TDDD38/726G82:
Adv. Programming in C++
Advanced Memory I

Christoffer Holm

Department of Computer and information science

Dynamic Memory

Prerequisites – automatic storage

- As previously discussed in the course (and hopefully other courses as well) the C++ memory model assumes a particular structure of memory.
- Local variables, parameters, return addresses etc. are stored in the memory region called the *stack*.
- The lifetime of these objects are bound to the surrounding scope meaning they are “destroyed” when the program execution leaves the scope.
- Sometimes these are called objects with *automatic storage*.
- The size of the stack is usually fixed during the complete runtime of the program. The size of each *stack frame* must be – in standard C++ – statically known, meaning it must be known during compile-time.
- This incurs certain restrictions on what can and cannot be done with the stack memory. Typically we can not create automatic objects whose lifetime exceeds the surrounding scope. We cannot dynamically create objects during runtime and we cannot create objects that are larger than the stack.
- All of these restrictions are handled by *dynamic memory*.

Dynamic Memory

Prerequisites – Dynamic memory

- Dynamic memory refers to the memory that holds objects that are not bound to a specific scope nor automatically managed.
- Dynamically allocated objects (i.e. the objects stored in dynamic memory) are stored in a region of memory usually called the *heap* (so named because originally this region was modelled as the data structure called heap).
- At the lowest level dynamic memory is handled by reqesting memory allotment from the operating system (during program execution). On top of this dynamically managed memory there is some type of infrastructure that allows for further granularity where objects can be dynamically placed within the alloted memory by the user code.
- The default way dynamic memory is handled in C++ is by using `malloc()` and `free()` which *allocates* and *deallocates* objects within the heap.
- However, these functions only manages memory, they do *not* manage *objects*. Dynamically allocated objects are handled using `operator new()` and `operator delete()`.

Dynamic Memory

Dynamic memory

`operator new()`

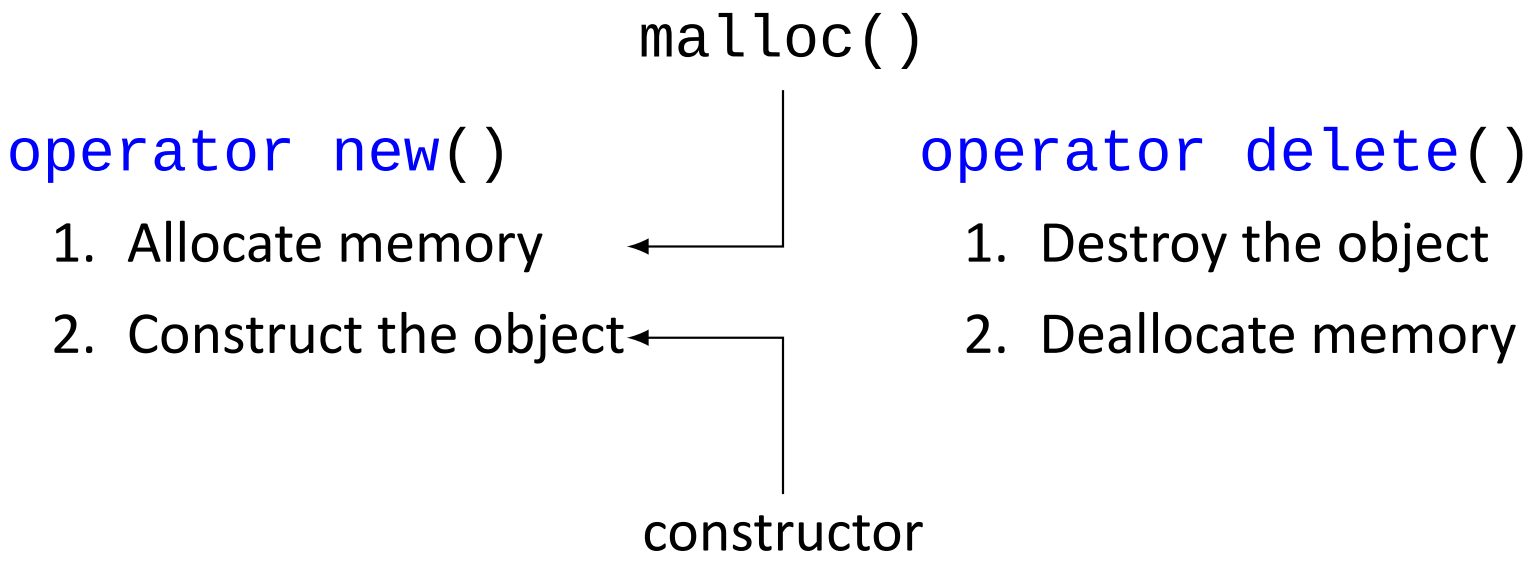
1. Allocate memory
2. Construct the object

`operator delete()`

1. Destroy the object
2. Deallocate memory

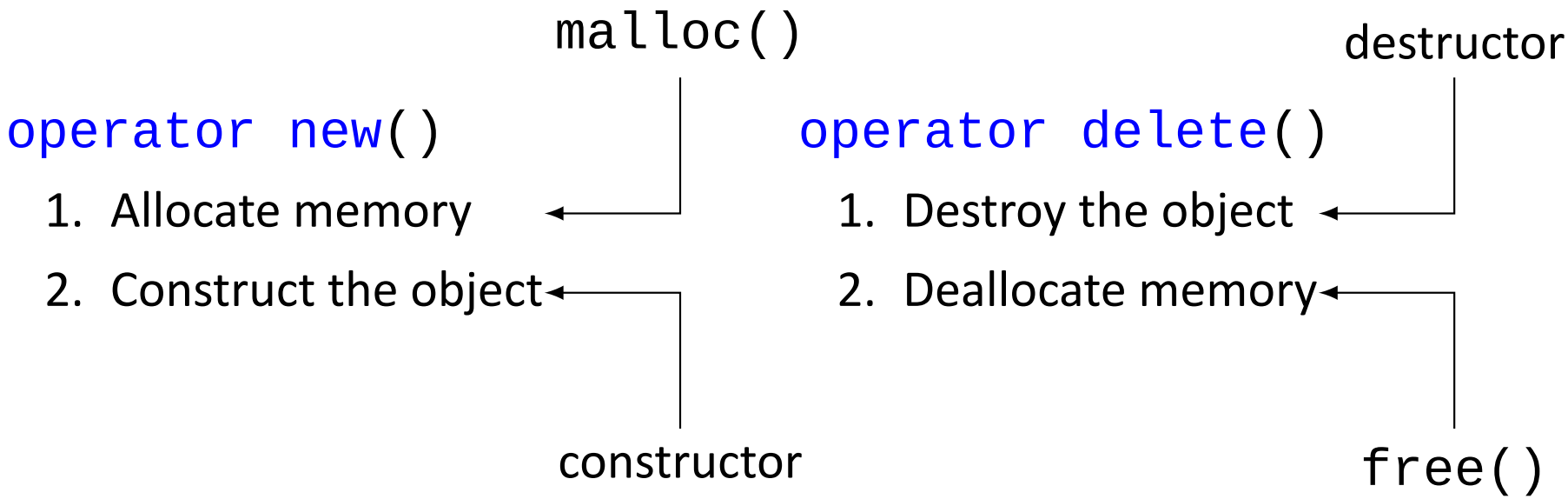
Dynamic Memory

Dynamic memory



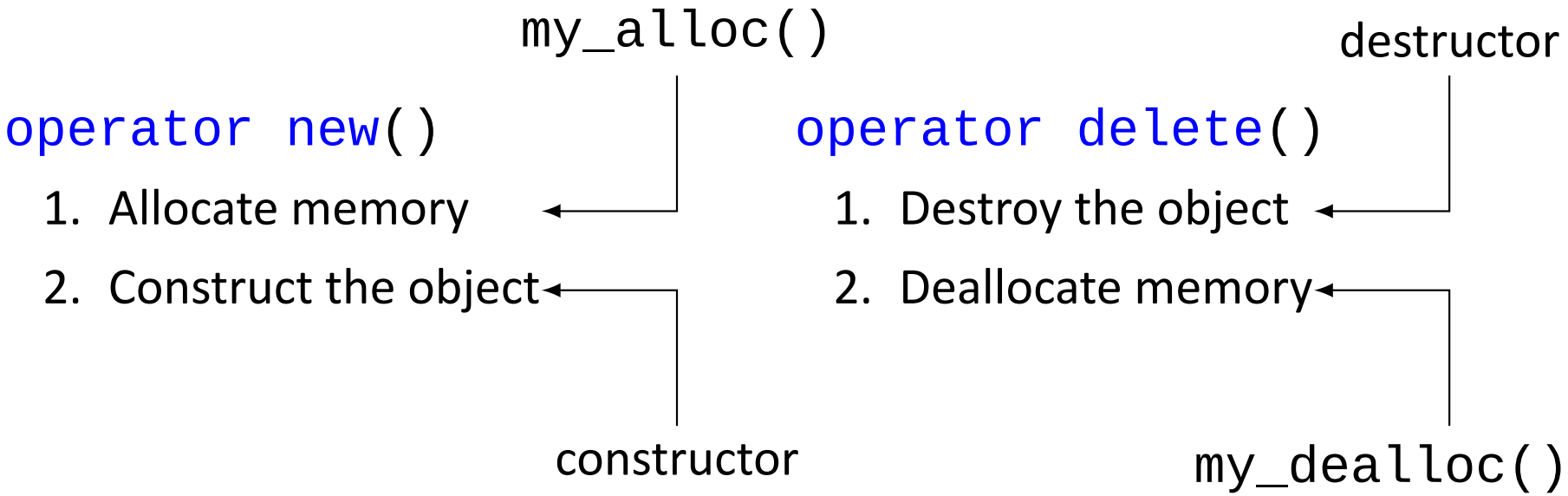
Dynamic Memory

Dynamic memory



Dynamic Memory

Replacing `malloc()` and `free()`



Dynamic Memory

Replacing `malloc()` and `free()`

- It is possible to change the allocation behaviour of `new` and `delete` by overloading `operator new()` and `operator delete()`.
- This allows us to change the global allocation strategy for our program. This means we can replace `malloc()` and `free()` with something that is more appropriate for our particular use-case.
- There is for example a popular replacement for `malloc()` called *jemalloc* (`https://jemalloc.net/`)
- For embedded systems `malloc()` might be too expensive, in those cases you might want a cheaper implementation (for example `https://github.com/rhempel/umm_malloc`).
- The possibilites are endless when it comes to efficient memory management, but either way: step one is to overload the default global behaviour.

Dynamic Memory

Replacing `malloc()` and `free()`

```
1  void* operator new(std::size_t nbytes)
2  {
3    return my_alloc(nbytes);
4  }
5
6  void operator delete(void* memory) noexcept
7  {
8    return my_dealloc(memory);
9  }
```

Dynamic Memory

Exploring dynamic memory

- In this course we will not be replacing `malloc()` for efficiency reasons, instead we will primarily do it to help us explore how memory actually works.
- On the next slide there is an implementation of an allocation strategy that simply *logs* all allocations which allows us to more explicitly see how dynamic memory is managed.

