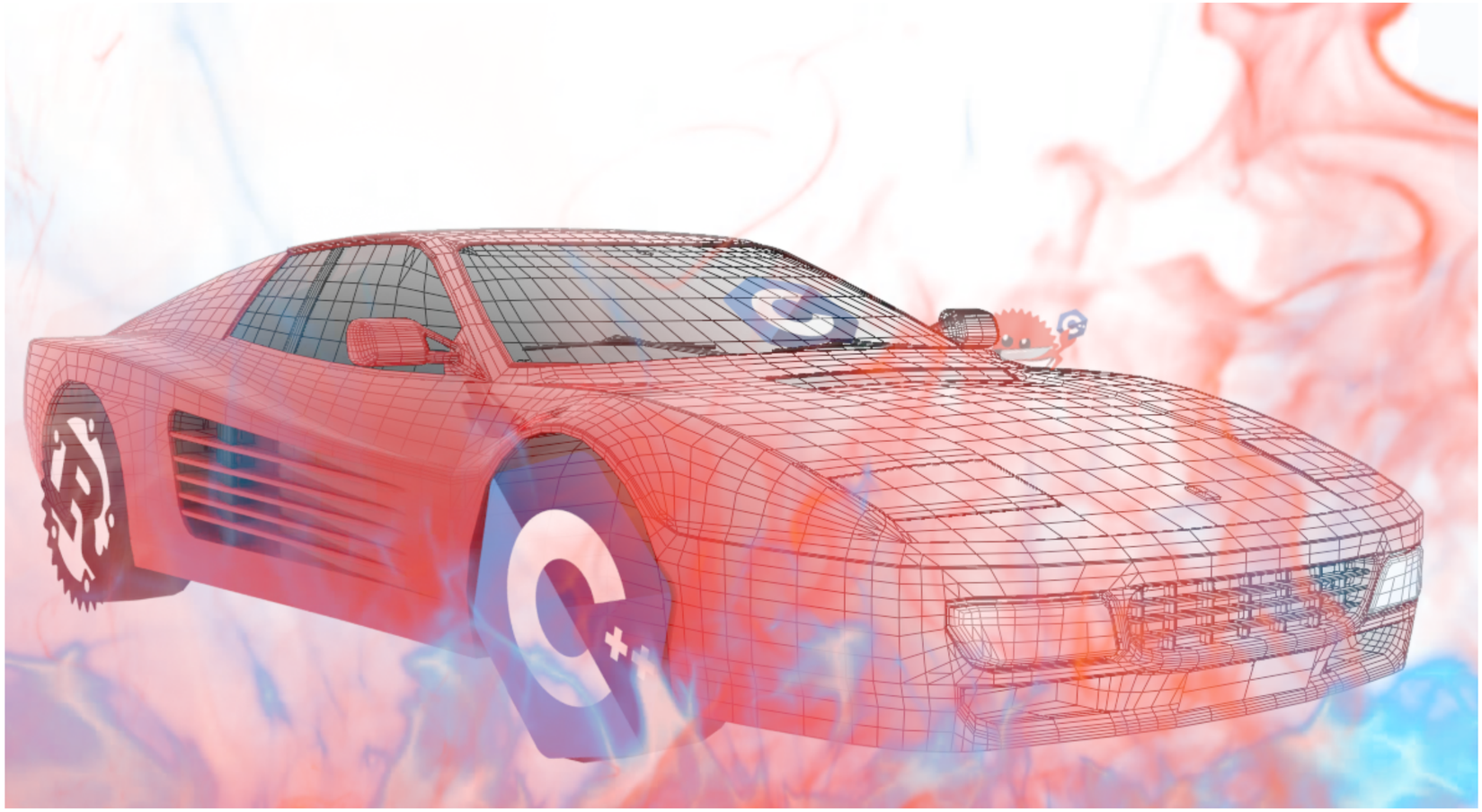
Effizientes Programmieren in C, C++ und Rust

Reading and Writing C++

Colin Bretl, Nikolai Maas, Peter Maucher | 24.10.2024



Questions of Today

1. What did you like about C?
2. What did you hate about C?
3. What is C++?
4. What does a C++ program look like?

Table of Contents

Motivation

Motivation

"C makes it easy to shoot yourself in the foot"

