Effective C++ Programming

NIE-EPC (v. 2021):

TRANSLATION UNITS AND OBSERVABLE BEHAVIOR, ODR, INTERNAL LINKAGE, INLINE, STATIC AND DYNAMIC LINKING, PIMPL

Translation units + observable behavior

- Is observable behavior related to translation units?
- **Yes** — observable behavior of some code = its observable behavior from the perspective of the translation unit where the code is.
- *Recapitulation:* translation units for `add.cpp` and `main.cpp`:
- Observable behavior of function add from the perspective of its translation unit:
  - returns the sum of its parameters.
- Observable behavior of function `main` from the perspective of its translation unit:
  - calls add with arguments 1 and 2, adds the returned value with 3, and returns the result.
- The generated machine code in object files `add.o` and `main.o` matches to this behavior.
- The linker only merges this machine code into a single executable binary file.

```
int add(int a, int b) {
  return a + b;
}
```

```
int add(int, int);

int main() {
  return add(1, 2) + 3;
}
```

```
main:
  sub  rsp, 8
  mov  esi, 2
  mov  edi, 1
  call add(int, int)
  add  rsp, 8
  add  eax, 3
  ret
```

```
add(int, int):
  lea eax, [rdi+rsi]
  ret
```

Translation units + observable behavior *(cont.)*

- *Alternative option:* what if we put all the code into a single translation unit? Source file (*left*), translation unit (*right*):

```
// main.cpp source file ver.3
int add(int a, int b) {
  return a + b;
}
int main() {
  return add(1, 2) + 3;
}
```

```
int add(int a, int b) {
  return a + b;
}
int main() {
  return add(1, 2) + 3;
}
```

- Does anything change?
- Observable behavior of `main` with respect to this translation unit — returning the value 6.
- Generated machine code with `g++ -O2`:

```
main:
  mov eax, 6
  ret
```

Translation units + observable behavior *(cont.)*

- In both cases, the very same source code:

```
// main.cpp source file ver.2
int add(int, int);
int main() { return add(1, 2) + 3; }
```

```
// add.cpp
int add(int a, int b)
{ return a + b; }
```


```
// main.cpp source file ver.3
int add(int a, int b) {
  return a + b;
}
int main() {
  return add(1, 2) + 3;
}
```

- In both cases, compiled with optimizations:

```
$ g++ -O2 -c add.cpp
$ g++ -O2 -c main.cpp
$ g++ main.o add.o
```

```
$ g++ -O2 -c main.cpp
$ g++ main.o
```

- In both cases, the same program runtime behavior:

```
$ ./a.out
$ echo $?
6
```

```
$ ./a.out
$ echo $?
6
```

- In both cases, the very different runtime (and memory) efficiency:

```
main:
  sub  rsp, 8
  mov  esi, 2
  mov  edi, 1
  call add(int, int)
  add  rsp, 8
  add  eax, 3
  ret
add(int, int):
  lea eax, [rdi+rsi]
  ret
```


```
main:
  mov eax, 6
  ret
```

Translation units and efficiency

- Should we put definitions of all the functions called in some translation unit into that translation unit? **NO!**
- However, it may be profitable for "short" functions (w.r.t. their runtime) that are called frequently (at runtime).
- *Example: fused vector multiply-add* (FMADD) operation defined in terms of *scalar FMADD*:

```
void scalar_fmadd( float& x, float y, float alpha ) {
  x += y * alpha;
}
```

```
void vector_fmadd( std::vector<float>& x, const std::vector<float>& y, float alpha ) {
  for (size_t i = 0; i < x.size(); i++)
    scalar_fmadd(x[i], y[i], alpha);
}
```

- *Experiment:* `scalar_fmadd` defined in (1) the same / (2) a different translation unit as/than `vector_fmadd`.
- *Observation:* in (1) the operation was **2.2× faster** than in (2).
- *Setup:* GCC, enabled optimizations, Intel Core i9.

