linker does not understand exports that were created by gcc.



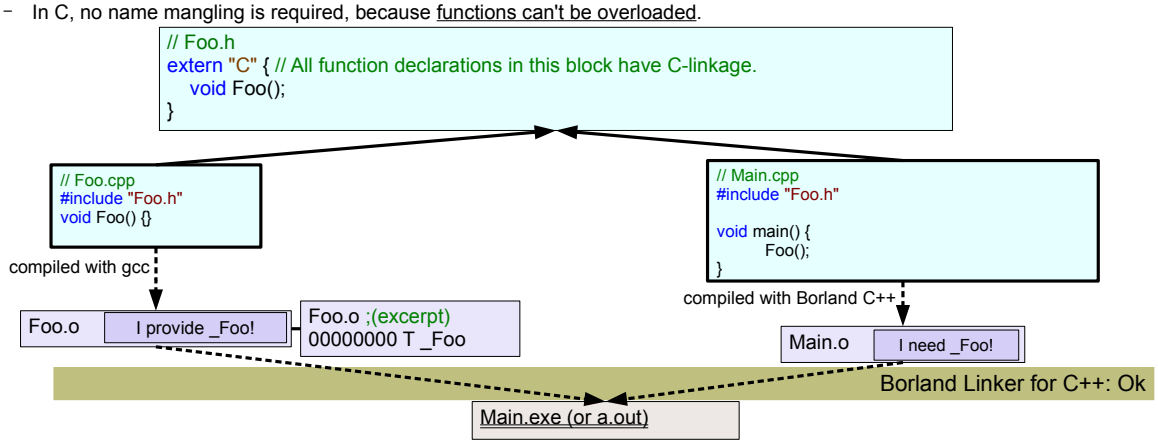
  - Linker error message: "unresolved symbol *Foo()*"

- C++ does not standardize name mangling and ABIs.
  - Technologies like the Component Object Model (COM) try to get rid of differing ABIs by standardization. Also between different ABIs of different programming languages!

- COM: the calling conventions have been normalized to __stdcall, declared exceptions as well as function overloads are not allowed.

The C-Linkage Specification

- With a C-linkage specification name mangling can be disabled.
  - In C, no name mangling is required, because functions can't be overloaded.



- The C standard requires functions' export symbols to have a _ as prefix ...
  - … and that's all to the ABI! – Symbols with that "format" are accepted by each linker!

- So, the ABI of C's o-files is predictable!
- Maybe it is required to add a specific calling convention that was "agreed upon" between the provider and consumer of the o-files.

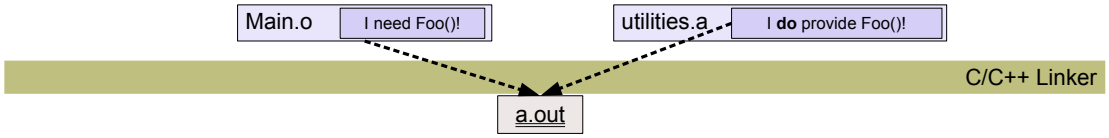
Static and dynamic linked Libraries

- Often o-files are linked together to library files (libraries).
  - These libraries are then linked with our o-files to get the resulting executable.
- Static (linked) libraries: Libraries can be linked to the executable at link time.
  - Typically these are .lib-files or .a-files.
  - The usage of static libraries enlarges the size of the resulting executable file!
    - Because the libraries are completely linked into the executable file.
- Dynamic (linked) libraries: Libraries can be linked to the executable at run time.
  - Dynamic libraries are sometimes called "shared libraries".
  - Typically these are .dll-files, .so-files or .dylib-files
  - On link time a stub file is needed to satisfy symbols that get loaded during run time.
  - Using dynamic libraries minimizes the size of resulting executable file!
    - Because the libraries are not completely linked into the executable file.
  - But the executable can only run, if the dynamic libraries can be found during run time.

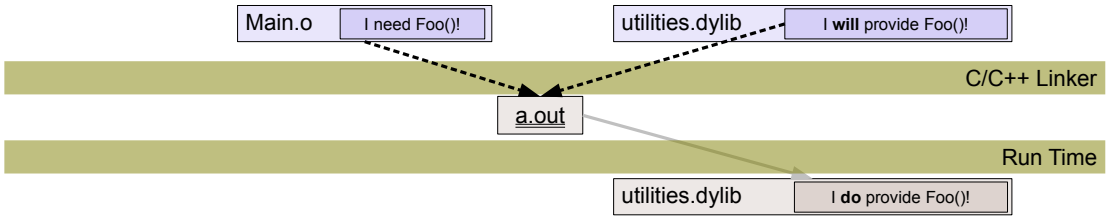
- The "so" in the extension of .so-files stands for "shared objects".

Static and dynamic linked Libraries in Action

- Using the static linked library "utilities.a":



- Using the dynamic linked library "utilities.dylib":



Thank you!