(3) C++ Abstractions

Nico Ludwig (@ersatzteilchen)

TOC

- Sources:
  - Bruce Eckel, Thinking in C++ Vol I
  - Bjarne Stroustrup, The C++ Programming Language

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!

Instances of UDTs are passed by Value

- As we learned, in C++ objects are getting passed by value to/from functions.
  - This is also true for instances of UDTs.
- We know that we can pass data by reference with pointers.
  - It was required to do so, e.g. modify values of a variable passed to a function.
- However, copying instances of a UDT can be costly, as all fields get copied.
- Avoiding to copy instances of UDTs is another strength of C++.
- C++ provides means to avoid copies of UDT-instances.
  - We have to revisit dynamic creation of objects.

Creation of UDTs on the Heap

- It's possible and often needed to create instances of UDTs on the heap like so:

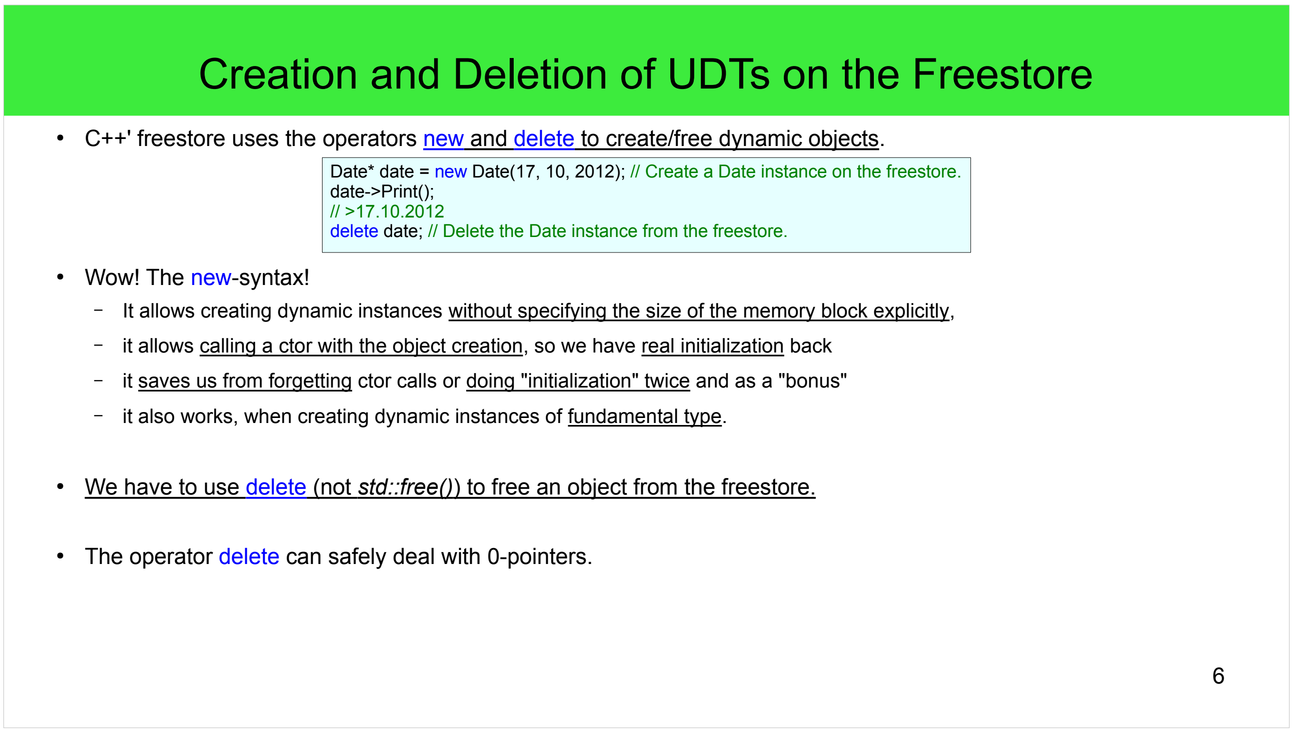
```
Date* date = static_cast<Date*>(std::malloc(sizeof(Date))); // No surprise...
if (date) {
        date->SetDay(17); date->SetMonth(10); date->SetYear(2012);
        date->Print();
        // >17.10.2012
        std::free(date);
}
```

- That's boring! It's also cumbersome and dangerous!

- Minor problems:
  - We have to do the calculation of the size of the memory-block manually.

- Major and dangerous problems:
  - We can not call ctors during creation, instead we have to initialize explicitly!
    - Dangers w/o ctors: forgetting to initialize, doing it twice or with erroneous values... ouch!

- Is there a better way? Yes, the C++ freestore.



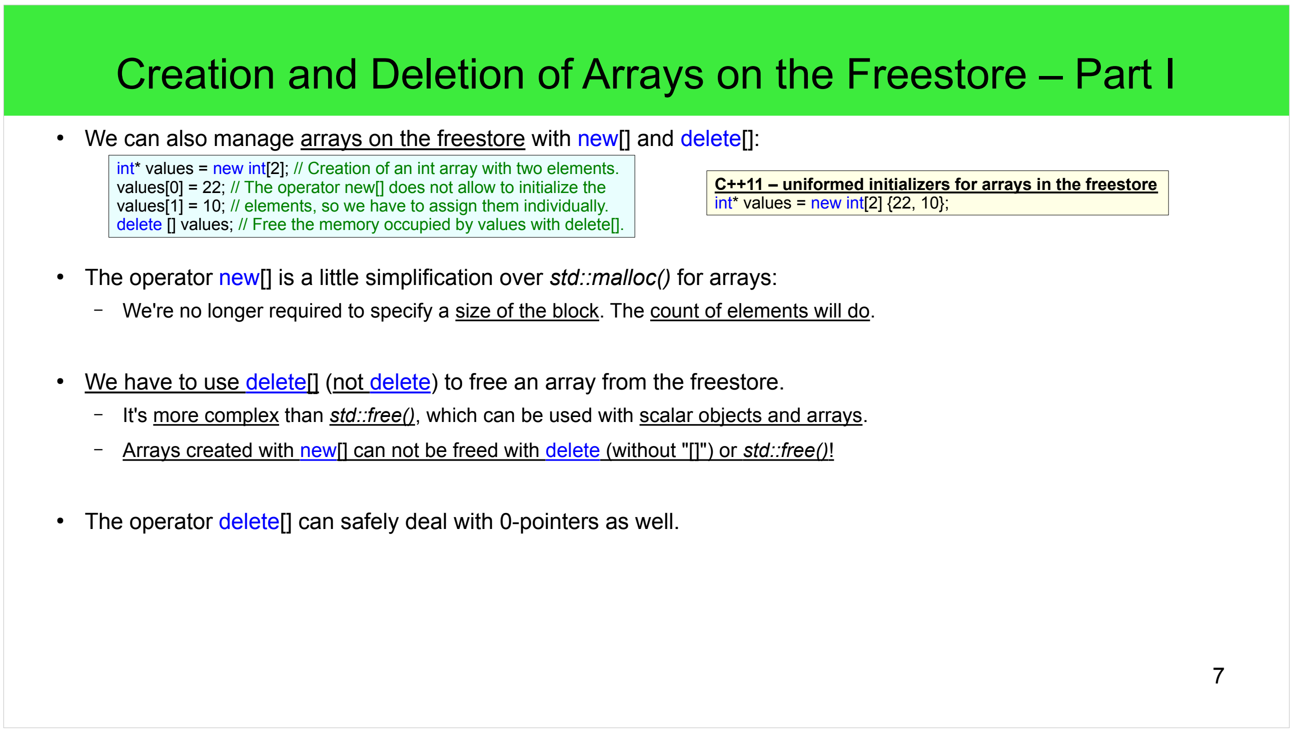
Creation and Deletion of UDTs on the Freestore