Matrix benchmark revisited

- *Original experiment:* the benchmark function was defined in the same translation unit where it was called.
- Putting benchmark function into a separate translation unit…

```
// benchmark.cpp

void benchmark(const double A[][32], const double B[][32], double C[][32]) {

  for (int n = 0; n < 1'000'000; n++)
    for (int i = 0; i < 32; i++)
      for (int j = 0; j < 32; j++)
        for (int k = 0; k < 32; k++)
          C[i][j] += A[i][k] * B[k][j];
}
```

- …and passing arrays (matrices) as arguments assure that the floating-point operations will be performed at runtime.
- Their effect is observable from outside of this translation unit (by reading elements of the array passed as an argument for C).
- *Measured performance on Surface:* 0.79 GFlop/s.
- *Fugaku:* 0.442 EFlop/s ⇒ 559 million times more capable.

Function *inlining*

- Resolution of observable behavior across function boundaries is an optimization technique called *inlining*.
- **Inlining** treats a function as a "macro" that is expanded at the place where the function is called.

```
int add(int a, int b) {
  return a + b;
}
int main() {
  return add(1, 2) + 3;
}
```

=

```
int main() {
  int a = 1;
  int b = 2;
  return (a + b) + 3;
}
```

=

```
main:
  mov eax, 6
  ret
```

- Some implementations allow to selectively control inlining.
- Disabling inlining restricts the resolution of observable behavior across function boundaries ⇒ each function call statement generates a corresponding `call` instruction.
- *Example:* compilation with `g++ -O2 -fno-inline` →

```
main:
  sub  rsp, 8
  mov  esi, 2
  mov  edi, 1
  call add(int, int)
  add  rsp, 8
  add  eax, 3
  ret
add(int, int):
  lea eax, [rdi+rsi]
  ret
```

- *Note:* no relation with the `inline` keyword.
- *Experiment:* `std::sort` 2.7× slower with disabled inlining.

Translation units *vs* source code files

- If a function needs to be called in some translation unit, its declaration must be in the **same** translation unit as well (*or, its definition, which is also a declaration*).
- *Implication?* Does it mean that we need to put this declaration into all **source code files**, where the function is called? **No.**
- *Recall:* translation unit = preprocessed source code file.
- One of preprocessor abilities: **inclusion of some source code file into another source code file**.
- *Preprocessor directive:* `#include`:
  - This directive replaces itself with the content of the included file.
  - It does that recursively (included file may also contain `#include` directives).
- *Example:* 2 source code files + translation unit for `main.cpp` (*right*):

```
// add.cpp

int add(int a, int b) {
  return a + b;
}
```

```
// main.cpp

#include "add.cpp"

int main() {
  return add(1, 2) + 3;
}
```

```
int add(int a, int b) {
  return a + b;
}

int main() {
  return add(1, 2) + 3;
}
```

Translation units *vs* source code files *(cont.)*

- *Convention:*
  - Files to be included into other files are typically called **„header files"** (shortly headers) and have file extensions as `.h`, `.hpp`, `.hxx`, `.hh`, *no one*.
  - Files to be transformed into translation units are typically called **„source files"** and have file extensions as `.cpp`, `.cxx`, `.cc`, etc.
- This is just a convention; compilers and preprocessors do not care about how we call files or their extensions.
- Typical intent of headers: API.

```cpp
// add.h
int add(int a, int b);
```

```cpp
// add.cpp
int add(int a, int b) {
  return a + b;
}
```

```cpp
// main.cpp
#include "add.h"
int main() {
  return add(1, 2) + 3;
}
```

```cpp
int add(int a, int b);
int main() {
  return add(1, 2) + 3;
}
```

```cpp
int add(int a, int b) {
  return a + b;
}
```

Translation units *vs* source code files *(cont.)*

- Some “piece of functionality” — interface (API) + implementation.
- *Ideal world:*
  - API in a header file (header files),
  - implementation in a source file (source files).
- Advantage of such separation:
  - With dynamically-linked libraries (`.so`, `.dll`), changing implementation details does not require recompilation of programs’ source code.
- *Reality:*
  - Implementation details are frequently in header files.
- *Examples:*
  - *Inlining* – function bodies (implementation) must be visible to enable their iniling.
  - *Class non-public members* — must be at least declared at the place where the class is defined.
  - *Templates* — definitions must be visible to enable their instantiation.
  - *Others* — functions with return type deduction, etc.