Dynamic Memory

Exploring dynamic memory

```
1  void* operator new(std::size_t n)
2  {
3    std::cout << "Allocated " << n << " bytes: ";
4    auto memory = std::malloc(n);
5    std::cout << memory << std::endl;
6    return memory;
7  }
8  void operator delete(void* memory) noexcept
9  {
10   std::cout << "Dealloc: " << memory << std::endl;
11   std::free(memory);
12 }
```

Dynamic Memory

How much dynamic memory is allocated?

```
1  #include <iostream>
2  #include <vector>
3
4  /* operator new() and operator delete() */
5
6  std::vector<int> v1 { 1, 2, 3 }; // 12 bytes?
7  int main()
8  {
9    std::cout << "Program running!" << std::endl;
10   int x { 5 };
11   std::vector<int> v2 { 4, 5 }; // 8 bytes?
12 }
```

Dynamic Memory

How much dynamic memory is allocated? – Output

```
Allocated 12 bytes: 0x5562b76286c0
Program running!
Allocated 8 bytes: 0x5562b76286e0
Dealloc: 0x5562b76286e0
Dealloc: 0x5562b76286c0
```

Dynamic Memory

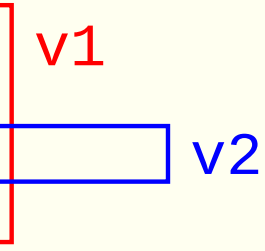
How much dynamic memory is allocated? – Output

```
Allocated 12 bytes: 0x5562b76286c0
Program running!
Allocated 8 bytes: 0x5562b76286e0
Dealloc: 0x5562b76286e0
Dealloc: 0x5562b76286c0
```

Dynamic Memory

How much dynamic memory is allocated? – Output

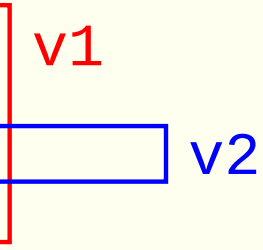
```
Allocated 12 bytes: 0x5562b76286c0
Program running!
Allocated 8 bytes: 0x5562b76286e0
Dealloc: 0x5562b76286e0
Dealloc: 0x5562b76286c0
```



Dynamic Memory

How much dynamic memory is allocated? – Output

```
Allocated 12 bytes: 0x5562b76286c0
Program running!
Allocated 8 bytes: 0x5562b76286e0
Dealloc: 0x5562b76286e0
Dealloc: 0x5562b76286c0
```

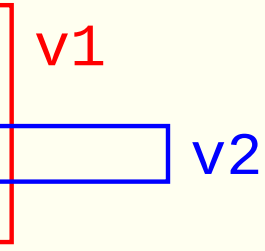


Where is `std::cout`?

Dynamic Memory

How much dynamic memory is allocated? – Output

```
Allocated 12 bytes: 0x5562b76286c0
Program running!
Allocated 8 bytes: 0x5562b76286e0
Dealloc: 0x5562b76286e0
Dealloc: 0x5562b76286c0
```

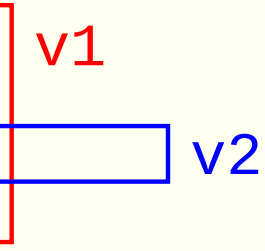


Where is `std::cout`? Where is `v1`?

Dynamic Memory

How much dynamic memory is allocated? – Output

```
Allocated 12 bytes: 0x5562b76286c0
Program running!
Allocated 8 bytes: 0x5562b76286e0
Dealloc: 0x5562b76286e0
Dealloc: 0x5562b76286c0
```

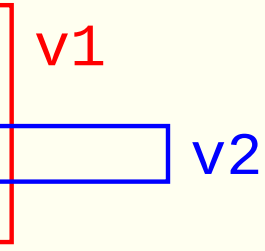


Where is `std::cout`? Where is `v1`? Where are the string literals?

Dynamic Memory

How much dynamic memory is allocated? – Output

```
Allocated 12 bytes: 0x5562b76286c0
Program running!
Allocated 8 bytes: 0x5562b76286e0
Dealloc: 0x5562b76286e0
Dealloc: 0x5562b76286c0
```



Where is `std::cout`? Where is `v1`? Where are the string literals? Where is the program code?

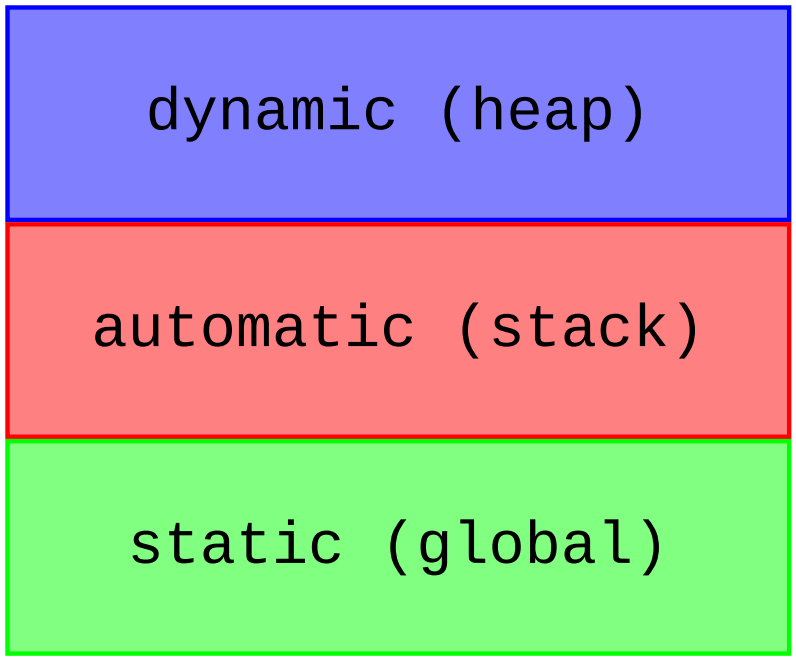
Dynamic Memory

Some observations

- `x` and `v2` will end up in the stack, so no dynamic allocations are needed for those (however, when `v2` is constructed it will allocate the actual content)
- The stack generally is created right before `main()` is called
- `std::cout` is an object, so it must exist somewhere. However, it is seemingly available *before* `main()` is called, since it worked in the constructor of `v1`, so it cannot exist on the stack.
- Likewise, `v1` is seemingly created before the stack.
- What is going on?