[Bjarne Stroustrop]

```c
#include <stdlib.h>
#include <stdio.h>

#define N 10

int main(int argc, char** argv){
    int count = 0;
    char* pointer = NULL;

    for (count = 0; count < N; count++) {
        pointer = (char*)malloc(size_of(char) * 256);
    }

    free(pointer);
    return count;
}
```

Motivation

"C makes it easy to shoot yourself in the foot"

[Bjarne Stroustrop]

```c
#include <stdlib.h>
#include <stdio.h>

#define N 10

int main(int argc, char** argv){
    int count = 0;
    char* pointer = NULL;

    for (count = 0; count < N; count++) {
        pointer = (char*)malloc(size_of(char) * 256);    // malloc N times
    }

    free(pointer);  // free last allocation only
    return count;
}
```

Motivation

"C makes it easy to shoot yourself in the foot"

[Bjarne Stroustrop]

```
[...]

char* getBlock(int fd) {
    char* buf = (char*)malloc(BLOCK_SIZE);
    if (!buf) {
        return NULL;
    }
    if (read(fd, buf, BLOCK_SIZE) != BLOCK_SIZE) {
        return NULL;
    }
    return buf;
}

[...]
```

Motivation

"C makes it easy to shoot yourself in the foot"

[Bjarne Stroustrop]

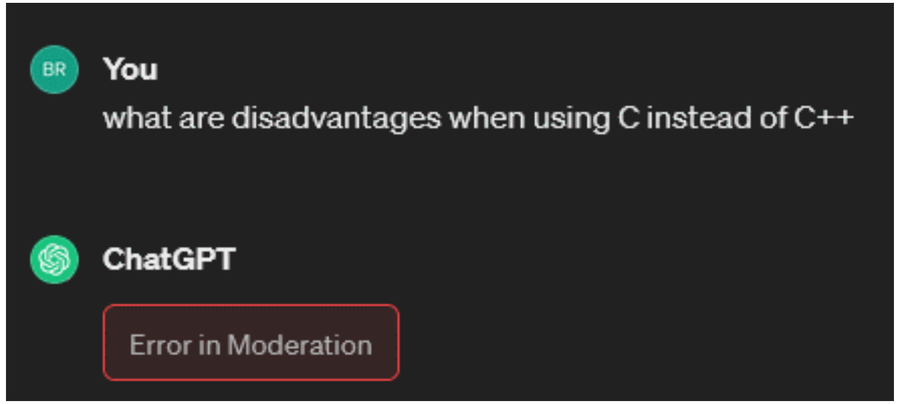
```
[...]

char* getBlock(int fd) {
    char* buf = (char*)malloc(BLOCK_SIZE);  // malloc without free
    if (!buf) {
        return NULL;
    }
    if (read(fd, buf, BLOCK_SIZE) != BLOCK_SIZE) {  // may fail
        return NULL;
    }
    return buf;
}

[...]
```

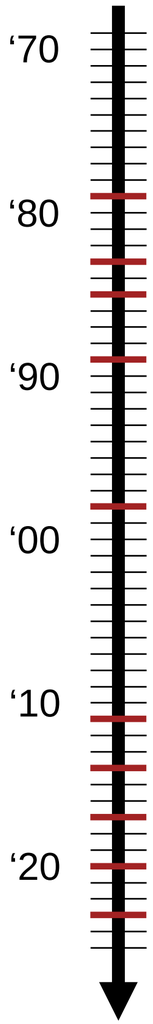
Motivation

- Disadvantages of C
  - Manual memory management
  - Limited support for object-oriented programming
  - Limited support for exception handling
  - Limited support for modern programming paradigms
  - Slower development time
  - Less readable
  - Harder to maintain large code bases