ODR

- *Example:* function definition in a header file to enable its inlining:

```
// add.h
int add(int a, int b) { return a + b; }
```

- *Example (cont.):* multiple source files that call this function:

```
// sub.cpp
#include "add.h"
int sub(int a, int b) {
  return add(a, -b);
}
```

```
// main.cpp
#include "add.h"
int main() {
  return add(1, 2) + 3;
}
```

- *Example (cont.):* compilation and linkage...

```
$ g++ -O2 -c sub.cpp
$ g++ -O2 -c main.cpp
$ g++ main.o sub.o
```

- ...resulted in the following **linker** error:

```
multiple definition of `add(int, int)'
```

ODR *(cont.)*

- Translation units for `sub.cpp` (*left*) and `main.cpp` (*right*):

```
int add(int a, int b) {
  return a + b;
}

int sub(int a, int b) {
  return add(a, -b);
}
```

```
int add(int a, int b) {
  return a + b;
}

int main() {
  return add(1, 2) + 3;
}
```

- *Problem:*
  - *C++ rule:* an entity (function, variable,…) generally cannot be defined in multiple translation units.
  - It can be defined only once — “one definition rule” (ODR).
  - *Exceptions/workarounds:* `static`, anonymous namespace, `inline`, templates.
- *Simplified explanation:*
  - Function add is defined in some translation unit and may be used in other translation units as well → it has so-called “*external linkage*”.
  - Linker “sees” multiple machine code of add — in `sub.o` and `main.o`.
  - It cannot decide to which one it should link the add calls.

Specifier `static`

- How to enable to define a function inside a header file (for example, for inlining purposes) without breaking ODR?
- *First option:* making this function `static`.

```
// add.h
static int add(int a, int b) { return a + b; }
```

- Translation units:

```
static int add(int a, int b) {
  return a + b;
}
int sub(int a, int b) {
  return add(a, -b);
}
```

```
static int add(int a, int b) {
  return a + b;
}
int main() {
  return add(1, 2) + 3;
}
```

- Functions add now have "*internal linkage*" — the definition in one translation unit is "private" to this unit ⇒ no ambiguity for a linker.
- These two definitions define two different functions (separate entities) regardless of their equivalent forms ⇒ no violation of ODR.
- Without `static` they would define a single function (entity).

Anonymous namespaces

- *Alternative option:* putting add into anonymous namespace:

```
// add.h

namespace {
  int add(int a, int b) { return a + b; }
}
```

- Translation units:

```
namespace {
  int add(int a, int b) {
    return a + b;
  }
}

int sub(int a, int b) {
  return add(a, -b);
}
```

```
namespace {
  int add(int a, int b) {
    return a + b;
  }
}

int main() {
  return add(1, 2) + 3;
}
```

- *Our case:* the same effect — add now have internal linkage.
- Anonymous namespace *vs* `static`:
  - Generally, anonymous namespaces are more “powerful”.
  - *For example*, we can put type definitions into anonymous namespace but cannot make types `static`.
  - Also, `static` has multiple meanings in different contexts, which might be confusing.

Confusion with static: example

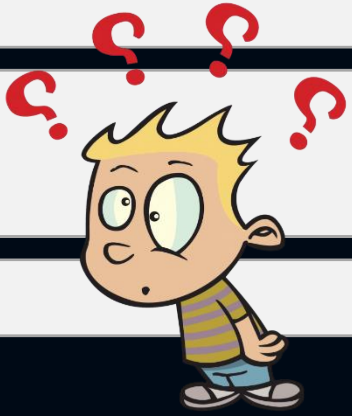
- *Example* of confusedness of `static`:

```
// X.h
class X {
  static void f();
};
void X::f() { }
```

```
// main.cpp
#include "X.h"
int main() { }
```

```
// other.cpp
#include "X.h"
```

```
$ g++ -O2 -c other.cpp
$ g++ -O2 -c main.cpp
$ g++ main.o other.o
```



```
multiple definition of `X::f()'
```

- Static member function `f` has external linkage
  ⇒ violation of ODR.
- Here, `static` indicates that `f` is related to class `X`, not to its instances.