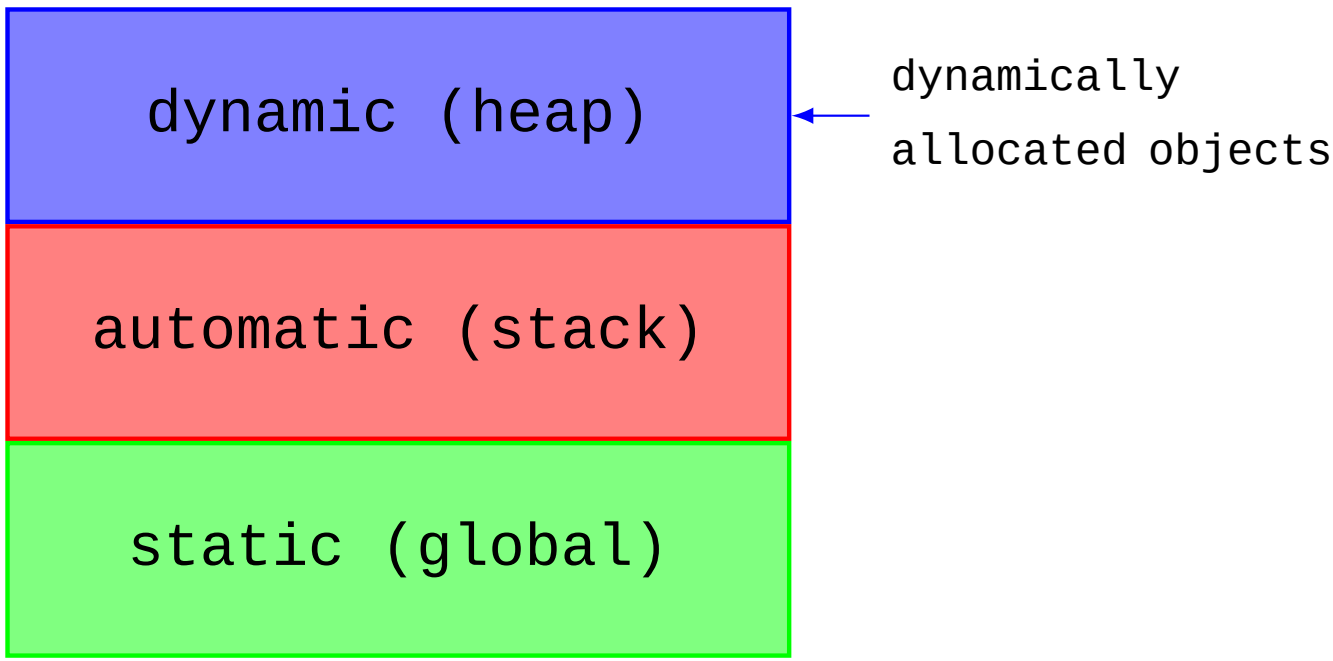
Static Memory

The complete model



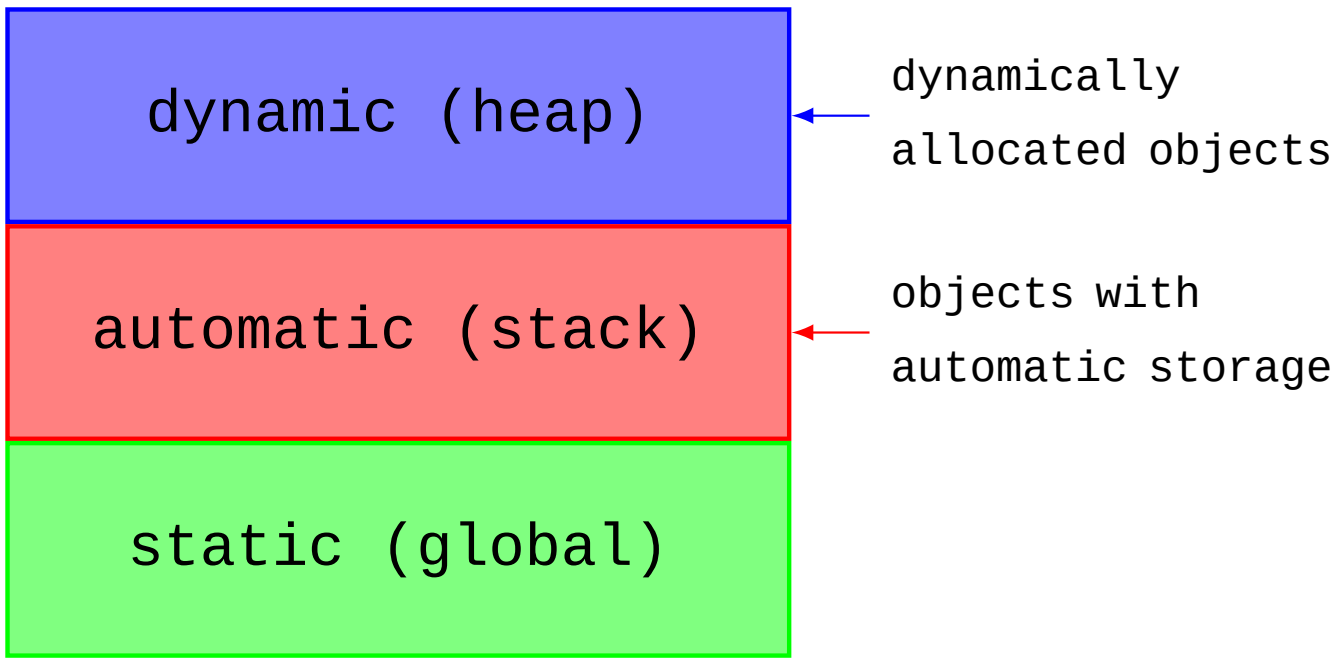
Static Memory

The complete model



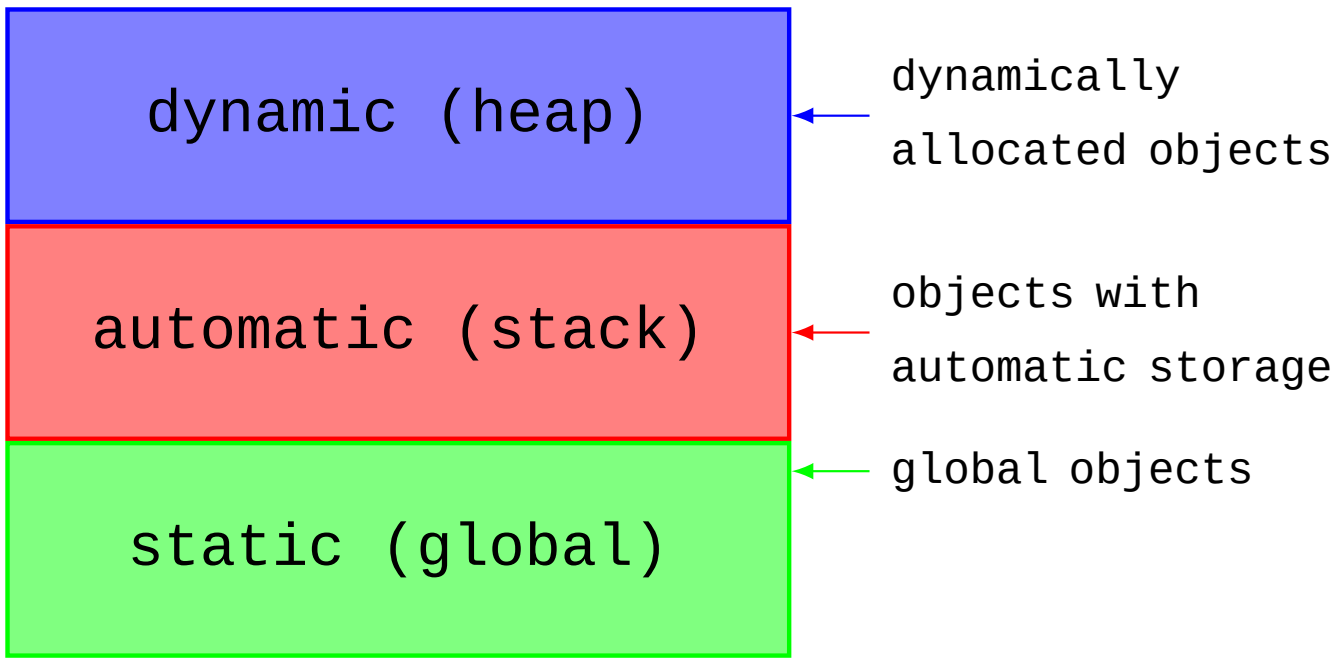
Static Memory

The complete model



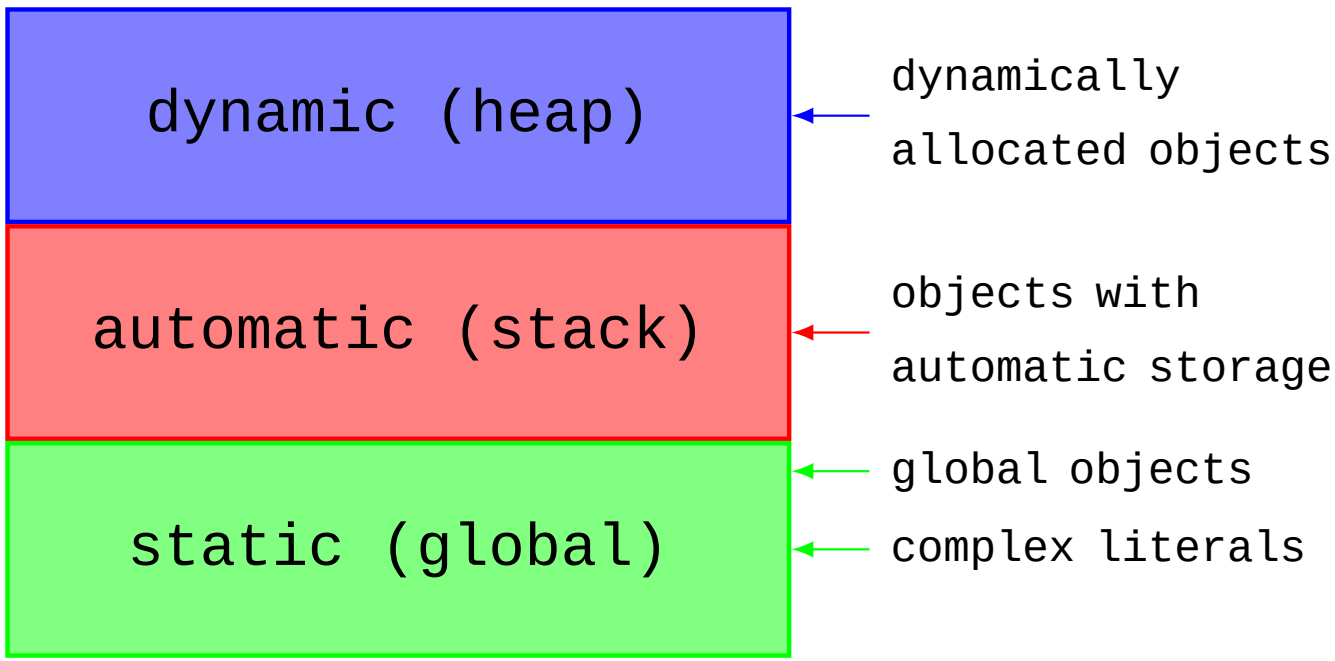
Static Memory

The complete model



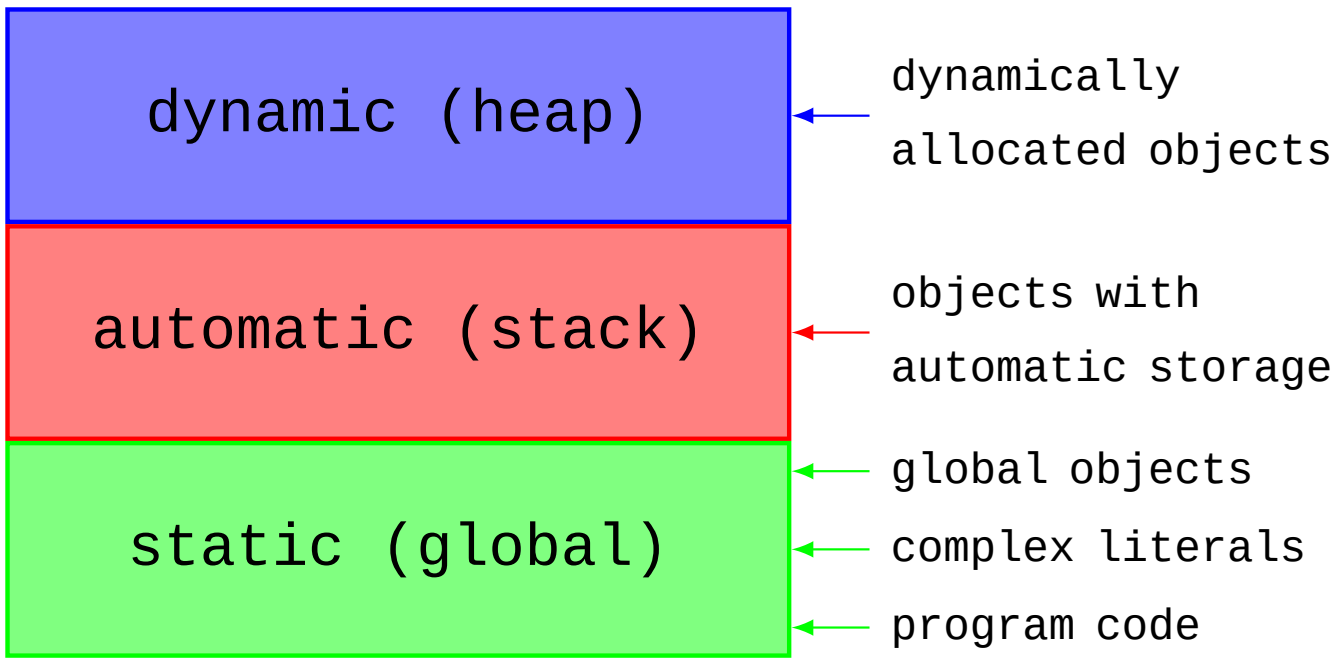
Static Memory

The complete model



Static Memory

The complete model



Static Memory

What is static memory

- The heap handles dynamic memory (dynamically allocated objects) while the stack handles automatically managed memory (local/automatic objects)
- Both the heap and the stack handles objects that are created *during* execution of the program, i.e. their existance depends on the logical behaviour during program execution
- But there are objects whose existance is known *statically* i.e. there are objects (and memory in generally) that provably will *always* exist regardless of what happens during program execution. In other words: there are objects that are known to exist during *compile-time*
- These are things such as global variables, static members, C-string literals (and other literals) as well as the machine code of the program
- All of these things are stored in the third memory region, often called the *data segment* or the *static memory region*.
- This region can be, depending on platform, further divided into subregions, but for the purposes of this course that does not matter.

Static Memory

Why do we care?

- Since the objects in static memory are known during compile-time this has some interesting consequences.
- The first observation we can make is that objects stored in static memory will be constructed either during compilation (if no runtime behaviour is required to construct them) or at program startup (if runtime behaviour is needed for initialization) and destroyed at program termination (i.e. at program exit)
- Each statically known object must generally be initialized in sequence, so during *static memory initialization* there might be objects that have not yet been constructed, but after all of them have been initialized, they all will exist until program termination.
- So objects stored in static memory is still automatically managed, but they are (generally) not local objects but rather some form of *global* objects.

Static Memory

Examples of statically known objects

```
1  int x { 5 }; // global variable
2
3  int main()
4  {
5    //      C-string literal
6    //              |
7    //              V
8    std::cout << "x = " << x << std::endl;
9  }
```

Static Memory

Interesting consequence – Why?

```
1  template <int& ref>
2  int& fun()
3  {
4    return ref;
5  }
6
7  int x { 1 }; // global
8  int main()
9  {
10   int y { 2 }; // local
11   std::cout << fun<x>() << std::endl; // OK
12   std::cout << fun<y>() << std::endl; // Not OK
13 }
```

Static Memory

Interesting consequence – Why?

- The location of the stack and the heap are determined at program startup (i.e. at the beginning of the runtime) by the operating system.
- Since some systems might apply *address space layout randomization* (a cybersecurity measure to avoid certain exploits and vulnerabilities) where the address space of a process is randomized at startup, **NO** addresses are known during compilation since this is the only memory that the *compiler* has control over.
- However, maybe we can, during compilation, find where an object is stored relative to a memory address we know *will* be known? When it comes to the stack and the heap, the answer is no, since the placement of objects in those regions are entirely based on what happens during program execution.
- This is however not true for the static region. The objects relative position in static storage is completely known during compilation.
- Non-type template parameters must be what is called *constant-evaluated expressions*, meaning they must be expressions that has meaning during compilation **and** can be evaluated during compile-time. This only applies to addresses (or references) into static storage, but not true for any other memory.

Static Memory

Static local variables

```
1  int next()
2  {
3    static int counter { 0 };
4    return counter++;
5  }
6
7  int main()
8  {
9    std::cout << next() << std::endl; // 0
10   std::cout << next() << std::endl; // 1
11   std::cout << next() << std::endl; // 2
12 }
```

Static Memory

Static local variables

- A second example of something which is stored in static memory is so-called *static local variables*
- These are local variables witin a function which are marked *static*. They represent global variables that are only accessible within that function.
- They have one major difference compared to ordinary global variables, namely that they are initialized the *first* time the function is *called* (so not at program startup)
- They are still destroyed at program termination.

Static Memory

Summary of the memory regions

- Dynamic (heap) – managed by the user
- Automatic (stack) – managed by the runtime
- Static (global) – managed by the compiler

Static Memory

Another example: Static members

```
1  struct Cls
2  {
3    static int y { 3 };
4  };
```

Static Memory

Another example: Static members

Doesn't compile...

```
1  struct Cls
2  {
3    static int y { 3 };
4  };
```

Static Memory

Another example: Static members

```
1  struct Cls
2  {
3    static int y { 3 };
4  };
```

Why not?