- C++' freestore uses the operators new and delete to create/free dynamic objects.

```
Date* date = new Date(17, 10, 2012); // Create a Date instance on the freestore.
date->Print();
// >17.10.2012
delete date; // Delete the Date instance from the freestore.
```

- Wow! The new-syntax!
  - It allows creating dynamic instances without specifying the size of the memory block explicitly,
  - it allows calling a ctor with the object creation, so we have real initialization back
  - it saves us from forgetting ctor calls or doing "initialization" twice and as a "bonus"
  - it also works, when creating dynamic instances of fundamental type.
- We have to use delete (not *std::free()*) to free an object from the freestore.
- The operator delete can safely deal with 0-pointers.

- In this course we make a distinction between heap and freestore. This distinction is not defined by the C++ standard. Instead it is used colloquially to separate the memory managed by *std::malloc()*/*std::free()* and new/delete, because they are assumed to be operating in different memory locations each.
- The implementation of the freestore can be based on the heap implementation, but not vice versa. This is a hint at the performance to be awaited: a faster implementation cannot be based on a slower one.



Creation and Deletion of Arrays on the Freestore – Part I

- We can also manage arrays on the freestore with new[] and delete[]:

```
int* values = new int[2]; // Creation of an int array with two elements.
values[0] = 22; // The operator new[] does not allow to initialize the
values[1] = 10; // elements, so we have to assign them individually.
delete [] values; // Free the memory occupied by values with delete[].
```

C++11 – uniformed initializers for arrays in the freestore

```
int* values = new int[2] {22, 10};
```

- The operator new[] is a little simplification over *std::malloc()* for arrays:
  - We're no longer required to specify a size of the block. The count of elements will do.
- We have to use delete[] (not delete) to free an array from the freestore.
  - It's more complex than *std::free()*, which can be used with scalar objects and arrays.
  - Arrays created with new[] can not be freed with delete (without "[]") or *std::free()*!
- The operator delete[] can safely deal with 0-pointers as well.

- One way handling arrays in the freestore could be implemented like so: When an array is created on the freestore, the freestore-allocator stores size information about the array (block size and count of elements) somewhere near the memory location of the array (usually just in front of its start address). This information is evaluated, when delete[] is called, to call the dtors of each element held in the array. How the size information is handled is not defined in C++, so it cannot be legally read or interpreted by the programmer. It also explains, how using delete[] on a non-array leads to undefined behavior (crash or corrupted freestore), because the required size information is not present or will be misinterpreted. In early versions of C++, it was required to pass the length of the array to delete[] explicitly as *delete[size]*.

Creation and Deletion of Arrays on the Freestore – Part II

- This code contains an error, that is not simple to detect:

```
const size_t length = 5;
char** cstringArray = static_cast<char**>( std::malloc(sizeof(char) * length ));

if (cstringArray) {
        for (int i = 0; i < length; ++i) {
            const int nameLength = 256;
            char* yourName = static_cast<char*>(std::malloc(sizeof(char) * nameLength));
            std::cout<<"Please enter your name (max. "<<nameLength - 1<<" characters!):"<<std::endl;
            std::cin.getline(yourName, nameLength);
            cstringArray[i] = yourName;
        }
        for (int i = 0; i < length; ++i) {
            std::cout<<cstringArray[i]<<std::endl;
            std::free(cstringArray[i]);
        }
        std::free(cstringArray);
}
```

- The problem: *cstringArray*'s size is calculated incorrectly! The behavior of this code is undefined at run time!
  - The calculation of the required size is error prone even in simple cases: Hey, we just want to create an array of cstrings!
  - The correct calculation must multiply the sizeof(char*) with the length of the desired cstring array, not only sizeof(char):

```
char** cstringArray = static_cast<char**>(std::malloc(sizeof(char*) * length)); // Ok!
```

- Now, we'll reformulate this code using the freestore with new[]/delete[].

Creation and Deletion of Arrays on the Freestore – Part III

- Esp. with new[], we have literally no chance to doing it wrong creating a cstring-array:

```
const size_t length = 5;
char** cstringArray = new char*[length];

for (int i = 0; i < length; ++i) {
        const int nameLength = 256;
        char* yourName = new char[nameLength];
        std::cout<<"Please enter your name (max. "<<nameLength - 1<<" characters!):"<<std::endl;
        std::cin.getline(yourName, nameLength);
        cstringArray[i] = yourName;
}
for (int i = 0; i < length; ++i) {
        std::cout<<cstringArray[i]<<std::endl;
        delete[] cstringArray[i];
}
delete[] cstringArray;
```

```
const size_t length = 5;
char** cstringArray = static_cast<char**>(std::malloc(sizeof(char) * length));
if (cstringArray) {
        for (int i = 0; i < length; ++i) {
            const int nameLength = 256;
            char* yourName = static_cast<char*>(std::malloc(sizeof(char) * nameLength));
            std::cout<<"Please enter your name (max. "<<nameLength - 1<<" characters!):"<<std::endl;
            std::cin.getline(yourName, nameLength);
            cstringArray[i] = yourName;
        }
        for (int i = 0; i < length; ++i) {
            std::cout<<cstringArray[i]<<std::endl;
            std::free(cstringArray[i]);
        }
        std::free(cstringArray);
}
```

- We cannot even formulate a wrong calculation:

```
char** cstringArray = new char[length]; // Invalid! Cannot initialize a variable of type 'char **' with an rvalue of type 'char *'
```

- The solution: There is no need to calculate something! The new[] operator handles everything for us.
  - We have replaced using the heap by using the freestore all over: the simplification in the code is amazing!

What if new or new[] fail?

- If new or new[] fail, they will throw an *std::bad_alloc* exception. – Huh?

- Exceptions are an advanced way to signal and handle run time errors.
  - Exceptions can be caught and handled, or they can be ignored.

- A fully ignored exception, won't be handled and leads to immediate program exit.

- => Let's make a contract:
  - We will not handle *std::bad_alloc* exceptions thrown by new.
    - If new fails, let the program exit immediately. That's the best we can do it for now.
  - We will not check the result of new against 0!

- (The option *new(std::nothrow)* returns a 0-pointer on failure instead of throwing.)
  - (We are not going to use the option *std::nothrow* for now!)

Excursus: Placement-new

- C++ allows creating objects with new on a non-freestore buffer:
  - 1. The operator new together with calling a ctor initializes the object.
  - 2. The passed pointer to a buffer is used as location of the object.
  - => This is called placement.