Specifier `inline`

- *Another option:* `inline` functions.

```
// add.h
inline int add(int a, int b) { return a + b; }
```

- Translation units:

```
inline int add(int a, int b) {
  return a + b;
}
int sub(int a, int b) {
  return add(a, -b);
}
```

```
inline int add(int a, int b) {
  return a + b;
}
int main() {
  return add(1, 2) + 3;
}
```

- Both definitions of add define a single function (entity).
- With `inline`, they do not break ODR — exception.
- Generally, definitions of the same `inline` definition in multiple translation units define a single entity ⇒ all these definitions must be identical.
- An inline function must be defined in each translation unit where it is used (called).

Internal linkage *vs* `inline`

1. **Internal linkage** (`static`, anonymous namespace):
   - Entities "private" for a translation unit.
   - May have different definitions in different translation units.
   - Usually defined in **source files**.
   - *Typical use case:* auxiliary entities private for some source file for which we don't want to care about breaking ODR.
2. **`inline`**
   - Reusable entities ⇒ usually defined in **header files**.
   - *Typical use case:* functions for which we want to enable their inlining.
   - *Note:* `inline` itself does not enable or even force inlining; it just helps avoiding ODR violation.

Member functions and `inline`

- Class non-static member functions are:
  - implicitly inline if they are defined within the class definition (body),
  - implicitly not-inline if they are defined outside of the class definition.
- *Example:*

```cpp
// Y.h ver. 1
class Y {
  void f() { }  // implicit inline
};
```

```cpp
// Y.h ver. 2
class Y {
  void f();
};
void X::f() { }  // not inline
```

```cpp
// main.cpp
#include "Y.h"
int main() { }
```

```cpp
// other.cpp
#include "Y.h"
```

```
$ g++ -O2 -c other.cpp
$ g++ -O2 -c main.cpp
$ g++ main.o other.o
```

```
multiple definition of `Y::f()'
```

Libraries

- Library — reusable coded functionality.
- *Example:* Library that contains a single function for adding two unsigned integer numbers:

```
unsigned add(unsigned a, unsigned b) {
  return a + b;
}
```

- In which form can this library be implemented?
- First option — putting the function(ality) into a library header file ⇒ “header-only library”:

```
// unsigned_add.h
#ifndef UNSIGNED_ADD_H
#define UNSIGNED_ADD_H
inline unsigned add(unsigned a, unsigned b) { return a + b; }
#endif
```

Header-only libraries

- Advantages:
  - Functions may be inlined ⇒ potentially higher performance.
  - Using of this library involves only inclusion of the header file(s).
  - No linking is involved.
  - Library users (developers of programs built upon this library) need only the header file(s) ⇒ no machine code (object files) are required.
  - Programs users do not need anything regarding the library (do not need to know about its existence at all).
- Disadvantages:
  - Library source code is “copy-pasted” into the programs source code.
  - ⇒ Machine code related to the library functionality is in the program binary file.
  - ⇒ When the implementation of the library functionality is updated and we want to get this updated functionality into programs, these need to be rebuilt (recompiled and relinked).

Header-only libraries (*cont.*)

- Real-world scenario:
  1. Someone develops a library; role — *library developer*.
  2. Someone else develops a program by using this library; role — *program developer*.
  3. The program is installed on computer systems and run; role — *program user*.
  4. Library developer updates the library code (*bug fixes, improved performance,…*).
  5. Program users want this updated library functionality in their programs.
- Library form — header only:
  - ⇒ Program needs to be rebuilt.
  - If the program source code is available, this can be done by program users.
  - Otherwise, it **needs to be done by program developer**.