Static Memory

Static members: how do they work?

file.h

```
1  #pragma once
2  struct Cls
3  {
4    static int y { 3 };
5    static void fun();
6  };
```

file.cc

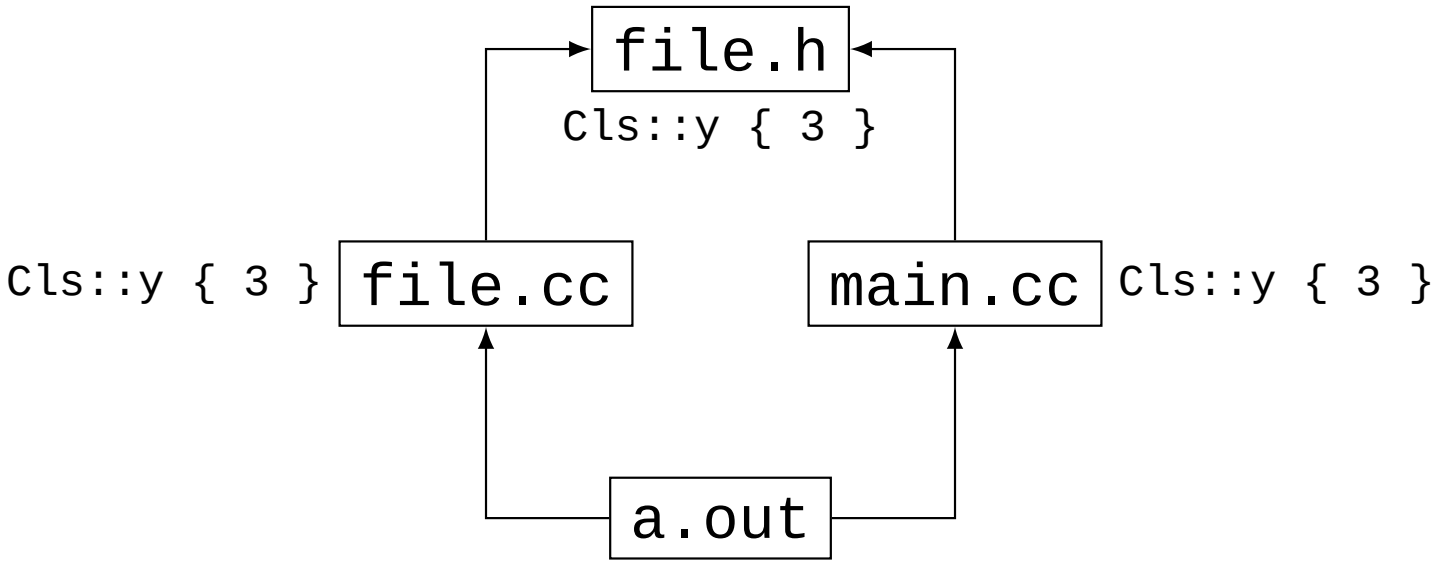
```
1  #include "file.h"
2
3
4  void Cls::fun()
5  {
6    std::cout << y << std::endl;
7  }
```

main.cc

```
1  #include "file.h"
2
3  int main()
4  {
5    // modify Cls::y
6    ++Cls::y;
7
8    // Print Cls::y
9    Cls::fun();
10 }
```