- Example: creating an instance of a UDT with new on the heap.

```
void* buffer = std::malloc(sizeof(Date)); // Create a blank buffer in the heap.
// Create a Date instance with a ctor and place it on that blank buffer in the heap:
Date* date = new (buffer) Date(17, 10, 2012);
std::free(buffer); // Free date's buffer from the heap.
```

  - The C++ placement-new syntax is simple to use:
    - 1. #include <new>.
    - 2. Allocate a buffer (from any memory) to place the to-be-created object in.
    - 3. Pass a pointer to the allocated buffer to placement-new.
    - 4. The buffer needs to be freed congruently (*std::malloc()* → *std::free()*) with how it was created.

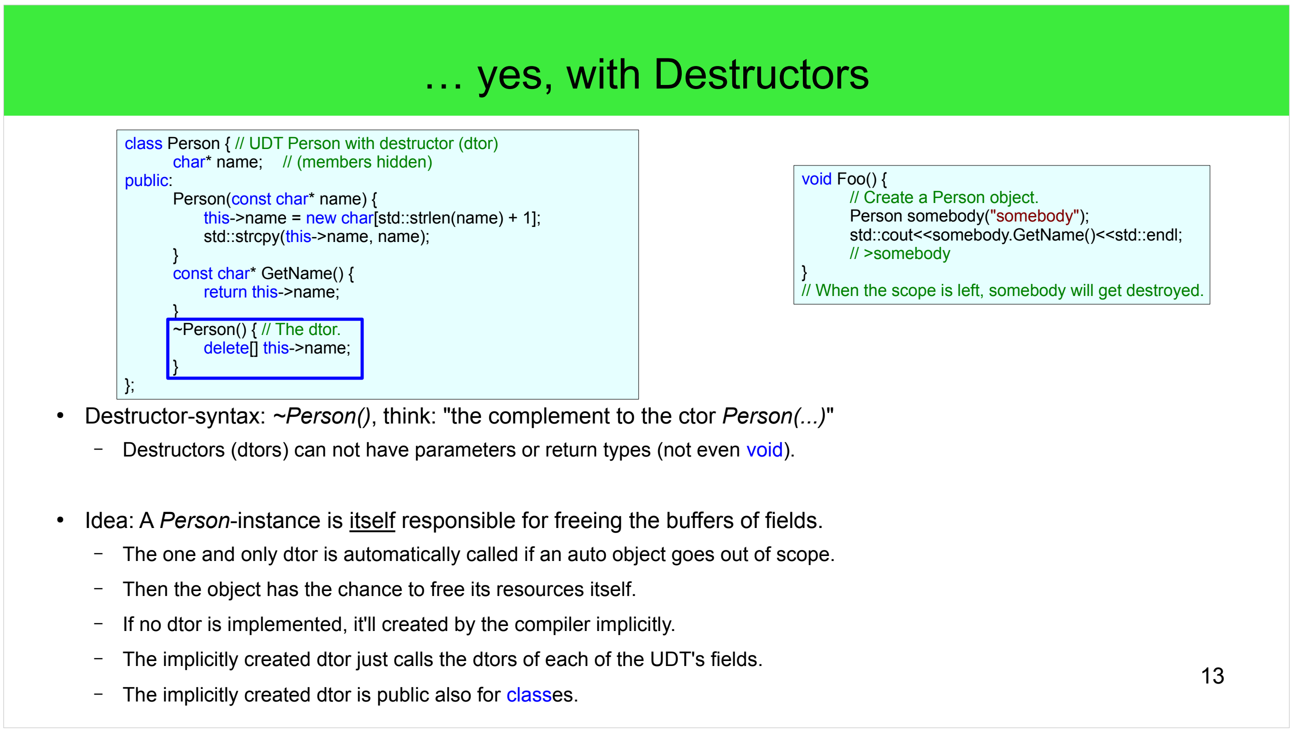
- Placement-new is often used with special 3rd party buffers.

A Problem: Freeing dynamic Fields in UDT Instances

```
class Person { // Another UDT Person
        char* name;
public:
        Person(const char* name) {
             this->name = new char[std::strlen(name) + 1];
             std::strcpy(this->name, name);
        }
        const char* GetName() {
             return this->name;
        }
};
```

```
void Foo() {
        // Create a Person object.
        Person somebody("somebody");
        std::cout<<somebody.GetName()<<std::endl;
        // >somebody
        // But... when to free Person::name from
        // freestore like so?:
        delete[] somebody.GetName(); // Really?
        // Freed it twice... oops!
        delete[] somebody.GetName();
}
```

- Hm... What's the problem with the UDT *Person*?
  - When we create a *Person* object, a buffer for *name* needs to be created dynamically.
  - As we learned we are responsible for freeing dynamically created memory.
  - But who is responsible? The user/creator of a *Person*-instance?
  - When should the user free *name*? And how?
  - Freeing could be done explicitly as shown in the example above.
  - ...but then freeing could be forgotten or done twice. Is there a better solution?



… yes, with Destructors

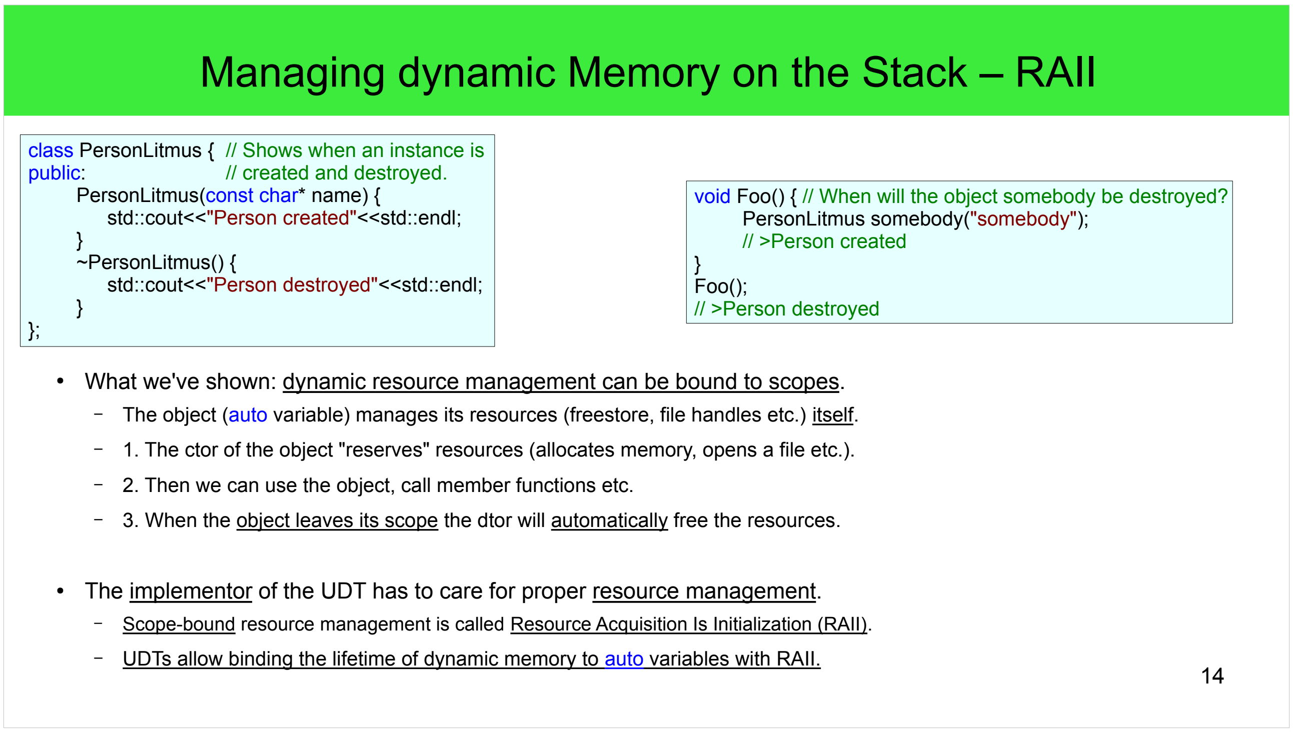
```
class Person { // UDT Person with destructor (dtor)
	char* name;	// (members hidden)
public:
	Person(const char* name) {
		this->name = new char[std::strlen(name) + 1];
		std::strcpy(this->name, name);
	}
	const char* GetName() {
		return this->name;
	}
	~Person() { // The dtor.
		delete[] this->name;
	}
};
```

```
void Foo() {
	// Create a Person object.
	Person somebody("somebody");
	std::cout<<somebody.GetName()<<std::endl;
	// >somebody
}
// When the scope is left, somebody will get destroyed.
```