Binary libraries

- Another option is to split the interface and implementation details between header and source files:

```
// unsigned_add.h

#ifndef UNSIGNED_ADD_H
#define UNSIGNED_ADD_H

unsigned add(unsigned, unsigned);  // declaration only

#endif
```

```
// unsigned_add.cpp

unsigned add(unsigned a, unsigned b) { return a + b; }  // definition
```

- Developers of programs using this library now do not need the library source code file which implements its functionality.
- Instead, they need only:
  - the header file (to be able to compile the program code that calls the library function),
  - the object file with the machine code of the function (to be able to link the final executable program together).

Binary libraries (*cont.*)

- The machine code of the program contains call instructions for the library function.
- These calls needs to be linked with the machine code of the function — the library machine code.
- **When** this linking happens?
- There are two options:
  1. The linking happens when the program binary executable file is built.
  2. The linking happens not until the program is executed.
- Ad 1. This approach is called "static linking" and the libraries used this way are called "statically linked libraries".
- Ad 2. This approach is called "dynamic linking" and the libraries used this way are called "dynamically linked libraries".
- In both cases, library machine code (object files) is "packed/ archived" into special file formats:
  - Ad 1. `.a` or `.lib` extensions (Windows / Linux),
  - Ad 2. `.dll` / `.so` extensions (ditto).

Statically linked libraries

- Linking happens when the program is built such that both:
  - the machine code of the program,
  - the machine code of the library,
- is together "copied" into the final program executable file.
- ⇒ The library machine code is "hard-wired" into the program.
- Advantages:
  - Program users do not need to care about the library at all; they do not need to know about its existence.
- Disadvantages:
  - Reflection of library functionality updates into programs still **requires their rebuilding** (namely, only relinking is required without program source code recompilation).
  - Program users still need to update program binary files to reflect the library updates.

Statically linked libraries (*cont.*)

- *Example:* building a static version of the libunsigned library:

```
// unsigned_add.cpp
unsigned add(unsigned a, unsigned b) { return a + b; }
```

```
$ g++ -O2 -c unsigned_add.cpp
$ ar rcs libunsigned.a unsigned_add.o
```

- The ar tool “archives” object files into a single library file.
- Program developers now need the library header file for program compilation…

```
// main.cpp
#include <iostream>
#include <unsigned_add.h>
int main() { std::cout << add(1u, 2u) << std::endl; }
```

- …and the library binary file for program linking:

```
$ g++ -O2 -c -I"/path/to/unsigned_add.h" main.cpp
$ g++ -O2 main.o -L"path/to/libunsigned.a" -lunsigned
```

- Program users do not need that library file to run the program.

Dynamically linked libraries

- Linking happens when the program is executed.
- The system where the program is run needs to find the library machine code and link it with the library function call instructions in the program.
- Disadvantages:
  - Program users need to install binary library files (`.dll`, `.so`).
- Advantages:
  - When the library source code is updated and the new machine code is redistributed to systems (for example, under system packages updates), then programs immediately reflect the library updates.
  - No program change is required; the program binary executable files remain the very same (no recompilation, no relinking).

Dynamically linked libraries (*cont.*)

- *Example:*

```
// unsigned_add.cpp
unsigned add(unsigned a, unsigned b) { return a + b; }
```

```
$ g++ -O2 -c -fPIC unsigned_add.cpp
$ g++ -shared -o libunsigned.so unsigned_add.o
```

- The library binary file is “archived” by GCC.
- Program developers need the library header file for program compilation…

```
// main.cpp
#include <iostream>
#include <unsigned_add.h>
int main() { std::cout << add(0xFFFFFFFFu, 1u) << std::endl; }
```

- …and the library binary file as well to “register” linking:

```
$ g++ -O2 -c -I"/path/to/unsigned_add.h" main.cpp
$ g++ -O2 main.o –L"path/to/libunsigned.so" -lunsigned
```

Dynamically linked libraries (*cont.*)