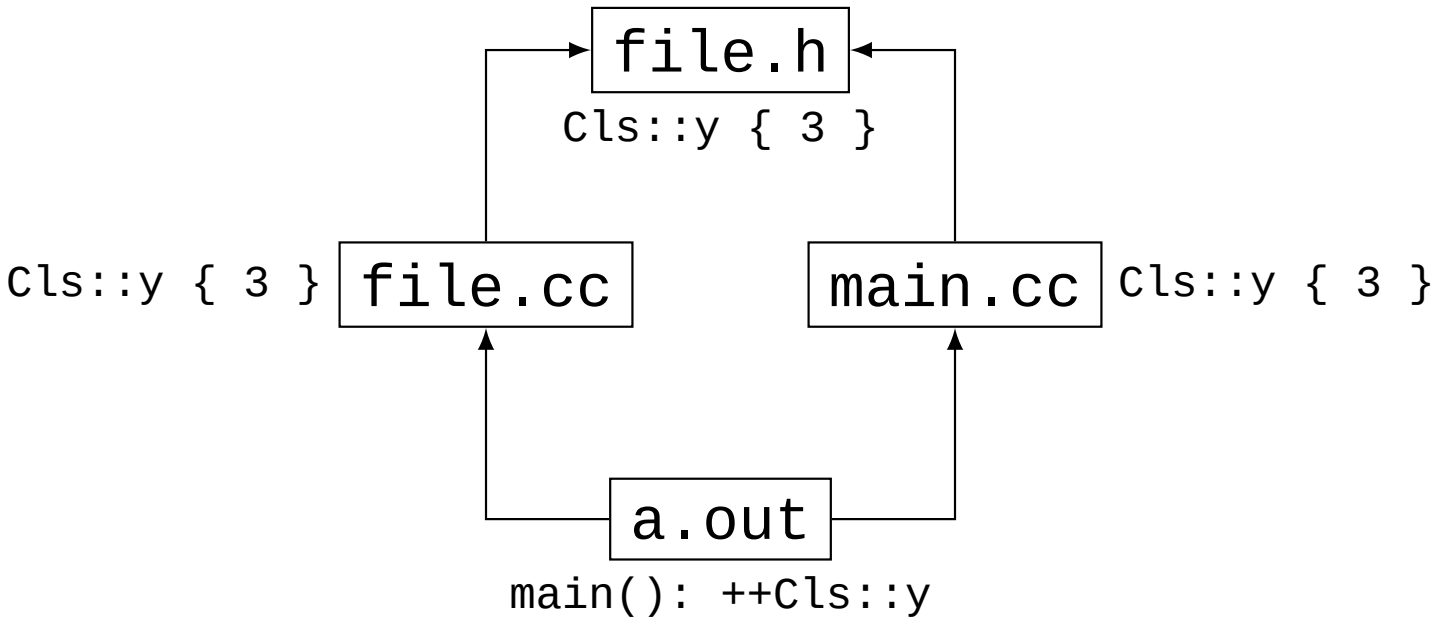
Static Memory

Static members and compilation



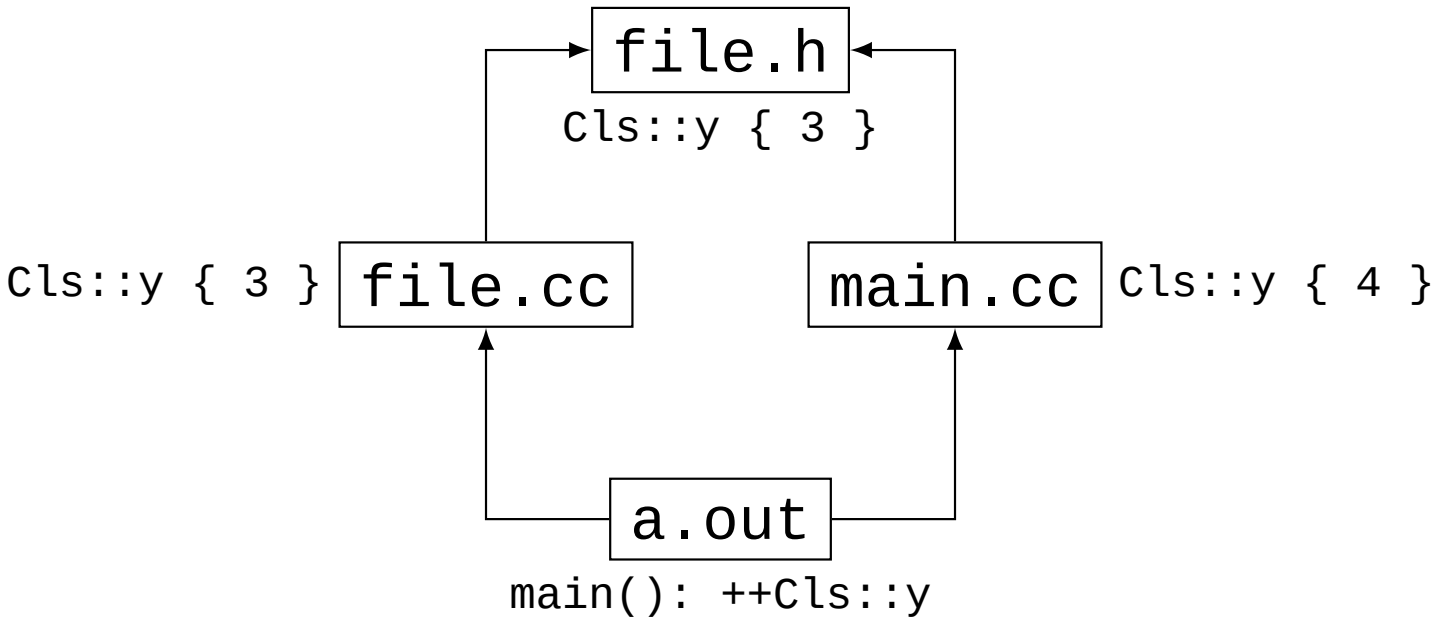
Static Memory

Static members and compilation



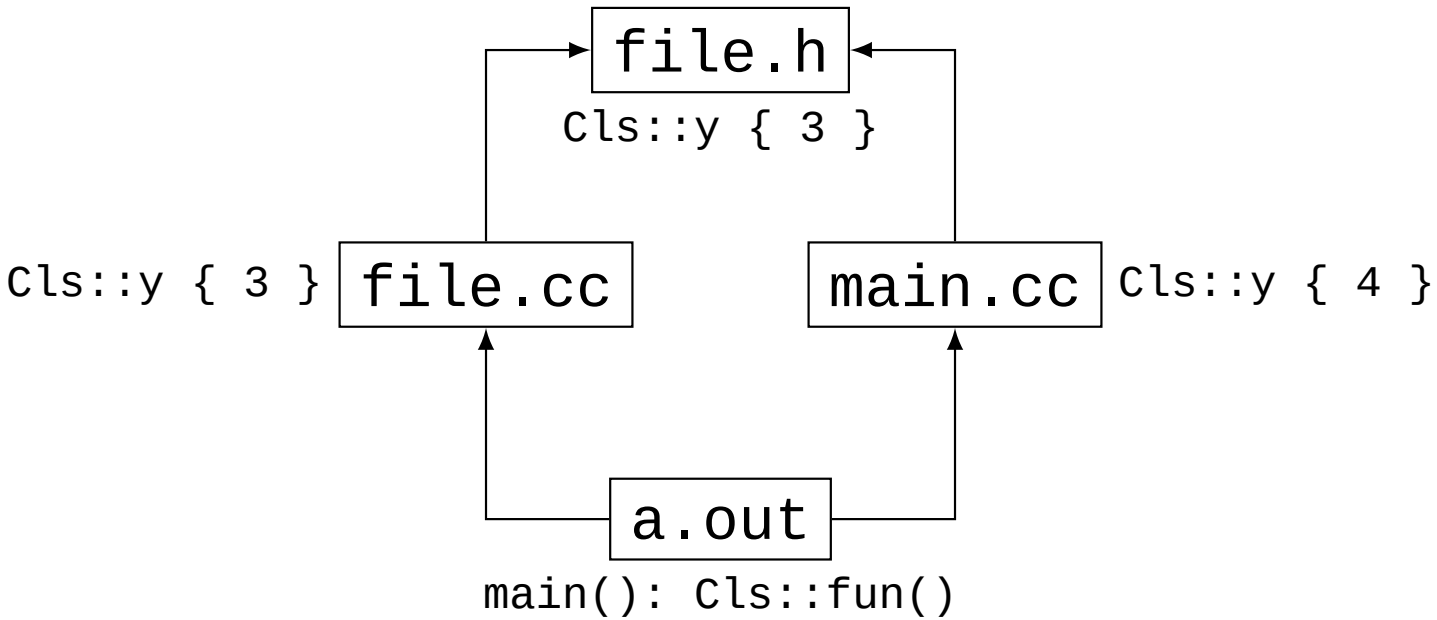
Static Memory

Static members and compilation



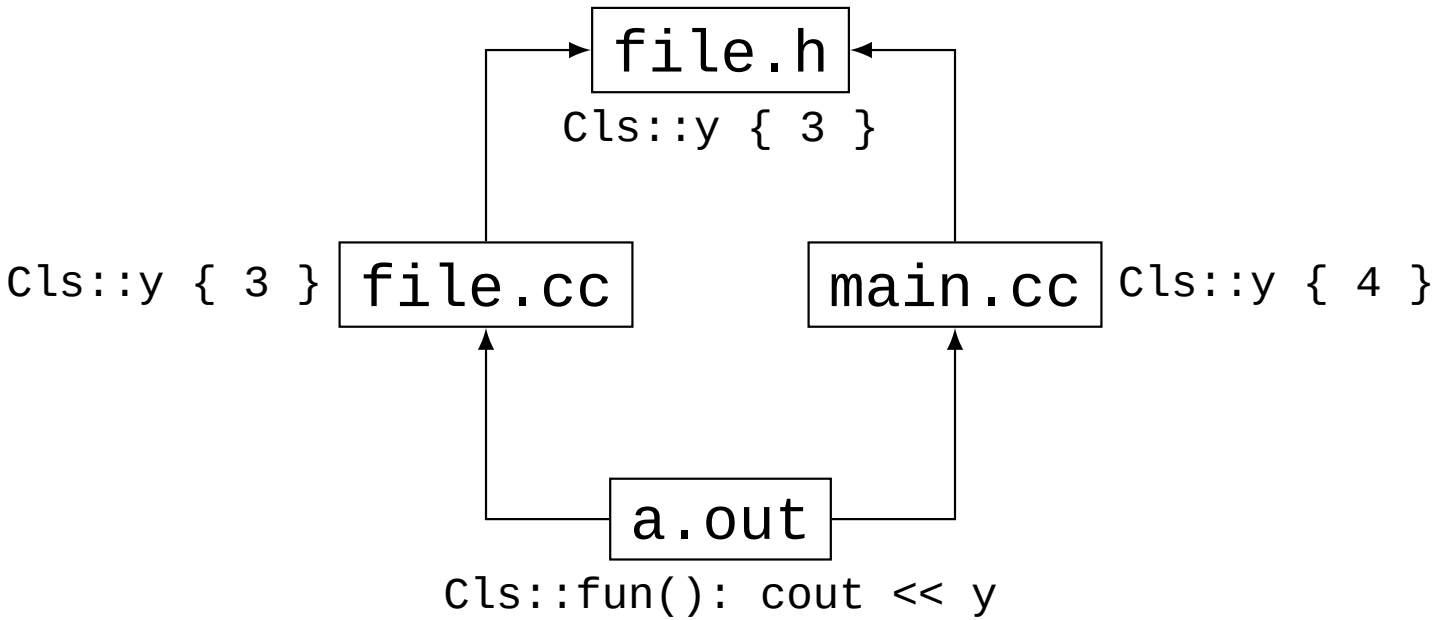
Static Memory

Static members and compilation



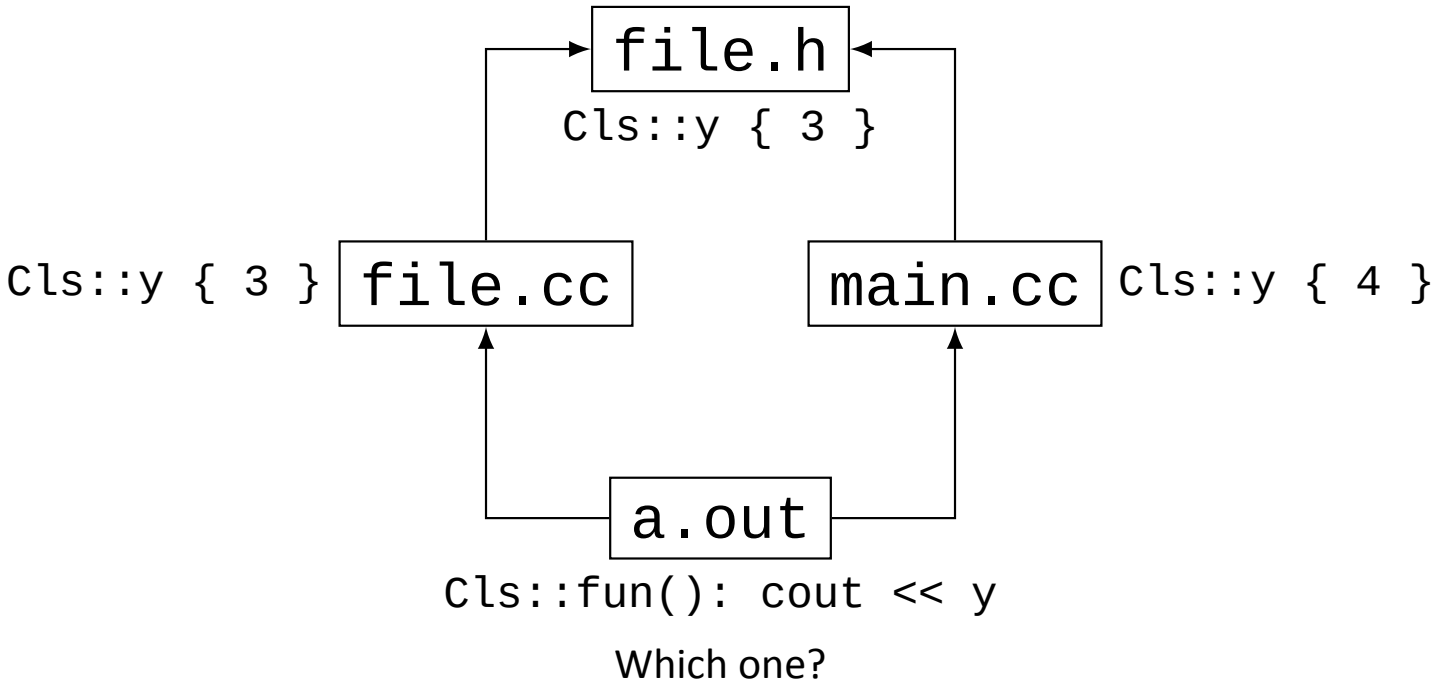
Static Memory

Static members and compilation



Static Memory

Static members and compilation



Static Memory

Static members and compilation

- A global variable *defined* in a header file will lead to each translation unit (which includes the header file) to have its *own* definition of the variable
- This means that `file.cc` and `main.cc` will have their own versions of `Cls::y`
- When `main()` is compiled, it will use the `Cls::y` version that is available within the same translation unit
- Likewise, `Cls::fun()` will use the `Cls::y` version which is available in the `file.cc` file.
- So we will have inconstent behaviour across different translation units
- To discourage this problem, the standard specifies that we are not allowed to initialize static members inside the class definition.

Static Memory

The fix

file.h

```
1  #pragma once
2  struct Cls
3  {
4    static int y;
5    static void fun();
6  };
```

file.cc

```
1  #include "file.h"
2
3  int Cls::y { 3 };
4  void Cls::fun()
5  {
6    std::cout << y << std::endl;
7  }
```

main.cc

```
1  #include "file.h"
2
3  int main()
4  {
5    // modify Cls::y
6    ++Cls::y;
7
8    // Print Cls::y
9    Cls::fun();
10 }
```

Static Memory

The fix

file.h

```
1  #pragma once
2  struct Cls
3  {
4    static int y;
5    static void fun();
6  };
```

file.cc

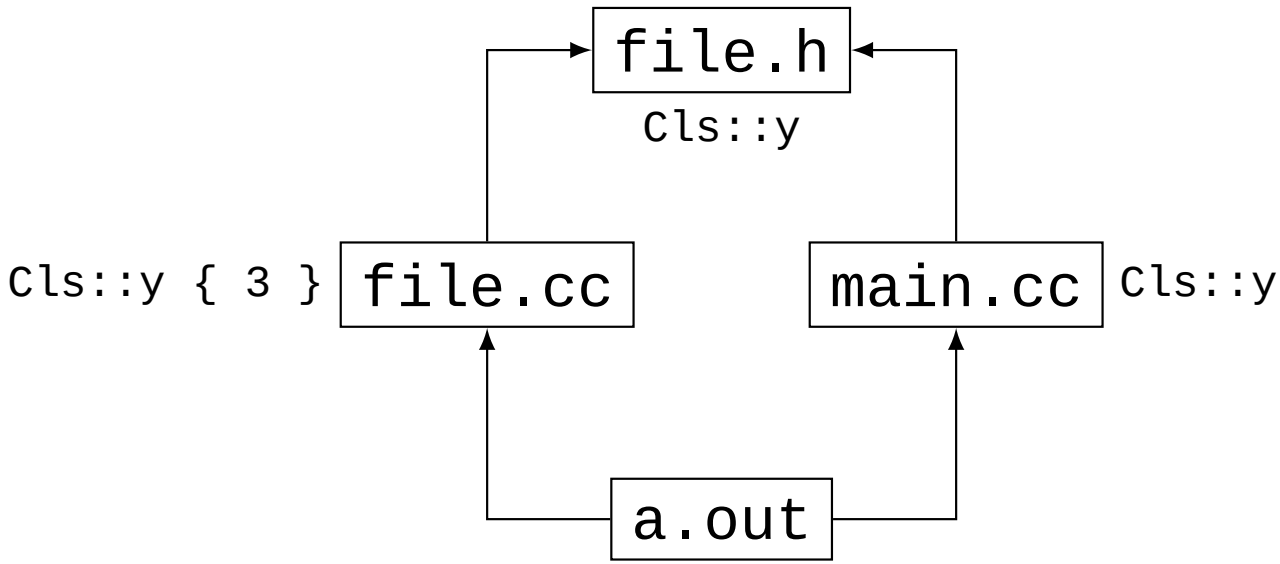
```
1  #include "file.h"
2
3  int Cls::y { 3 };
4  void Cls::fun()
5  {
6    std::cout << y << std::endl;
7  }
```

main.cc

```
1  #include "file.h"
2
3  int main()
4  {
5    // modify Cls::y
6    ++Cls::y;
7
8    // Print Cls::y
9    Cls::fun();
10 }
```