- Destructor-syntax: *~Person()*, think: "the complement to the ctor *Person(...)*"
  - Destructors (dtors) can not have parameters or return types (not even void).
- Idea: A *Person*-instance is <u>itself</u> responsible for freeing the buffers of fields.
  - The one and only dtor is automatically called if an auto object goes out of scope.
  - Then the object has the chance to free its resources itself.
  - If no dtor is implemented, it'll created by the compiler implicitly.
  - The implicitly created dtor just calls the dtors of each of the UDT's fields.
  - The implicitly created dtor is public also for classes.

13

- Dtor syntax in short:
  - The dtor's name must be exactly the name of the UDT with the '~'-prefix.
  - We can have only one dtor, i.e. no overloads are allowed.



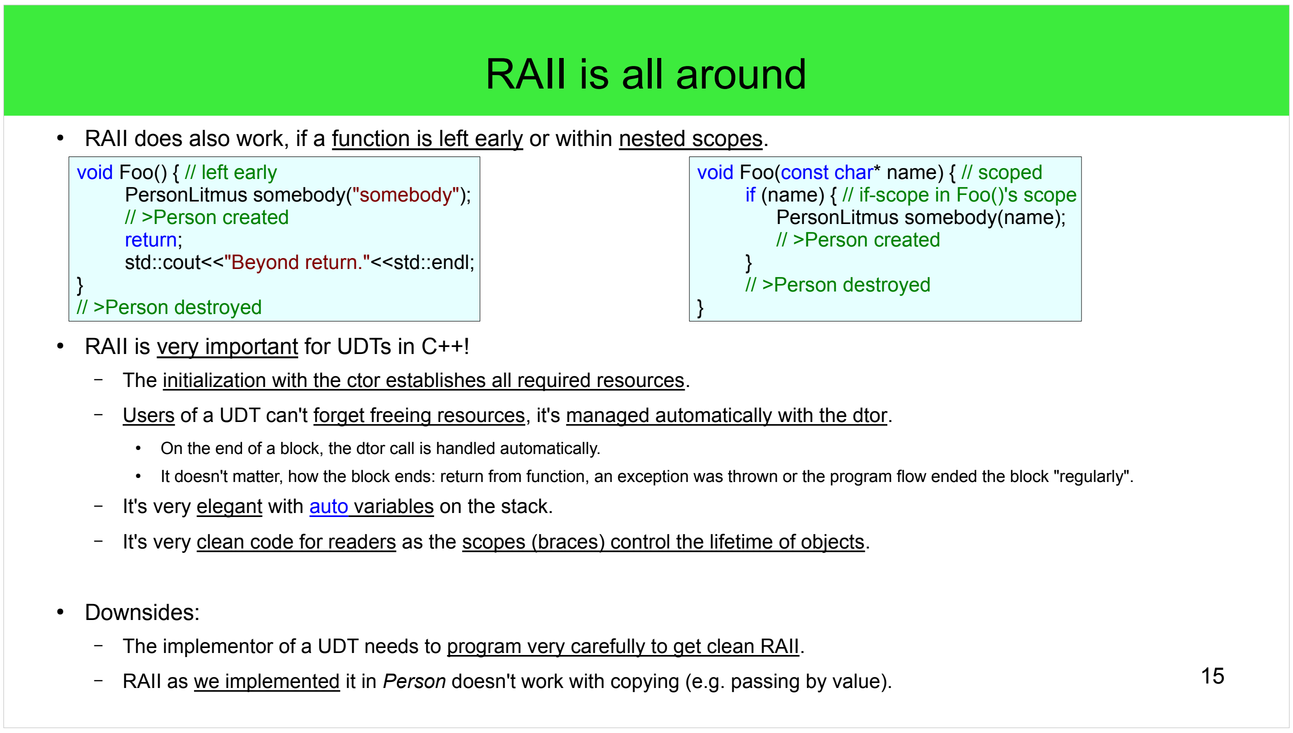
Managing dynamic Memory on the Stack – RAII

```
class PersonLitmus {  // Shows when an instance is
public:               // created and destroyed.
    PersonLitmus(const char* name) {
        std::cout<<"Person created"<<std::endl;
    }
    ~PersonLitmus() {
        std::cout<<"Person destroyed"<<std::endl;
    }
};
```

```
void Foo() { // When will the object somebody be destroyed?
    PersonLitmus somebody("somebody");
    // >Person created
}
Foo();
// >Person destroyed
```

- What we've shown: dynamic resource management can be bound to scopes.
  - The object (auto variable) manages its resources (freestore, file handles etc.) itself.
  - 1. The ctor of the object "reserves" resources (allocates memory, opens a file etc.).
  - 2. Then we can use the object, call member functions etc.
  - 3. When the object leaves its scope the dtor will automatically free the resources.
- The implementor of the UDT has to care for proper resource management.
  - Scope-bound resource management is called Resource Acquisition Is Initialization (RAII).
  - UDTs allow binding the lifetime of dynamic memory to auto variables with RAII.

- Using additional output in functions (e.g. on the console or into log files) is called tracing. Tracing is an important means to debug code as it is the next level of "*printf()*-debugging". And it is useful to understand what our code does...
- RAII allows us to view auto objects from a different perspective: scopes not only control the visibility of auto variables, but also the lifetime of objects.
- In this case the *PersonLitmus* object is stored in an auto variable. So its lifetime is controlled by the stack, where scopes control the lifetime of objects in C++. On the end of the scope the *PersonLitmus* object will be popped of the stack automatically and the dtor will also be called automatically.
- The lifetime of autos also ends, if their scope is left with a return statement or throwing an exception.
- If we create an auto *PersonLitmus*-array of n items, we'll see n ctor calls and n dtor calls respectively.