- The library binary file `libunsigned.so` is now needed by users of the program when this gets executed:

```
$ ./a.out
./a.out: error while loading shared libraries: libunsigned.so: cannot open shared object
file: No such file or directory
$ export LD_LIBRARY_PATH="path/to/libunsigned.so:$LD_LIBRARY_PATH"
$ ./a.out
0
```

- Now, changing the implementation of the library…

```cpp
// unsigned_add.cpp

#include <iostream>
#include <limits>

unsigned add(unsigned a, unsigned b) {
  if (std::numeric_limits<unsigned>::max() - a < b)
    std::cerr << "WARNING: unsigned overflow occurred" << std::endl;

  return a + b;
}
```

- …and rebuilding and redistribution of new `libunsigned.so` is automatically reflected in the program:

```
$ ./a.out
WARNING: unsigned overflow occurred
0
```

Library updates

- Does any update of library code need rebuilding of programs with dynamically-linked libraries?

1. **Updates of library interface:**
   - Machine code of programs that interacts with the library machine code is generated based on the code in library headers.
   - $\Rightarrow$ Changes in these headers usually require program machine code regeneration.

- *Examples:*
  - *Compatible changes:* adding new entities (functions, global variables,…).
  - *Incompatible changes:* changing numbers and types of function parameters, types of returned values, names of entities, removing entities,…
- $\Rightarrow$ It is important to design a library interface in such a way that it would require minimum changes in the future.
- *Note:* Sometimes, it's very hard, since developers do not know which way their libraries will evolve.

Library updates (*cont.*)

2. **Updates of library implementation:**

- *Ideal world:*
  - Header files — functionality interface (what does library do).
  - Source files/machine code — implementation (how does library do that).
  - ⇒ Changes in the implementation do not require rebuilding of programs.
- *Reality:*
  - Implementation details sometimes need to be in header files.
  - Their updates may break binary compatibility between program and library machine code.
  - ⇒ Program rebuilds are required.

Library updates (*cont.*)

- *Example:* library code:

```cpp
// X.h
class X {
  long i_;
public:
  X();
};
```

```cpp
// X.cpp
#include "X.h"
X::X() : i_(0x12L) { }
```

- Program code (*left*) and its translation unit (*right*):

```cpp
// main.cpp
#include <X.h>
int main() {
  X x;
}
```

```cpp
class X {
  long i_;
public:
  X();
};
int main() {
  X x;
}
```

- Observable behavior of `main`:
  - calls default constructor of `X`,
  - calls destructor of `X`,
  - returns `0` to the caller (implicit return for `main`).

C++ object model

- **Object** in C++ = instance of some type (not necessarily class).
- Constructor call is a part of class object initialization.
- To **initialize an object** of type `T`, a **storage** for its binary representation must be provided first.
- This storage must satisfy two requirements:
  - It must be **large enough** — `sizeof(T)` bytes.
  - It must be **properly aligned** — address divisible by `alignof(T)`.
- *Example:*

```
std::cout<< sizeof(long) << alignof(long);  // printed '88' on x86_64/Linux
```

- *Observation:*
  - A storage for any object of type `long` must have 8 bytes and be aligned to an address divisible by 8 on this system.
  - *Note:* It's prescribed by its ABI ⟹ it holds for any compiler.

C++ object model (*cont.*)

- Binary representation of class objects consists of binary representation of its subobjects.

```
class X {
  long i_;   // member variable
public:
  X();
};
```

- Class subobjects:
  - base class objects,
  - non-static member variables.
- Binary representation of class objects is generally implementation-defined.
- Class X has only one subobject — member variable of type `long`.
- ⇒ In our case, the binary representation of X objects was the same as the binary representation of `long` objects.

```
std::cout << sizeof(X) << alignof(X);  // printed '88' on x86_64/Linux/Clang
```

C++ object model (*cont.*)

- **Non-static member function:**
  - A function that is called "on an object of some class".
  - This function needs a "reference" to the object on which it is called.
  - *Note:* This reference is available inside the function body in the form of the pointer `this`.
  - This reference represents a hidden parameter of all non-static member functions.
  - Its form is implementation-dependent.
  - Typical implementations passes an address of the object as argument for this parameter.
- **Constructor** = (special) member function:
  - Object does not yet exist; the purpose of constructors is to create it.
  - ⇒ The caller passes the address of the storage where the object should be initialized ⇒ this storage must be prepared by the caller.
  - Constructor then initializes a binary representation of the object in this storage.