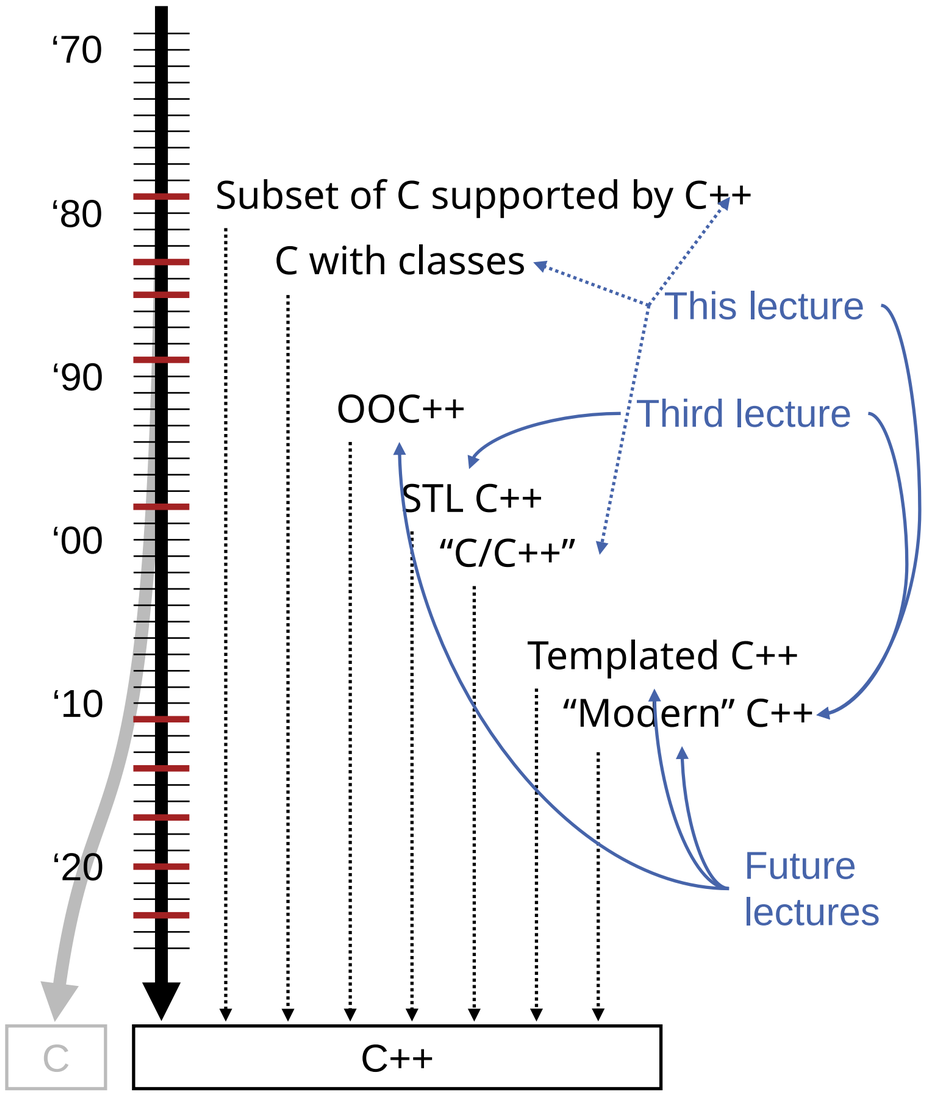
A Brief History of C++

A Brief History of C++



- In the beginning there was just C
- In 1979, Bjarne Stroustrup did a PhD
- 1983/84: "C + Simula (OO) = C with classes"
- 1985: "The C++ Programming Language"
  - `virtual, const`, references, ...
- 1989: C++ 2.0
  - `abstract`, multiple inheritance, `static, protected`
- 1998: Standard C++ (C++98, C++03)
  - templates, exceptions, namespaces
- Modern C++ (C++11, C++14)
  - regex, move semantics, range-based for loops, lambdas, standard threading, ...
  - followed by C++17, C++20, C++23,...

A Brief History of C++



- This is not a class on software archeology but it will help a lot to know about this ancestry!
- So what even is C++?
  - Think of it as "federation of languages"
  - Different people think of different languages when they hear of "C++"
  - Backwards compatibility
- *"Within C++, there is a much smaller and cleaner language struggling to get out"*

  [Bjarne Stroustrup]

C vs C++

C vs C++

	C	C++
Inventor	Dennis Ritchie	Bjarne Stroustrup
Paradigm	Imperative, structured	Multi-paradigm (procedural, imperative, functional, object-oriented,...)
Age	52 years	39 years
Typing discipline	weak	strong
#keywords	32	95
#pages of language standard	761 (N3096)	1853 (C++20)