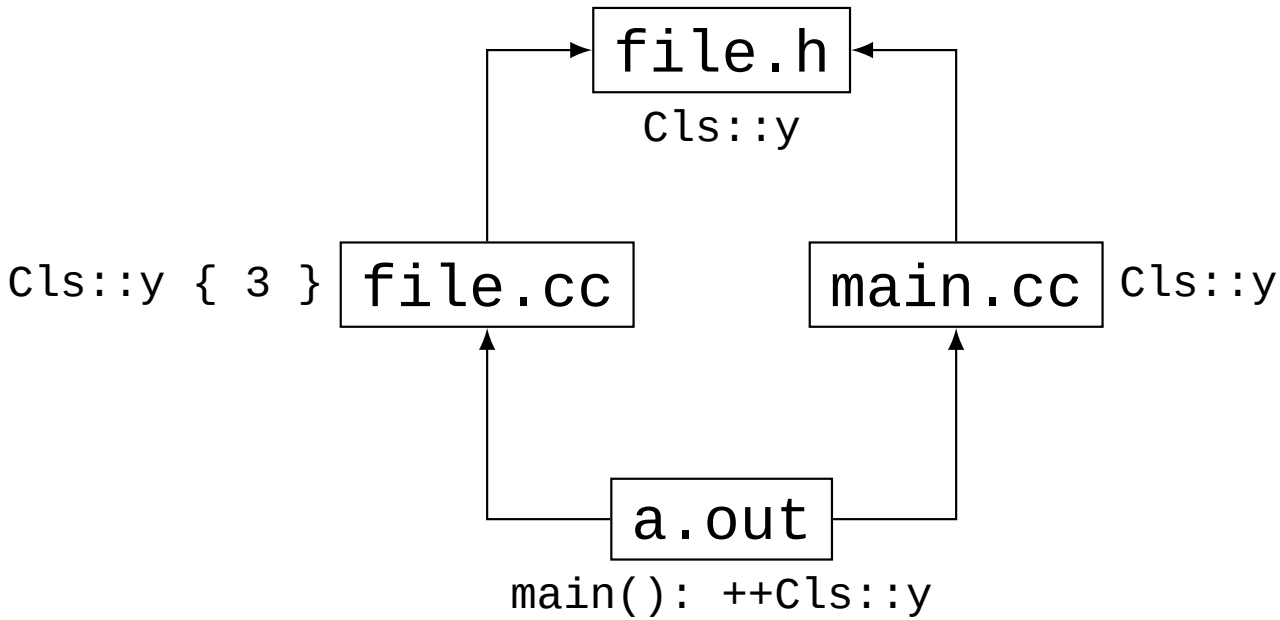
Static Memory

The fix



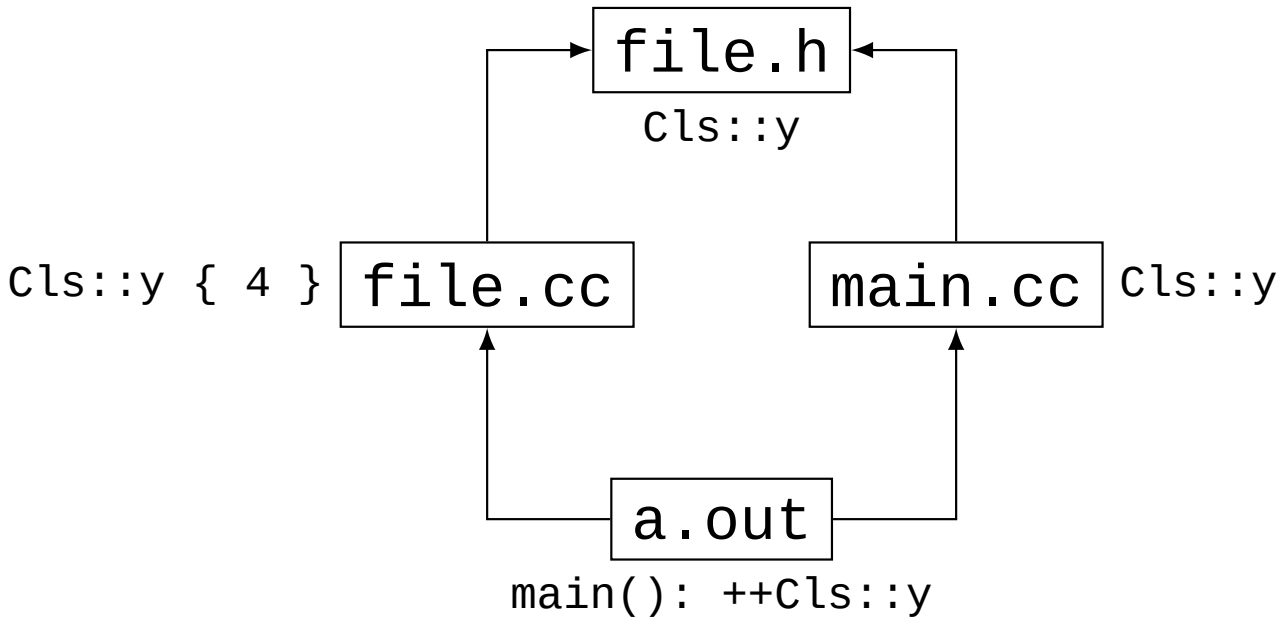
Static Memory

The fix



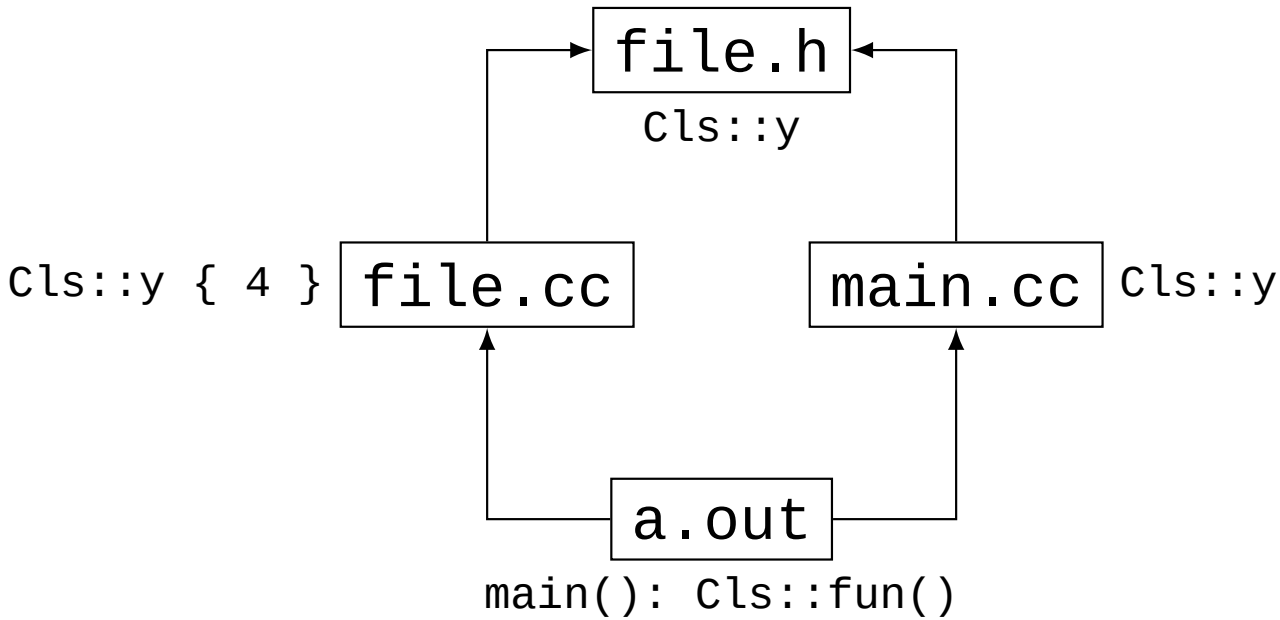
Static Memory

The fix



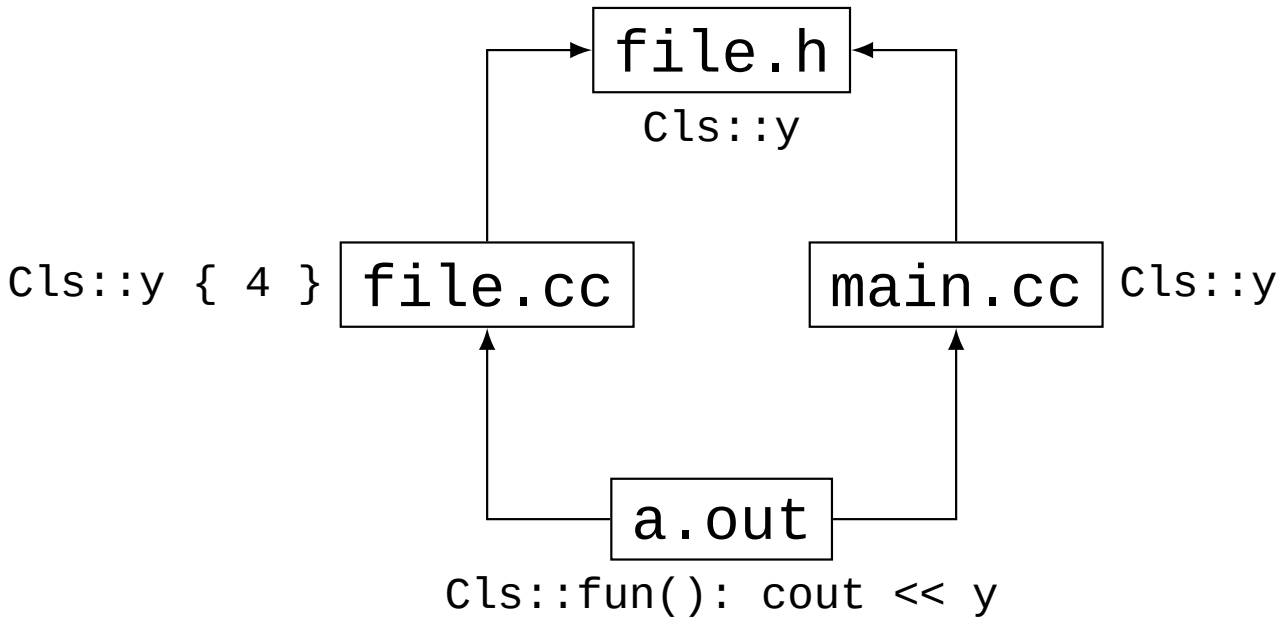
Static Memory

The fix



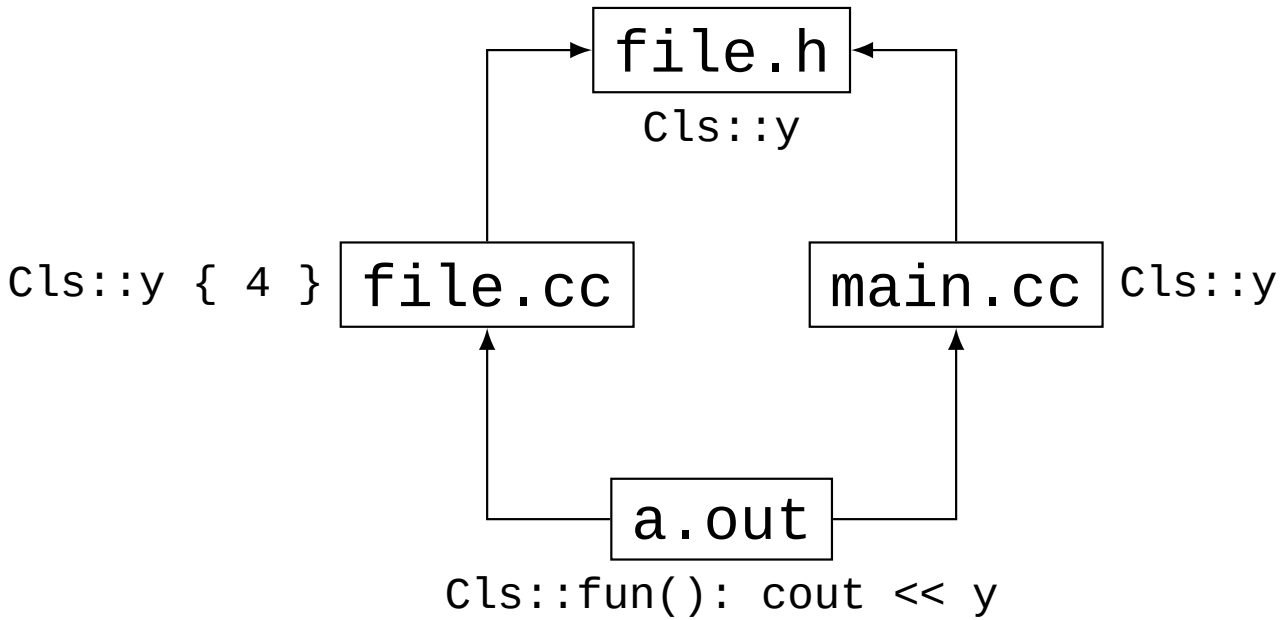
Static Memory

The fix



Static Memory

The fix



Now there is only one option!