C++ object model (*cont.*)

- Translation unit (*left*), machine code x86_64/Linux/Clang (*right*):

```cpp
class X {
  long i_;
public:
  X();
};
int main() {
  X x;
}
```

```asm
main:
  push  rax
  mov   rdi, rsp
  call  X::X()
  xor   eax, eax
  pop   rax
  ret
```

- In `main`, `x` is a non-static local variable.
  - ⇒ `x` is a new object created in each `main` call.
- `main` needs to:
  - prepare a storage for `x`,
  - call a default constructor of `X`
  - while passing the address of the storage to the constructor.
- *Observation:*
  - Storage for `x` is allocated on the stack (at address `rsp` decreased by 8).
  - Its address is to constructor passed in `rdi` register (as prescribed by ABI).

C++ object model (*cont.*)

- *Other considerations:*
- Definition of the constructor is not in the program translation unit.
  - ⇒ Its call cannot be "inlined"; it must be called explicitly at the machine code level.
- Variable x is an object with so-called "automatic storage duration":
  - Its lifetime ends when the function is leaved.
  - ⇒ Its destructor must be called.
- Destructor of X:
  - There is no custom declaration/definition.
  - ⇒ The destructor is automatically defined.
  - ⇒ As if it was defined (*simplification*) this way

```
class X {
  long i_;
public:
  X();
  ~X() { }
};
```

- ⇒ Destruction of x may be inlined and has no observable behavior.
  - ⇒ It does not generate any machine code under optimizations.

C++ object model (*cont.*)

- Translation unit for the library source file / machine code:

```
class X {
  long i_;
public:
  X();
};
X::X() : i_(0x12L) { }
```

```
X::X()
  mov   qword ptr [rdi], 0x12
  ret
```

- *Recall:* the address of the storage for the constructed object is passed in `rdi`.
- Constructor initializes the binary representation of the constructed object in this storage.
- Binary representation of an object of `X` consists of the binary representation of its member variable `i_`.
- This member variable is initialized with hex value 12.
- ⇒ This value needs to be stored to the address pointed to by `rdi`.

Library updates (*cont.*)

- Summary:
  - *Left:* machine code stored in the program binary file.
  - *Right:* machine code stored in the library binary file (`.so`).

```
main:
  push  rax
  mov   rdi, rsp
  call  X::X()
  xor   eax, eax
  pop   rcx
  ret
```

```
X::X()
  mov   qword ptr [rdi], 0x12
  ret
```

- *Observation:*
  - The machine code responsible for the allocation (and deallocation) of the storage for the constructed object is "hardcoded" in the program file.
- ⇒ Any change of the library that requires a different way of storage allocation will break the binary compatibility.

Library updates (*cont.*)

- Example of binary incompatible library update — adding another member variable:

```
// X.h
class X {
  long i_, j_;
public:
  X();
};
```

```
// X.cpp
#include "X.h"
X::X() : i_(0x12L), j_(0x34L) { }
```

- Machine code of program (*left*) and new library (*right*):

```
main:
  push  rax
  mov   rdi, rsp
  call  X::X()
  xor   eax, eax
  pop   rcx
  ret
```

```
X::X()
  mov   qword ptr [rdi], 0x12
  mov   qword ptr [rdi+8], 0x34
  ret
```

- *Problem:*
  - Program allocates 8 bytes on the stack, but constructor internally **overwrites 16 bytes!**

Library updates (*cont.*)

```
main:
  push  rax
  mov   rdi, rsp
  call  X::X()
  xor   eax, eax
  pop   rcx
  ret
```

```
X::X()
  mov  qword ptr [rdi], 0x12
  mov  qword ptr [rdi+8], 0x34
  ret
```

- *Consequence:*
  - A constructor overwrites some **part of the stack that does not belong to the constructed object**.
- In our case, it overwrites the return address where the ret instruction “jumps” when the constructor is finished.
  - In *best case*, the program just crashes.
  - In *worst case*, it starts executing some (un)predicted machine code.
- $\Rightarrow$ Such a vulnerability might be generally exploited.