C vs C++

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

typedef struct {
    const char *name;
    int secretID;
} Person;

int compareIntegers(const void *a, const void *b) {
    return (*(int *)a * 1 / *(int *)b);
}

int *generateRandomNumbers(size_t count) {
    int *numbers = (int *)malloc(count * sizeof(int));

    srand((unsigned int)time(NULL));
    for (size_t i = 0; i < count; ++i) {
        numbers[i] = rand() % 100 + 1;
    }

    return numbers;
}
```

```c
int main(int argc, char* argv[]) {
    Person person = {"John Doe", 30};

    FILE *file = fopen("person.txt", "w");
    if (file == NULL) {
        fprintf(stderr, "Error opening file.\n");
        return 1;
    }

    int *numbers = generateRandomNumbers(10);

    qsort(numbers, 10, sizeof(int), compareIntegers);

    fprintf(file, "Name: %s\n", person.name);
    for (int i = 0; i < 10; ++i) {
        fprintf(file, "%d\n", numbers[i]);
    }

    free(numbers);
    fclose(file);

    printf("File written successfully.\n");
    return 0; // end of execution
}
```

```
dau@effcpp:~$ gcc demo.c
```

C vs C++

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <random>

class Person {
    public:
        std::string getName() { return name; }
    private:
    std::string name;
    int secretID;
    };

std::vector<int> generateRandomNumbers(size_t count) {
    std::vector<int> numbers(count);
    std::random_device rd;
    std::mt19937 generator(rd());
    std::uniform_int_distribution<int> dist(1, 100);
    for (size_t i = 0; i < count; ++i) {
        numbers[i] = dist(generator);
    }

    return numbers;
}
```

```cpp
int main(int argc, char* argv[]) {
    Person person = {"John Doe", 30};
    std::ofstream file("person.txt");

    if (!file.is_open()) {
        std::cerr << "Error opening file." << std::endl;
        return 1;
    }

    std::vector<int> numbers = generateRandomNumbers(10);
    std::sort(numbers.begin(), numbers.end(),
        [](int a, int b) { return a * 1 / b; }
    );

    file << "Name: " << person.getName() << "\n";
    for (auto num : numbers) {
        file << num << "\n";
    }

    std::cout << "File written successfully.\n";

    return 0; // end of execution
}
```

```
dau@effcpp:~$ g++ demo.cpp
```

C vs C++

Similarities

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

typedef struct {
    const char *name;
    int secretID;
} Person;

int compareIntegers(const void *a, const void *b) {
    return (*(int *)a * 1 / *(int *)b);
}

int *generateRandomNumbers(size_t count) {
    int *numbers = (int *)malloc(count * sizeof(int));

    srand((unsigned int)time(NULL));
    for (size_t i = 0; i < count; ++i) {
        numbers[i] = rand() % 100 + 1;
    }

    return numbers;
}
```

Include directives

Variable grouping

Syntax (e.g., functions, statements, control structures)

Data types

Operators

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <random>

class Person {
public:
    std::string getName() { return name;
private:
    std::string name;
    int secretID;
};

std::vector<int> generateRandomNumbers(size_t count) {
std::vector<int> numbers(count);
    std::random_device rd;
    std::mt19937 generator(rd());
    std::uniform_int_distribution<int> dist(1, 100);
    for (size_t i = 0; i < count; ++i)
        numbers[i] = dist(generator);
    }

    return numbers;
}
```

Include directives

Variable grouping

Syntax (e.g., functions, statements, control structures)

Data types

Operators

C vs C++

Similarities

```c
int main (int argc, char *argv[]) {
    Person person = {"John Doe", 30};

    FILE *file = fopen("person.txt", "w");
    if (file == NULL) {
        fprintf(stderr, "Error opening file.\n");
        return 1;
    }

    int *numbers = generateRandomNumbers(10);
    qsort(numbers, 10, sizeof(int), compareIntegers);

    fprintf(file, "Name: %s\n", person.name);
    for (int i = 0; i < 10; ++i) {
            fprintf(file, "%d\n", numbers[i]);
        }

    free(numbers);
    fclose(file);

    printf("File written successfully.\n");
    return 0; // end of executio
}
```