RAII is all around

- RAII does also work, if a function is left early or within nested scopes.

```
void Foo() { // left early
        PersonLitmus somebody("somebody");
        // >Person created
        return;
        std::cout<<"Beyond return."<<std::endl;
}
// >Person destroyed
```

```
void Foo(const char* name) { // scoped
        if (name) { // if-scope in Foo()'s scope
            PersonLitmus somebody(name);
            // >Person created
        }
        // >Person destroyed
}
```

- RAII is very important for UDTs in C++!
  - The initialization with the ctor establishes all required resources.
  - Users of a UDT can't forget freeing resources, it's managed automatically with the dtor.
    - On the end of a block, the dtor call is handled automatically.
    - It doesn't matter, how the block ends: return from function, an exception was thrown or the program flow ended the block "regularly".
  - It's very elegant with auto variables on the stack.
  - It's very clean code for readers as the scopes (braces) control the lifetime of objects.
- Downsides:
  - The implementor of a UDT needs to program very carefully to get clean RAII.
  - RAII as we implemented it in *Person* doesn't work with copying (e.g. passing by value).

- An example for a very important C++ type using RAII is *std::string*. All further container types in the STL make use of RAII.
- RAII means that a resource that requires e.g. dynamic memory or other operating system resources, will be initialized and freed analogously to the lifetime of a variable. – Practically it means that a resource from the heap can be controlled by a variable on the stack! – This can be implemented with user defined types.

Dynamic Memory and RAII

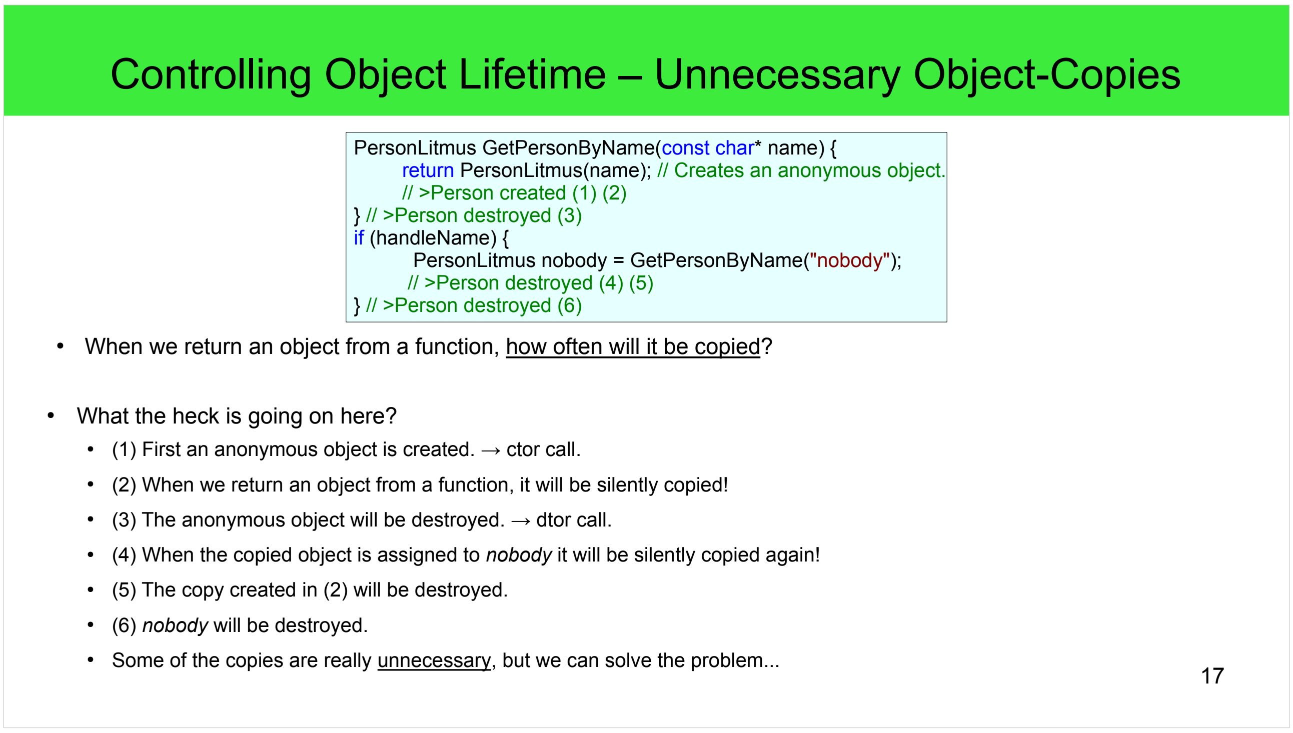
- Again: explicit caring for dynamic memory vs. RAII must be fully understood!

- Explicitly caring for dynamic memory in the scope of an if-statement:

```
if (handleName) { // Explicit - no RAII:
        char* name = new char[5];
        std::strcpy(name, "somebody");
        //... doing some some stuff with name
        delete [] name; // Here explicit deletion is required!
} // The dynamic memory "behind" the pointer name needs to be freed explicitly, because the memory
// to which name points to is not bound to name's lifetime. But name's (i.e. the pointer's) lifetime is
// controlled by the scope that is controlled by the stack.
// => The pointer name does not represent the dynamic memory, it is only a pointer to the memory!
```

- Automatic caring for dynamic memory with RAII in the scope of an if-statement:

```
if (handleName) { // Automatic - with RAII:
        Person somebody("somebody");
        //... doing some some stuff with somebody
} // The dynamic memory, encapsulated by the object somebody, will be automatically freed, when
// somebody leaves its scope and is popped from the stack. On leaving the scope, Person's dtor will
// free the dynamic memory automatically. The lifetime of the dynamic memory is now controlled by
// the stack.
// => The object somebody does itself represent and care for the dynamic memory encapsulated by its
// type.
```



Controlling Object Lifetime – Unnecessary Object-Copies

```
PersonLitmus GetPersonByName(const char* name) {
        return PersonLitmus(name); // Creates an anonymous object.
        // >Person created (1) (2)
} // >Person destroyed (3)
if (handleName) {
          PersonLitmus nobody = GetPersonByName("nobody");
         // >Person destroyed (4) (5)
} // >Person destroyed (6)
```

- When we return an object from a function, <u>how often will it be copied</u>?

- What the heck is going on here?
  - (1) First an anonymous object is created. → ctor call.
  - (2) When we return an object from a function, it will be silently copied!
  - (3) The anonymous object will be destroyed. → dtor call.
  - (4) When the copied object is assigned to *nobody* it will be silently copied again!
  - (5) The copy created in (2) will be destroyed.
  - (6) *nobody* will be destroyed.
  - Some of the copies are really <u>unnecessary</u>, but we can solve the problem...

- By default most C++ compilers are able to create code that suppresses the generation of copies on returning object. – This feature is called <u>Return Value Optimization (RVO)</u>. RVO has been disabled in this example to unleash the full story of copied objects (gcc-option: *-fno-elide-constructors*).
- The shown silent copies are not created with the ctor accepting a const char*, and also mind that there exists no dctor in *PersonLitmus*. The copies are created by the automatically provided "copy constructor". We'll learn about the copy constructor in a future lecture.