Static Memory

The fix

- To fix this, we must ensure that there is only *one* definition in the entire code base.
- The easiest and safest way to ensure that is to declare the variable in the header file and then *define* (initialize) it in corresponding implementation file.
- That way, we ensure that the linker can only find exactly *one* definition of the variable

Static Memory

Mistake – **Don't do this**

file.h

```
1  #pragma once
2  struct Cls
3  {
4    static int y;
5    static void fun();
6  };
7  int Cls::y { 3 }; // The mistake
```

file.cc

```
1  #include "file.h"
2
3  void Cls::fun()
4  {
5    std::cout << y << std::endl;
6  }
```

main.cc

```
1  #include "file.h"
2
3  int main()
4  {
5    // modify Cls::y
6    ++Cls::y;
7
8    // Print Cls::y
9    Cls::fun();
10 }
```

Static Memory

Why is this allowed?

```
1 struct Cls
2 {
3   int const y { 3 }; // allowed, why?
4 };
```

LINKÖPING UNIVERSITY

One Definition Rule (ODR)

Similar problems

file.h

```
1  #pragma once
2
3  int fun(int x)
4  {
5    return x + 1;
6  }
7
8  int other();
```

file.cc

```
1  #include "file.h"
2
3  int other()
4  {
5    return fun(1);
6  }
```

file.cc

```
1  #include "file.h"
2  #include <iostream>
3
4  int main()
5  {
6    std::cout << other() << std::endl;
7    std::cout << fun(3) << std::endl;
8  }
```

One Definition Rule (ODR)

Similar problems

file.h

```
1  #pragma once
2
3  int fun(int x)
4  {
5    return x + 1;
6  }
7
8  int other();
```

file.cc

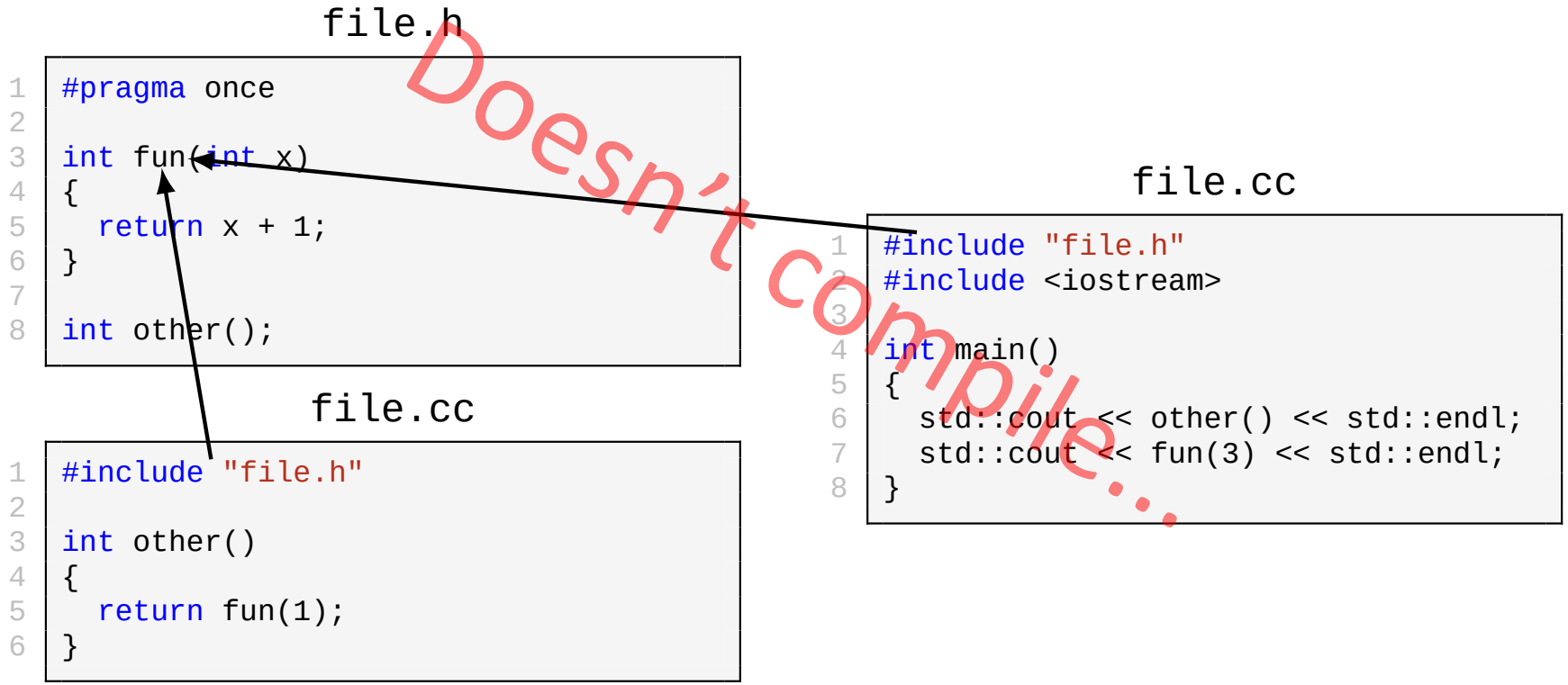
```
1  #include "file.h"
2
3  int other()
4  {
5    return fun(1);
6  }
```

file.cc

```
1  #include "file.h"
2  #include <iostream>
3
4  int main()
5  {
6    std::cout << other() << std::endl;
7    std::cout << fun(3) << std::endl;
8  }
```

One Definition Rule (ODR)

Similar problems



One Definition Rule (ODR)

Similar problems

file.h

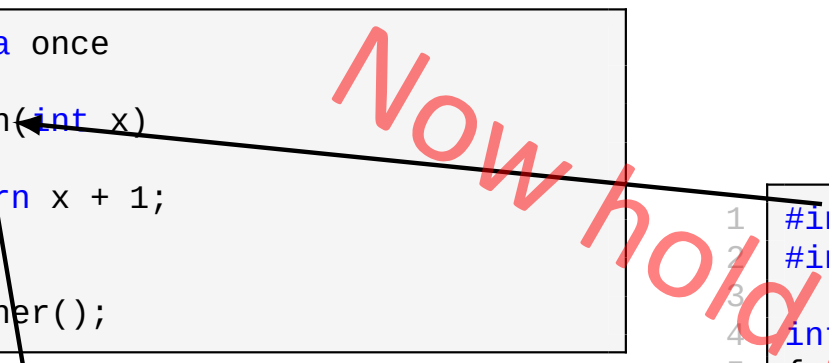
```
1  #pragma once
2
3  int fun(int x)
4  {
5    return x + 1;
6  }
7
8  int other();
```

file.cc

```
1  #include "file.h"
2
3  int other()
4  {
5    return fun(1);
6  }
```

file.cc

```
1  #include "file.h"
2  #include <iostream>
3
4  int main()
5  {
6    std::cout << other() << std::endl;
7    std::cout << fun(3) << std::endl;
8  }
```



One Definition Rule (ODR)

Why does this work?

file.h

```
1  #pragma once
2
3  struct Cls
4  {
5    int fun(int x)
6    {
7      return x + 1;
8    }
9  };
```

file.cc

```
1  #include "file.h"
2
3  // some code that uses Cls::fun
```

file.cc

```
1  #include "file.h"
2  #include <iostream>
3
4  int main()
5  {
6    Cls c { };
7    std::cout << c.fun(3) << std::endl;
8  }
```

One Definition Rule (ODR)

Why does this work?

file.h

```
1  #pragma once
2
3  struct Cls
4  {
5    int fun(int x)
6    {
7      return x + 1;
8    }
9  };
```

file.cc

```
1  #include "file.h"
2
3  // some code that uses Cls::fun
```

file.cc

```
1  #include "file.h"
2  #include <iostream>
3
4  int main()
5  {
6    Cls c { };
7    std::cout << c.fun(3) << std::endl;
8  }
```

One Definition Rule (ODR)

Why does this work?

file.h

```
1  #pragma once
2
3  template <typename T>
4  T fun(T x)
5  {
6    return x + 1;
7  }
```

file.cc

```
1  #include "file.h"
2
3  // some code that uses Cls::fun
```

file.cc

```
1  #include "file.h"
2  #include <iostream>
3
4  int main()
5  {
6    std::cout << fun(3) << std::endl;
7  }
```

One Definition Rule (ODR)

Why does this work?

file.h

```
1  #pragma once
2
3  template <typename T>
4  T fun(T x)
5  {
6    return x + 1;
7  }
```

file.cc

```
1  #include "file.h"
2
3  // some code that uses Cls::fun
```

file.cc

```
1  #include "file.h"
2  #include <iostream>
3
4  int main()
5  {
6    std::cout << fun(3) << std::endl;
7  }
```

Why does this work?!

One Definition Rule (ODR)

Definition

- Entities[1] may only be defined once per translation unit

[1] classes, enums, functions, variables or templates

One Definition Rule (ODR)

Definition

- Entities[1] may only be defined once per translation unit
- For an entity to be used it must first be defined

[1] classes, enums, functions, variables or templates

One Definition Rule (ODR)

Definition

- Entities[1] may only be defined once per translation unit
- For an entity to be used it must first be defined
- Defining a non-inlined entity across multiple translation units is ill-formed

[1] classes, enums, functions, variables or templates

One Definition Rule (ODR)

Definition

- Entities[1] may only be defined once per translation unit
- For an entity to be used it must first be defined
- Defining a non-inlined entity across multiple translation units is ill-formed

[1] classes, enums, functions, variables or templates

One Definition Rule (ODR)

Discussion on static members

- The ODR is the reason behind the issues we have seen so far
- The program is ill-formed if there are multiple definitions of the same entity
- However, the ODR rule does give an exception to this, namely *inlined* entites
- What does that mean? Could that explain why member functions and function templates doesn't have the same problems?

One Definition Rule (ODR)

`inline` – Exceptions to the ODR

```
1  inline int fun(int x)
2  {
3    return x + 1;
4  }
5
6  struct Cls
7  {
8    static inline int y { 5 };
9  };
10
11 inline int z { 2 };
```

One Definition Rule (ODR)

`inline` – Exceptions to the ODR

- The `inline` keyword produces inlined entites which are exempted from the ODR
- What this means in particular is that there may be multiple definitions of the same entity across multiple translation units
- All member functions defined inside class definitions and all templates are *implicitly* marked as `inline`
- The compiler assumes that all of the definitions are *exactly* the same, meaning it is undefined behaviour if their definitions differ between translation units

One Definition Rule (ODR)

Undefined behaviour

TU #1

```
1 inline int fun(int x)
2 {
3   return x + 1;
4 }
```

TU #2

```
1 inline int fun(int x)
2 {
3   return x + 2;
4 }
```

One Definition Rule (ODR)

Undefined behaviour

TU #1

```
1 inline int fun(int x)
2 {
3   return x + 1;
4 }
```

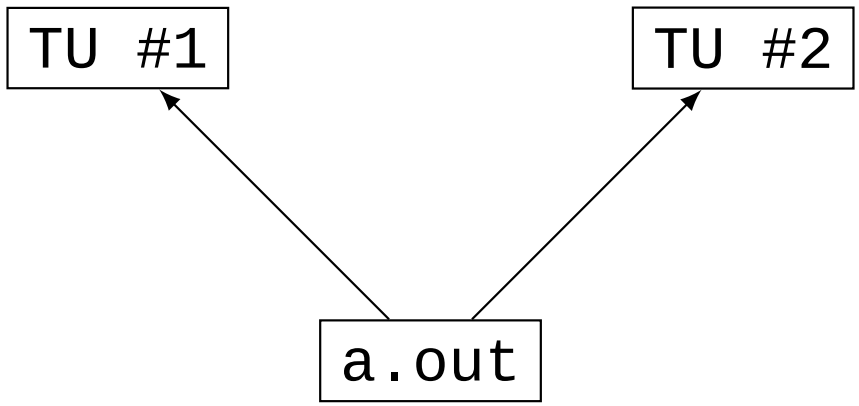
TU #2

```
1 inline int fun(int x)
2 {
3   return x + 2;
4 }
```

Undefined behaviour!