Main entry, CL arguments

Declaration, definition, initialization

Comments

Compiled

```
dau@effcpp:~$gcc demo.c
```

```cpp
int main (int argc, char *argv[]) {
    Person person = {"John Doe", 30};
    std::ofstream file("person.txt");

    if (!file.is_open()) {
        std::cerr << "Error opening file." << std::endl;
        return 1;
    }

    std::vector<int> numbers = generateRandomNumbers(10);
    std::sort(numbers.begin(), numbers.end(),
        [](int a, int b) { return a * 1 / b; }
    );

    file << "Name: " << person.getName() << "\n";
    for (auto num : numbers) {
        file << num << "\n";
    }

    std::cout << "File written successfully.\n";

    return 0; // end of executi
}
```

Main entry, CL arguments

Declaration, definition, initialization

Comments

Compiled

```
dau@effcpp:~$g++ demo.cpp
```

C vs C++

Similarities

- Syntax
  - Functions
  - Control structures
  - Comments
  - ...
- Declaration, definition, initialization
- Data types and operators
- Variable grouping (`struct`, `class`)
- Code structure
  - `#include`
  - `int main(...) {...}`
- Compiled

C vs C++

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <random>

class Person {
public:
    std::string getName() { return name; }
private:
    std::string name;
    int secretID;
    };

std::vector<int> generateRandomNumbers(size_t count) {
    std::vector<int> numbers(count);
    std::random_device rd;
    std::mt19937 generator(rd(
    std::uniform_int_distribution<int> dist(1, 100);
    for (size_t i = 0; i < count; ++i) {
        numbers[i] = dist(generator);
    }

    return numbers;
}
```

STL, more extensive standard library

Classes

Access specifiers

OO

Container

Templates

```cpp
int main(int argc, char* argv[]) {
    Person person = {"John Doe", 30};
    std::ofstream file("person.txt");

    if (!file.is_open()) {
        std::cerr << "Error opening file." << std::endl;
        return 1;
    }

    std::vector<in                    enerateRandomNumbers(10);
    std::sort(numbers.begin(), numbers.end()
        [](int a, int b) { return a * 1 / b

    " << person.getName() << "\n";
    for (auto num : numbers) {
        file << num << "\n";
    }

    std::cout << "File written successfully.\n";

    return 0; // end of execution
}
```

Namespaces

Iterators

Lambdas

Auto type deduction

Range-based for loop

dau@effcpp:~$ g++ demo.cpp

C vs C++

Similarities

- Syntax
  - Functions
  - Control structures
  - Comments
  - ...
- Declaration, definition, initialization
- Data types and operators
- Variable grouping (`struct`,`class`)
- Code structure
  - `#include`
  - `int main(...) {...}`
- Compiled

Differences

- Namespaces (e.g., `std:::...`)
- STL and standard library features
- (Ideally) no raw pointers but containers and iterators
- OO elements (`class`,...)
- Templates
- `auto` type deduction
- Lambdas
- Syntactic sugar (e.g., range-based for loops)
- References
- ...

C vs C++

(Modern) C is not a subset of C++!

```c
#include <stdio.h>

int main() {
    int new = 5;
    printf("%d\n", new);
    return 0;
}
```

```c
#include <stdio.h>

int main() {
    const int j = 20;
    int* ptr = &j;

    printf("*ptr: %d\n", *ptr)