Controlling Object Lifetime – Automatic Arrays revisited

- For matters of completeness we'll analyze how arrays work with RAII.

```
void Foo() {
        PersonLitmus persons[] = { PersonLitmus("trish"), PersonLitmus("john") };
        // >Person created
        // >Person created
}
// >Person destroyed // anonymous copies.
// >Person destroyed
// >Person destroyed // the array elements
// >Person destroyed
```

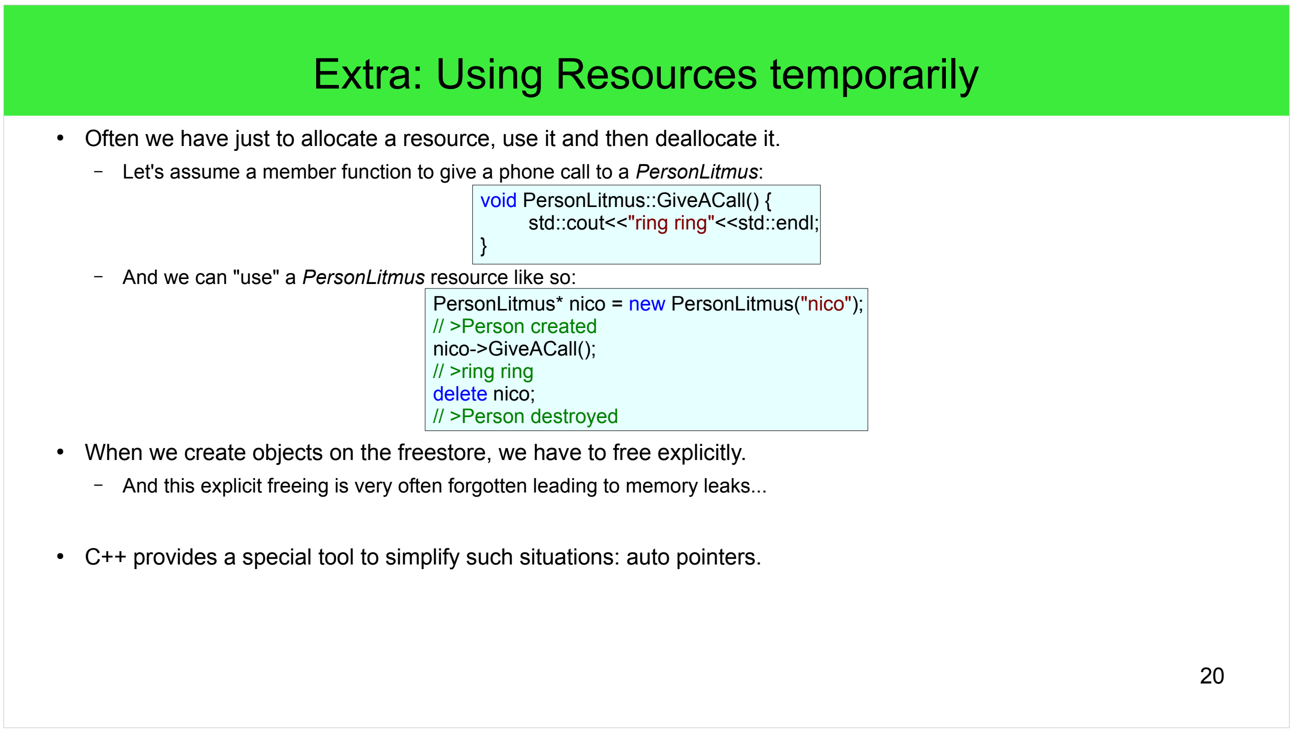
- On destruction, the dtors of all objects in the array will be called.
  - Mind that dtors for the anonymous copies are called as well.

- Therefor we call them automatic arrays (in opposite to dynamic arrays):
  - the lifetime of an automatic array's objects is bound to the array's scope.

Controlling Object Lifetime – Avoiding Object-Copies

- We can get rid of the superfluous copies by using the freestore.
  - The stack manages objects by scope automatically, it also creates the extra copies.
  - But here we as programmers have to reclaim the control of an object's lifetime!
  - So we have to make an objects lifetime independent from the scope of its variable.
  - We learned that we can control the lifetime of objects created in the dynamic memory.
  - => We have to create a *PersonLitmus* in the freestore with the new operator.
    - We will also call the ctor by using the new operator.
  - If the object's lifetime is independent of its scope, when will it be freed?
  - => Of course with delete! Mind that we are responsible for dynamic memory!
    - Calling delete will also call the dtor.

```
PersonLitmus* GetPersonByName(const char* name) {
        return new PersonLitmus(name); // Creates an anonymous object.
        // >Person created
}
if (handleName) {
        PersonLitmus* nobody = GetPersonByName("nobody");
        delete nobody;
} // >Person destroyed
```



Extra: Using Resources temporarily

- Often we have just to allocate a resource, use it and then deallocate it.
  - Let's assume a member function to give a phone call to a *PersonLitmus*:

```
void PersonLitmus::GiveACall() {
        std::cout<<"ring ring"<<std::endl;
}
```

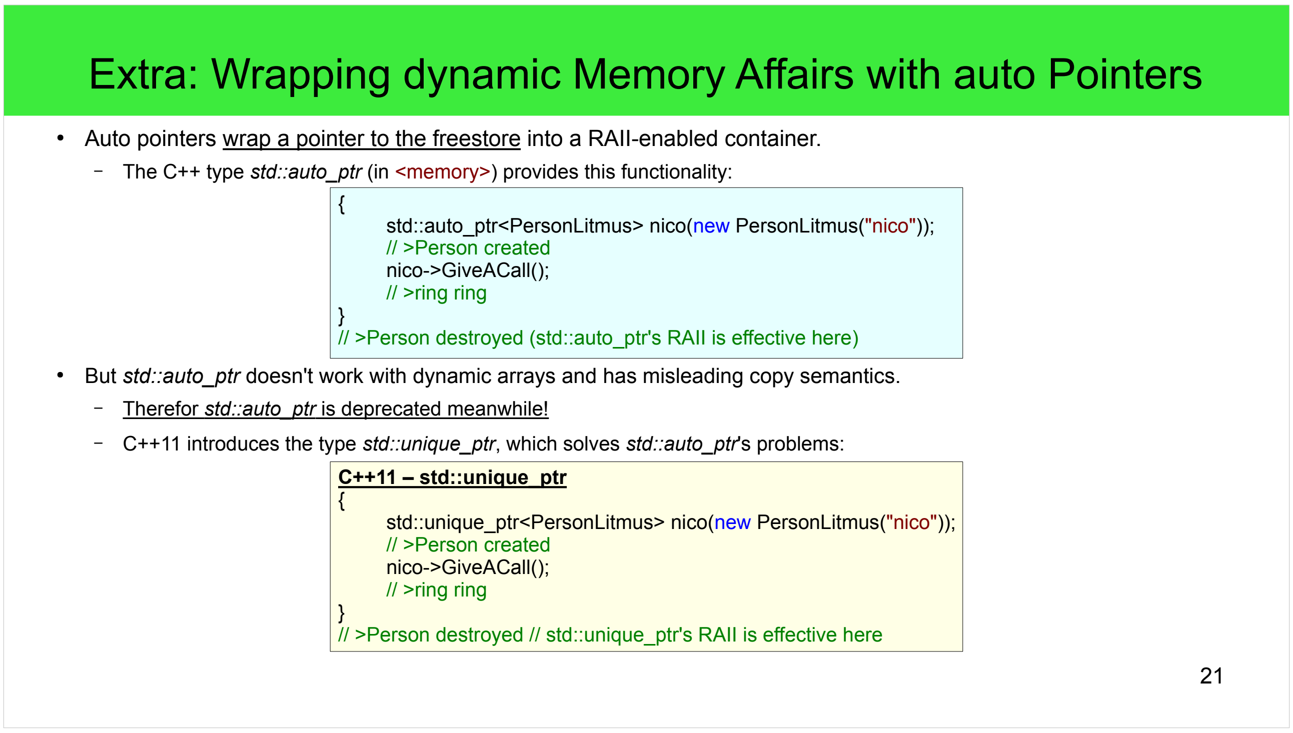
  - And we can "use" a *PersonLitmus* resource like so:

```
PersonLitmus* nico = new PersonLitmus("nico");
// >Person created
nico->GiveACall();
// >ring ring
delete nico;
// >Person destroyed
```