Library updates (*cont.*)

- *Recapitulation:*
  - Original library code (*black*), updates (with *red*):

```
// X.h
class X {
  long i_, j_;
public:
  X();
};
```

```
// X.cpp
#include "X.h"
X::X() : i_(0x12L), j_(0x34L) { }
```

- Updates logically involved only implementation library details:
  - Whatever is private to a class is logically related to implementation of its functionality.
- However, even private members **must be declared within class definition** (body).
  - ⇒ These declarations must appear in header files.

- C++ generally does not allow a complete logical separation of interface and implementation details between header and source files, respectively.
- *Workaround:* PIMPL idiom.

PIMPL idiom

- *PIMPL = Pointer-to-IMPLementation.*
- Technique for "true" hiding of all implementation details out of header files.
- *Functionality:*
  - Original class is "split" into two classes.
  - *First* — "implementation" class — contains the original class implementation details including all its member variables.
  - *Second* — "interface" class — contains:
    1) a pointer to the implementation class object,
    2) interface of the original class = its public member functions.
  - Public member functions of the interface class need to invoke member functions of the implementation class.
    - This is done internally inside library source files.
- *Outcome:*
  - Interface class does not need to be changed when implementation class member variables and member functions are updated in any way.
  - ⇒ If the "true interface" — public member functions — is not changed, there is no need for programs rebuilding with dynamically linked libraries.

PIMPL idiom *(cont.)*

- *Example — original class:*

```cpp
// X.h
class X {
  long i_;
public:
  X();
  void do_something();
};
```

```cpp
// X.cpp
#include <iostream>
#include "X.h"
X::X() : i_(0x12L) { }
X::do_something() {  std::cout << i_;  }
```

- *"PIMPLed" solution:*

```cpp
// X.h
class X {  // interface class
  class Impl;
    // -> implementation...
    // ...class declaration
  Impl* pimpl_;
    // -> pointer...
    // ...to implementation
public:
  X();
  ~X();
  void do_something();
};
```

```cpp
// X.cpp
#include <iostream>
#include "X.h"
class X::Impl {  // implementation class
  long i_;
public:
  Impl::Impl() : i_(0x12L) { }
  void do_something() {  std::cout << i_;  }
};
// interface-implementation binding:
X::X() : pimpl_( new Impl{} ) { }
X::~X() { delete pimpl_; }
X::do_something() {  pimpl_->do_something();  }
```

PIMPL idiom *(cont.)*

- *Library update:*

```cpp
// X.h
class X {
  class Impl;
  Impl* pimpl_;
public:
  X();
  ~X();
  void do_something();
};
```

```cpp
// X.cpp
#include <iostream>
#include "X.h"
class X::Impl {
  long i_, j_;
public:
  Impl::Impl() : i_(0x12L), j_(0x34L) { }
  void do_something() {
    std::cout << i_ << " " << j_;
  }
};
X::X() : pimpl_( new Impl{} ) { }
X::~X() { delete pimpl_; }
X::do_something() {  pimpl_->do_something();  }
```

- *Observation:*
  - No change in the header file ⇒ no need for program rebuild (with dynamic library linking).
- *Notes:*
  - In `X.h`, class `X::impl` is only declared ⇒ it is so-called "*incomplete type*".
  - It is legal to declare a pointer-to-incomplete type (`pimpl_` in our case).
  - All the operations that require this type to be complete (such as `new` and `delete` *expressions*) are in the source file.
    - The class gets complete after its definition.