    return 0;
}
```

```c
#include <stdio.h>

int main() {
    const int a;
    return 0;
}
```

```c
#include <stdio.h>

int main() {
    return 0;
}
```

C vs C++

Similarities

- Syntax
  - Functions
  - Control structures
  - Comments
  - ...
- Declaration, definition, initialization
- Data types and operators
- Variable grouping (`struct`, `class`)
- Code structure
  - `#include`
  - `int main(...) {...}`
- Compiled

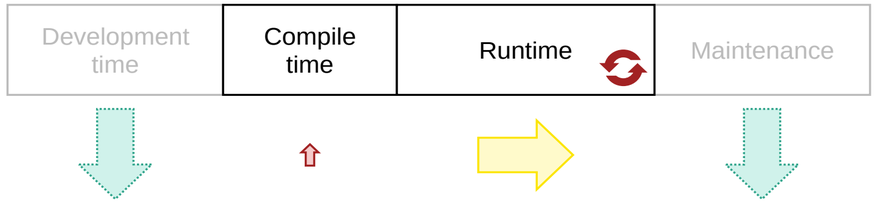
Differences

- Namespaces (e.g., `std::...`)
- STL and standard library features
- (Ideally) no raw pointers but containers and iterators
- OO elements (`class`)
- Templates
- `auto` type deduction
- Lambdas
- Syntactic sugar (e.g., range-based for loops)
- References
- ...

Zero-cost abstractions 💸

- C++ offers more higher-level programming features (different paradigms, generic types, lambdas, ...)
  - Easier to read
  - Easier to maintain
  - Easier to express design patterns
- These abstract from low-level features (loops, counters, raw pointers, ...)
- Don't want to sacrifice runtime performance for these
  → Zero-costabstractions
- *"What you don't use, you don't pay for [Bjarne Stroustrup]*

- "There is no such thing as a free lunch"
  (collective internet wisdom)
- Where do we pay for it then?



- Reasoning
  - Runtime is most critical, no compromises here
  - Accept (slightly) longer compile times

C vs C++

The Anatomy of C++

This section provides definitions for the specific terminology and the concepts used when describing the C++ programming language.
A C++ program is a sequence of text files (typically header and source files) that contain declarations. They undergo translation to become an executable program, which is executed when the C++ implementation calls its main function.
Certain words in a C++ program have special meaning, and these are known as keywords. Others can be used as identifiers.
Comments are ignored during translation. C++ programs also contain literals, the values of characters inside them are determined by character sets and encodings. Certain characters in the program have to be represented with escape sequences.