- When we create objects on the freestore, we have to free explicitly.
  - And this explicit freeing is very often forgotten leading to memory leaks...
- C++ provides a special tool to simplify such situations: auto pointers.

20

- In this case we say that a resource is <u>unmanaged</u> (this is virtually a .NET term). "Unmanaged" means that the resource needs to be freed by the programmer, because it is not managed by hardware or software. Examples of such resources are esp. files/file handles and network or database connections.



Extra: Wrapping dynamic Memory Affairs with auto Pointers

- Auto pointers wrap a pointer to the freestore into a RAII-enabled container.
  - The C++ type *std::auto_ptr* (in <memory>) provides this functionality:

```
{
        std::auto_ptr<PersonLitmus> nico(new PersonLitmus("nico"));
        // >Person created
        nico->GiveACall();
        // >ring ring
}
// >Person destroyed (std::auto_ptr's RAII is effective here)
```

- But *std::auto_ptr* doesn't work with dynamic arrays and has misleading copy semantics.
  - Therefor *std::auto_ptr* is deprecated meanwhile!
  - C++11 introduces the type *std::unique_ptr*, which solves *std::auto_ptr*'s problems:

C++11 – std::unique_ptr
```
{
        std::unique_ptr<PersonLitmus> nico(new PersonLitmus("nico"));
        // >Person created
        nico->GiveACall();
        // >ring ring
}
// >Person destroyed // std::unique_ptr's RAII is effective here
```

- Sometimes auto pointers are called smart pointers, but C++' *std::auto_ptr* is not very smart.
- *std::unique_ptr* does also support dynamic arrays created in the freestore.

Summary: Controlling Object Lifetime

- Dealing with objects created on the stack (autos):
  - We've to use "complete" objects on the stack, which will lead to silent copies.
  - We've less control over an object's lifetime, RAII conveniently binds lifetime to scopes.
- Dealing with dynamically created objects:
  - We have to use pointers; function interfaces need to use pointers as well.
    - The objects will not be copied! – We'll only pass around a shortcuts to objects.
  - We have full control over an object's lifetime, but also full responsibility for the lifetime.
  - The operators new and delete conveniently work with ctors and the dtor.
- => C++ programmers decide, whether objects are created on stack or freestore.
  - On many other languages/platforms a type declares, where it is created (e.g. C#/.NET).
- We can also create objects on the heap (*std::malloc()*/placement-new/*std::free()*).
  - But then we have to call construction functions and the dtor explicitly.

- Having a "complete" object on the stack means that all fields of the object will be stored on the function's stack. This is typically the situation, where PODs are used.
- Somewhat strange is the fact, the C++ allows a type to delete its own instance in a member function!

```
class Person {
        int age;
public:
        void Release() {
             delete this;
             // After this is deleted, we can no longer access its
             // instance members.
        }
};
Person* p = new Person;
p->Release(); // This only works, if p was created on the freestore.
```

- This is possible, because delete doesn't really delete the instance represented by this, but the memory occupied by the instance fields of the instance. – Therefor we can no longer access instance fields after this was deleted, e.g. in *Person* we are not allowed to access *age* in *Release()*, after this was deleted.
- Of course, this "technique" only works, if a *Person* instance was created on the freestore.
- This "technique" is often used, when COM coclasses are implemented in C++. In this case, instances of those classes are created in a controlled environment, the so called class factory.

The Lifetime of static/global Objects

- The ctors of global objects are called on the start of the program.
  - In a translation unit (tu) the initialization order is from top to bottom.
  - Between tus the order is not defined, but often it depends on the link order.
    - Link order: the sequence, in which o-files are being passed to the linker.
- The ctors of other static objects, e.g. local statics, can be deferred.
- The dtors of globals and other static objects are called when the programs ends.
  - The dtors of globals are called in the opposite order to initialization in a tu.
  - Between tus the order is not defined, but often it depends on the link order as well.