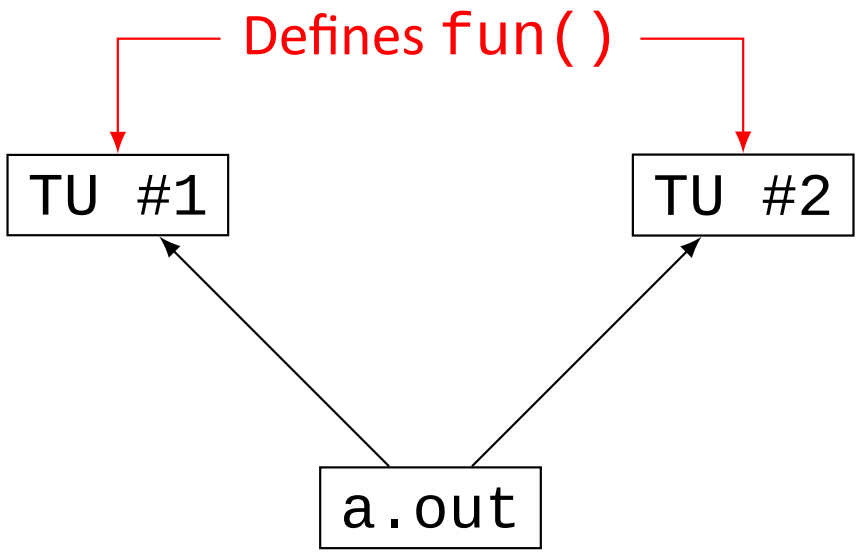
One Definition Rule (ODR)

During linking



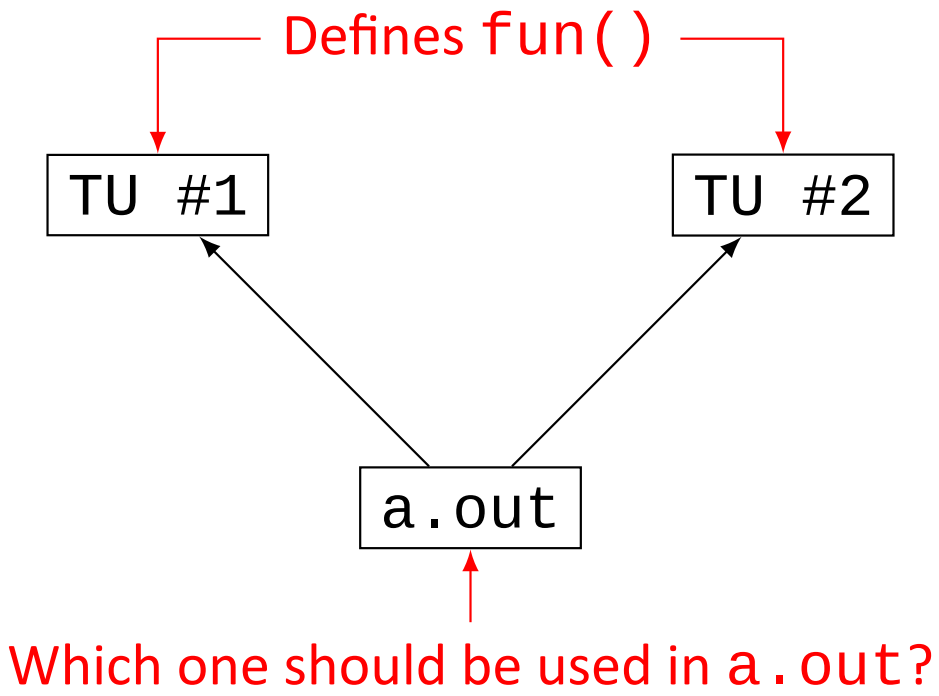
One Definition Rule (ODR)

During linking



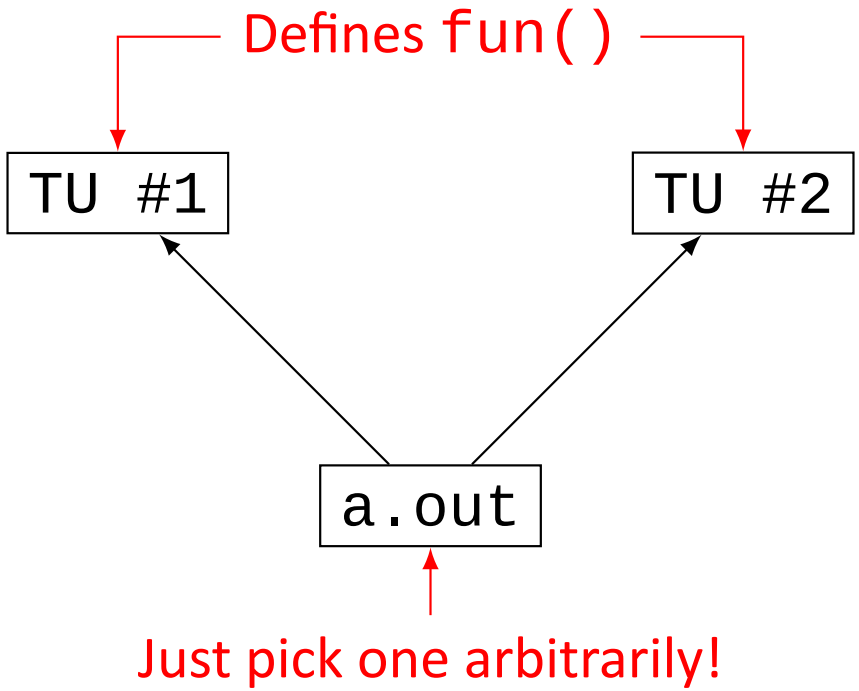
One Definition Rule (ODR)

During linking



One Definition Rule (ODR)

During linking



One Definition Rule (ODR)

During linking

- If there are multiple definitions of the same inlined entity during *linking*, then the linker must pick *one* of them to serve as the global definition
- It is arbitrary which one the linker will pick, it can vary between compilations even
- Therefore all of them must be *exactly* the same, otherwise you will run into inconsistent behaviours (as seen previously)

One Definition Rule (ODR)

Common gotcha

file.h

```
1 #pragma one
2 int global;
```

file.cc

```
1 #include "file.h"
2 int global { 3 };
```

One Definition Rule (ODR)

Common gotcha

Doesn't compile...

file.h

```
1 #pragma one
2 int global;
```

file.cc

```
1 #include "file.h"
2 int global { 3 };
```

One Definition Rule (ODR)

Common gotcha

- A common mistake one does when trying to define global variables is to try the same thing as we do with static members
- I.e. to only declare it in the header and then defining it in the implementation file
- However, note that `int global;` is still a *definition*, where we leave the value uninitialized (it still *creates* the variable)
- This means that we are trying to create multiple (conflicting) definitions
- How do we resolve this?

One Definition Rule (ODR)

Bad solution

file.h

```
1 #pragma one
2 inline int global { 3 };
```

file.cc

```
1 #include "file.h"
```

One Definition Rule (ODR)

Bad solution

- We could resolve this by making the global variable inline
- But this forces each translation unit which uses the variable to locally define it (causing work to be done in the compilation step)
- These locally defined definitions will then be thrown out in favor of one arbitrarily selected one
- This is the same reason why we want to keep any definitions out of header files if possible

One Definition Rule (ODR)

Better solution

file.h

```
1 #pragma one
2 // declare that the variable exists
3 extern int global;
```

file.cc

```
1 #include "file.h"
2 // define it in ONE translation unit
3 int global { 3 };
```

One Definition Rule (ODR)

`extern`

- The `extern` keyword is mainly used to tell the compiler that the definition of an entity can be found somewhere in *one* translation unit.
- This is how we create global variables with *one* definition
- It has some other usages as well (in particular when interfacing C++ and C code with each other (not covered in this course))

One Definition Rule (ODR)

A **HUGE** problem with the compilation model

TU #1

```
1  // helper function
2  void fun()
3  {
4    std::cout << "A\n";
5  }
6
7  int main()
8  {
9    fun();
10 }
```

TU #2

```
1  // helper function
2  void fun()
3  {
4    std::cout << "B\n";
5  }
6
7
8  // ...
9  // Some other stuff
10 // ...
```

One Definition Rule (ODR)

A **HUGE** problem with the compilation model

TU #1

```
1  // helper function
2  void fun()
3  {
4    std::cout << "A\n";
5  }
6
7  int main()
8  {
9    fun();
10 }
```

TU #2

```
1  // helper function
2  void fun()
3  {
4    std::cout << "B\n";
5  }
6
7
8  // ...
9  // Some other stuff
10 // ...
```

Doesn't compile...

One Definition Rule (ODR)

A **HUGE** problem with the compilation model

- Header files are not really special, their contents are usually just copy-pasted into translation units
- What this means is that during compilation and linking the compiler/linker does not rely on header files to know what should and should not be defined
- Instead, each time an entity is declared the compiler and linker assumes that this entity should be globally available
- Because of this, if we have two separate entites which happens to have the exact same signature (as seen in the example on the previous slide) the compile assumes they are *meant* to be the same function
- This causes a linker error since there seemingly are multiple definitions of the same entity
- This can cause issues (in particular in large code bases)
- Because of this we would like to be able to define entities that are *truly* only available within singular translation units

One Definition Rule (ODR)

Global entities bound to *one* translation unit

```
1  // this variable is not exposed to other
2  // translation units, it only exists here
3  static int x { 3 };
4
5  // everything inside an anonymous namespace
6  // only exists within the translation unit
7  namespace
8  {
9    int y { 5 };
10   void fun() { /* ... */ }
11 }
```

One Definition Rule (ODR)

Global variables bound to *one* translation unit

- An entity defined in the global scope (or inside a namespace) which is marked `static` is an entity that is not exposed what-so-ever to any other translation units, it is truly local to this file and this file only
- The same is true for anything defined inside an anonymous namespace
- This means that the compiler is allowed to perform *any* optimization on it that it chooses to because it does not have to consider any restrictions imposed by the linker
- For helper functions, variables, classes etc. this is recommended since it increases the compilers ability to optimize, and it guarantees that it doesn't clash with names from other translation units

One Definition Rule (ODR)

Resolving the issue

TU #1

```cpp
// helper function
static void fun()
{
  std::cout << "A\n";
}

int main()
{
  fun();
}
```

TU #2

```cpp
// helper function
static void fun()
{
  std::cout << "B\n";
}


// ...
// Some other stuff
// ...
```

One Definition Rule (ODR)

Resolving the issue

TU #1

```
1  // helper function
2  static void fun()
3  {
4    std::cout << "A\n";
5  }
6
7  int main()
8  {
9    fun();
10 }
```

TU #2

```
1  // helper function
2  static void fun()
3  {
4    std::cout << "B\n";
5  }
6
7
8  // ...
9  // Some other stuff
10 // ...
```

No problems!

One Definition Rule (ODR)

Why is this a **bad** idea?

TU #1

```
1  // helper function
2  inline void fun()
3  {
4    std::cout << "A\n";
5  }
6
7  int main()
8  {
9    fun();
10 }
```

TU #2

```
1  // helper function
2  inline void fun()
3  {
4    std::cout << "B\n";
5  }
6
7
8  // ...
9  // Some other stuff
10 // ...
```

One Definition Rule (ODR)

Common misconception

- *function inlining* $\neq$ `inline`

One Definition Rule (ODR)

Common misconception

- *function inlining* $\neq$ `inline`
- `static` *allows* for *function inlining*

One Definition Rule (ODR)

Common misconception

- *function inlining* $\neq$ `inline`
- `static` *allows* for *function inlining*
- No guarantees though

One Definition Rule (ODR)

Static Initialization Order Fiasco

- within translation unit: well defined

One Definition Rule (ODR)

Static Initialization Order Fiasco

- within translation unit: well defined
- between translation units: ambigous

One Definition Rule (ODR)

Static Initialization Order Fiasco

- within translation unit: well defined
- between translation units: ambigous
- global streams are always initialized *first*

One Definition Rule (ODR)

Static Initialization Order Fiasco

- within translation unit: well defined
- between translation units: ambigous
- global streams are always initialized *first*
- https://en.cppreference.com/w/cpp/language/siof.html

One Definition Rule (ODR)

Static Initialization Order Fiasco

- within translation unit: well defined
- between translation units: ambigous
- global streams are always initialized *first*
- `https://en.cppreference.com/w/cpp/language/siof.html`
- **DO NOT** assume anything during static initialization

Summary

Object overview

Each object has a:

- storage duration (dynamic, automatic or static)
- size (number of bytes)
- address (memory address of first byte)
- alignment (required address divisibility)

Summary

`static`

`static` can be used in several situations:

- On entites declared in a namespace: ignore ODR
- On variables declared in classes: static member
- On variables in functions: static local variable

Summary

ODR

- `inline` – Ignore ODR
- `static` – Translation-unit-local
- `extern` – definition in translation unit

Summary

Lifetime – creation

Initialization happens during:

- program startup (global variables)
- Variable definition (local variables)
- First call to function (static local variables)
- call to `operator new( )` (dynamic variables)

Summary

Lifetime – destruction

Destruction happens during:

- program termination (global and static local variables)
- Scope exit (local variables)
- call to `operator delete( )` (dynamic variables)

www.liu.se

LINKÖPING UNIVERSITY