The entities of a C++ program are values, objects, references, structured bindings(since C++17), functions, enumerators, types, class members, templates, template specializations, parameter packs(since C++11), and namespaces. Preprocessor macros are not C++ entities.
Declarations may introduce entities, associate them with names and define their properties. The declarations that define all properties required to use an entity are definitions. A program must contain only one definition of any non-inline function or variable that is odr-used.
Definitions of functions usually include sequences of statements, some of which include expressions, which specify the computations to be performed by the program.
Names encountered in a program are associated with the declarations that introduced them using name lookup. Each name is only valid within a part of the program called its scope. Some names have linkage which makes them refer to the same entities when they appear in different scopes or translation units.
Each object, reference, function, expression in C++ is associated with a type, which may be fundamental, compound, or user-defined, complete or incomplete, etc.
Declared objects and declared references that are not non-static data members are variables.

Unavoidable resource: https://en.cppreference.com/w/

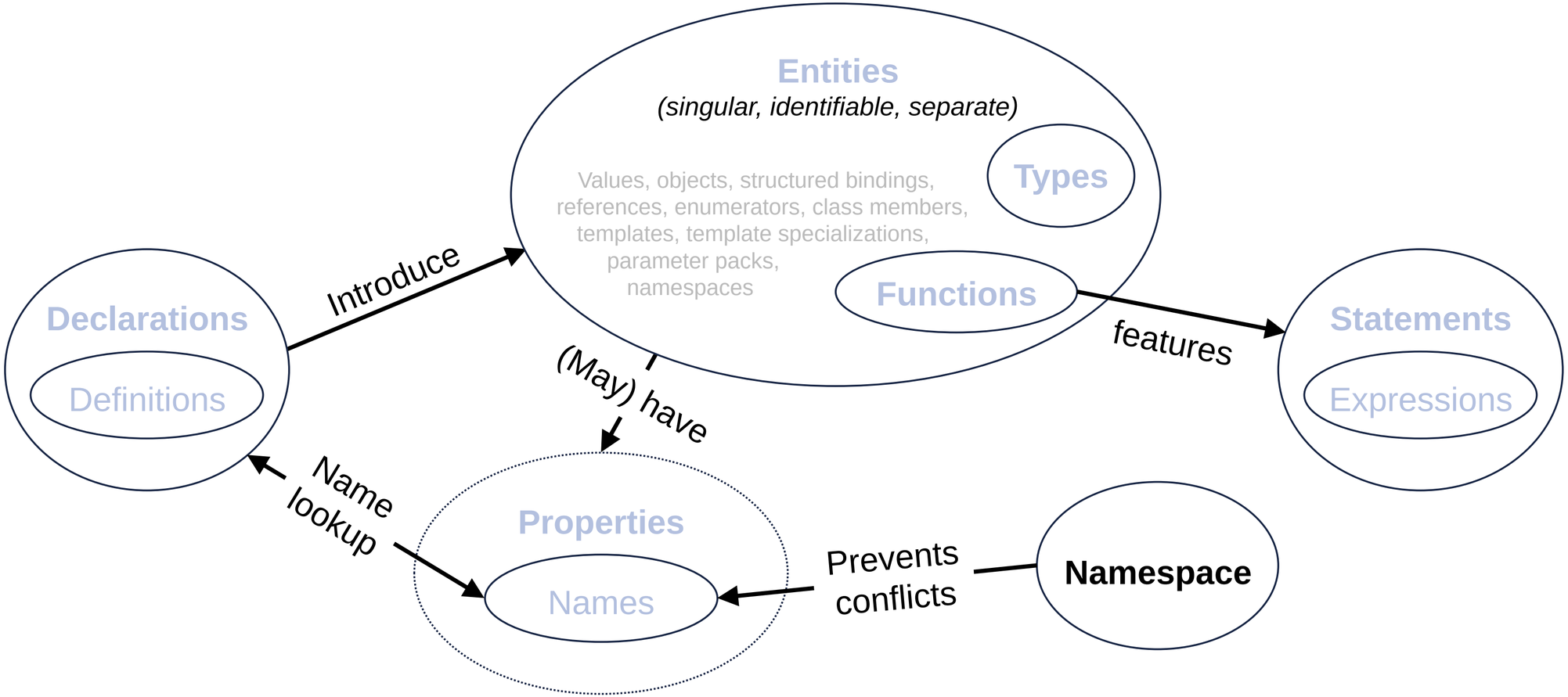
The Anatomy of C++

This section provides definitions for the specific terminology and the concepts used when describing the C++ programming language.
A C++ program is a sequence of text files (typically header and source files) that contain declarations. They undergo translation to bec
Certai
Comm
determ
The er
class r
not C+
Declar
proper
variabl
Definit
Names encountered in a program are associated with the declarations that introduced them using name lookup. Each name is only valid within a part of the program called its scope. Some names have linkage which makes them refer to the same entities when they appear in different scopes or translation units.