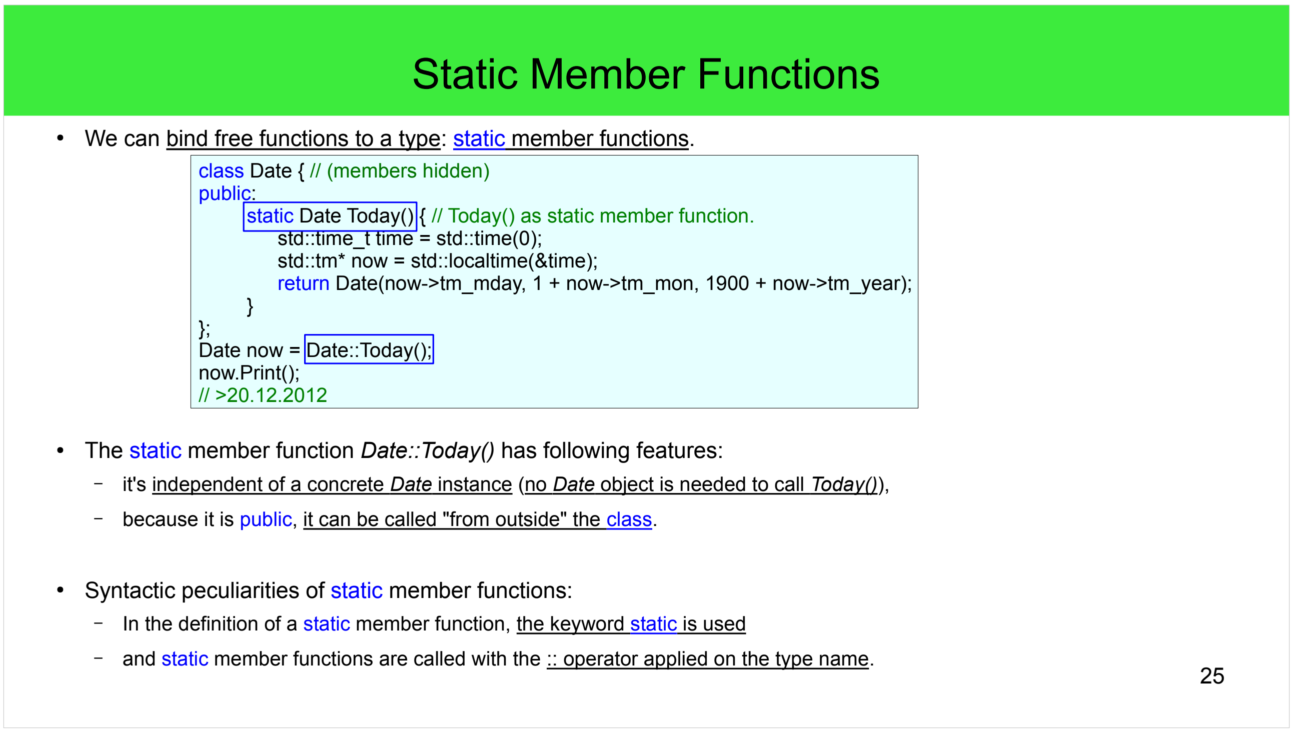
Instance independent Functionality

- Sometimes we need UDT-utility-functions that are independent of an instance.
  - Often such functions end in free functions, i.e. not as member functions, e.g. like so:

```
Date Today() { // Today() returns the current date in a Date object.
        std::time_t t = std::time(0); // Get the current time (API from <ctime>).
        std::tm* now = std::localtime(&t);
        return Date(now->tm_mday, 1 + now->tm_mon, 1900 + now->tm_year);
}
Date now = Today();
now.Print();
// >20.12.2012
```

- Features of the free function *Today()*:
  - It is independent of a concrete instance of the type *Date*.
  - But it creates a new *Date*-instance!

- The idea of having free utility-functions is not bad, but:
  - the belonging to to a type is not given, but *Today()* somehow does depend on *Date*,
  - and we can do better...



Static Member Functions

- We can bind free functions to a type: static member functions.

```
class Date { // (members hidden)
public:
	static Date Today() { // Today() as static member function.
		std::time_t time = std::time(0);
		std::tm* now = std::localtime(&time);
		return Date(now->tm_mday, 1 + now->tm_mon, 1900 + now->tm_year);
	}
};
Date now = Date::Today();
now.Print();
// >20.12.2012
```

- The static member function *Date::Today()* has following features:
  - it's independent of a concrete *Date* instance (no *Date* object is needed to call *Today()*),
  - because it is public, it can be called "from outside" the class.

- Syntactic peculiarities of static member functions:
  - In the definition of a static member function, the keyword static is used
  - and static member functions are called with the :: operator applied on the type name.

- The static keyword must not be repeated on a non-inline definition.

More about static Members

- We can also define static fields in a UDT.
  - (But non-integral, non-constant static fields can not be defined inline!)

```
class Date { // (members hidden)
public: // seconds per gregorian year:
	static const double secondsPerGregorianYear;
};
// The definition of secondsPerGregorianYear has to be non-inline.
const double Date::secondsPerGregorianYear = 31556952.2;

std::cout<<Date::secondsPerGregorianYear<<std::endl;
// >3.1557e+07
```

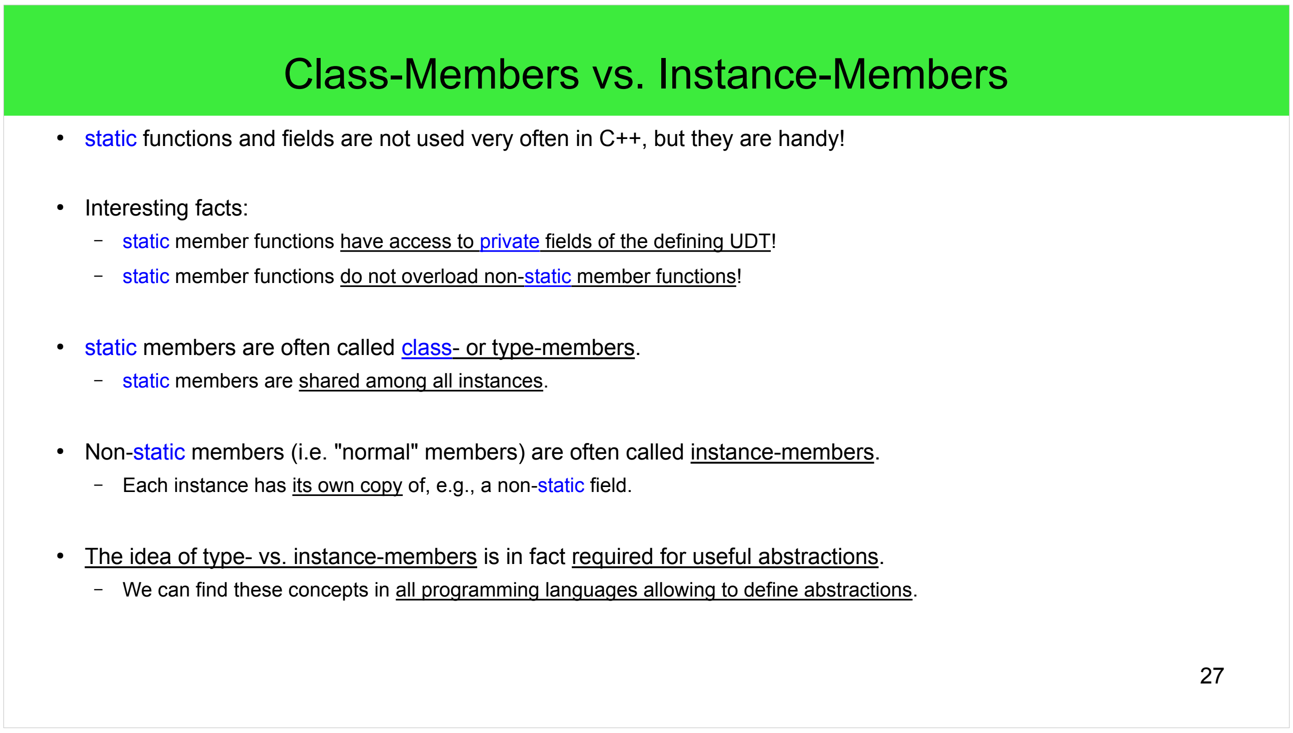
C++11 – in class initialization of constant static fields

```
class Date { // (members hidden)
public:
	constexpr static double secondsPerGregorianYear = 31556952.2;
};
```

- All static (non-private) members can additionally be accessed via instances:

```
Date now = Date::Today();
Date now2 = now.Today(); // Via instance.
Date* pointerToNow = &now;
Date now3 = pointerToNow->Today(); // Via pointer to instance.
```

- Common misunderstandings according static members:
  - They have nothing to do with the static (i.e. global) storage class.
  - We can't have a this-pointer in static member functions, because there's no instance.



Class-Members vs. Instance-Members

- static functions and fields are not used very often in C++, but they are handy!
- Interesting facts:
  - static member functions have access to private fields of the defining UDT!
  - static member functions do not overload non-static member functions!
- static members are often called class- or type-members.
  - static members are shared among all instances.
- Non-static members (i.e. "normal" members) are often called instance-members.
  - Each instance has its own copy of, e.g., a non-static field.
- The idea of type- vs. instance-members is in fact required for useful abstractions.
  - We can find these concepts in all programming languages allowing to define abstractions.

- A static member function can access the private members (esp. the fields) of a passed object that has the same UDT, in which this static member function has been defined. Nevertheless, a static member function has no this-pointer!

Thank you!