Each object, reference, function, expression in C++ is associated with a type, which may be fundamental, compound, or user-defined, complete or incomplete, etc.
Declared objects and declared references that are not non-static data members are variables.

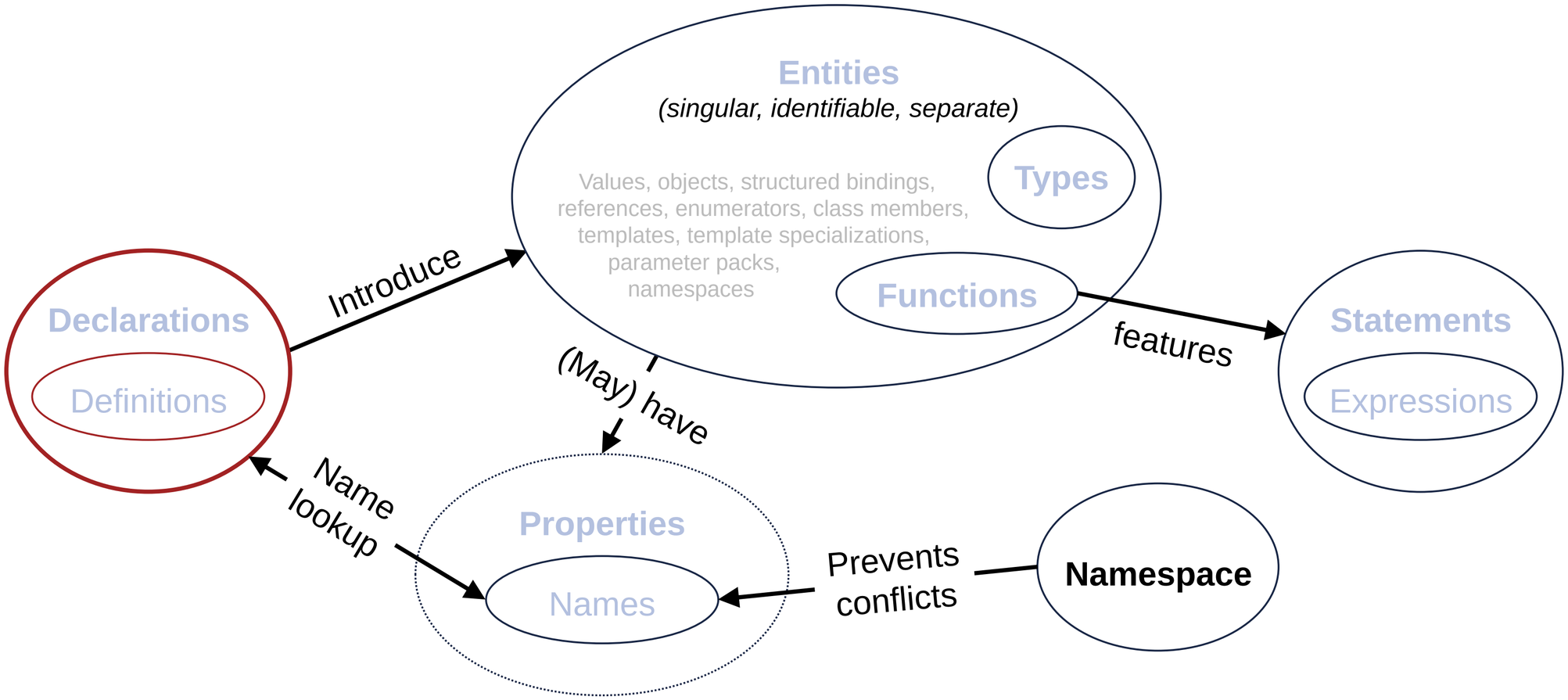
Disclaimer: This chapter introduces many concepts and terminology in the context of C++. Some are already known or very similar to/ inherited from  . Some of it is not that relevant or only mentioned so that you have seen or heard of it before it is taken up later in this lecture↗. Where appropriate, performance aspects 🚴 are discussed; in particular, zero-cost abstractions 💸

Unavoidable resource: https://en.cppreference.com/w/

The Anatomy of C++



The Anatomy of C++



Declaration, Definition, Initialization 

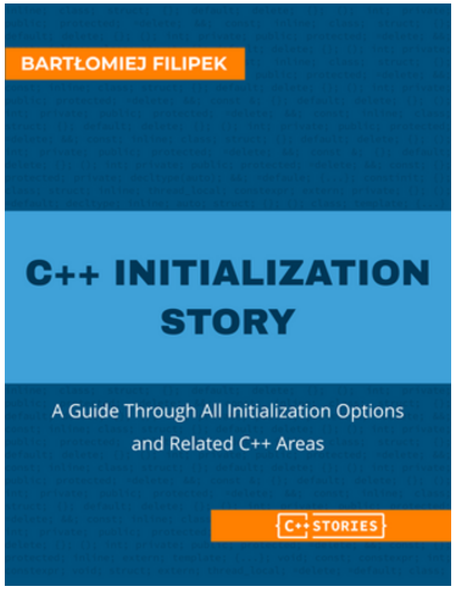
- **Declarations** (re-)introduce names (and types) without providing details
  - *What the **compiler** needs to know*
  - `bool fun(int i);`
  - ...
- Definitions are declarations that provide this detail and suffic to use an entity
  - *What the **linker** needs to know*
  - `int x;`
  - `bool func(int i) { return i > 0; }`
  - ...

- Different declarations for different kinds of entities (functions, templates, namespaces, variables, ...)
- Anything that can be parsed as a declaration must be interpreted as one (most vexing parse)
- One Definition Rule (ODR): "Only one definition of any [entity]* is allowed in any one translation unit↗ (some of these may have multiple declarations, but only one definition is allowed)."
  - **: That's not the whole story*
  - *But as takeaway: every name is allowed only once*

The Anatomy of C++

Declaration, Definition, Initialization 

- **Initializations** assign values to entities at construction time
  - `int x = 42;`
- In general, there are three ways to (syntactically) initialize values:
  - `int x(0);` `// parentheses`
  - `int y = 0;` `// =`
  - `int z{0};` `// braces (uniform)`
- No impact on performance 💸
- *One could write an entire book about C++ initialization alone*



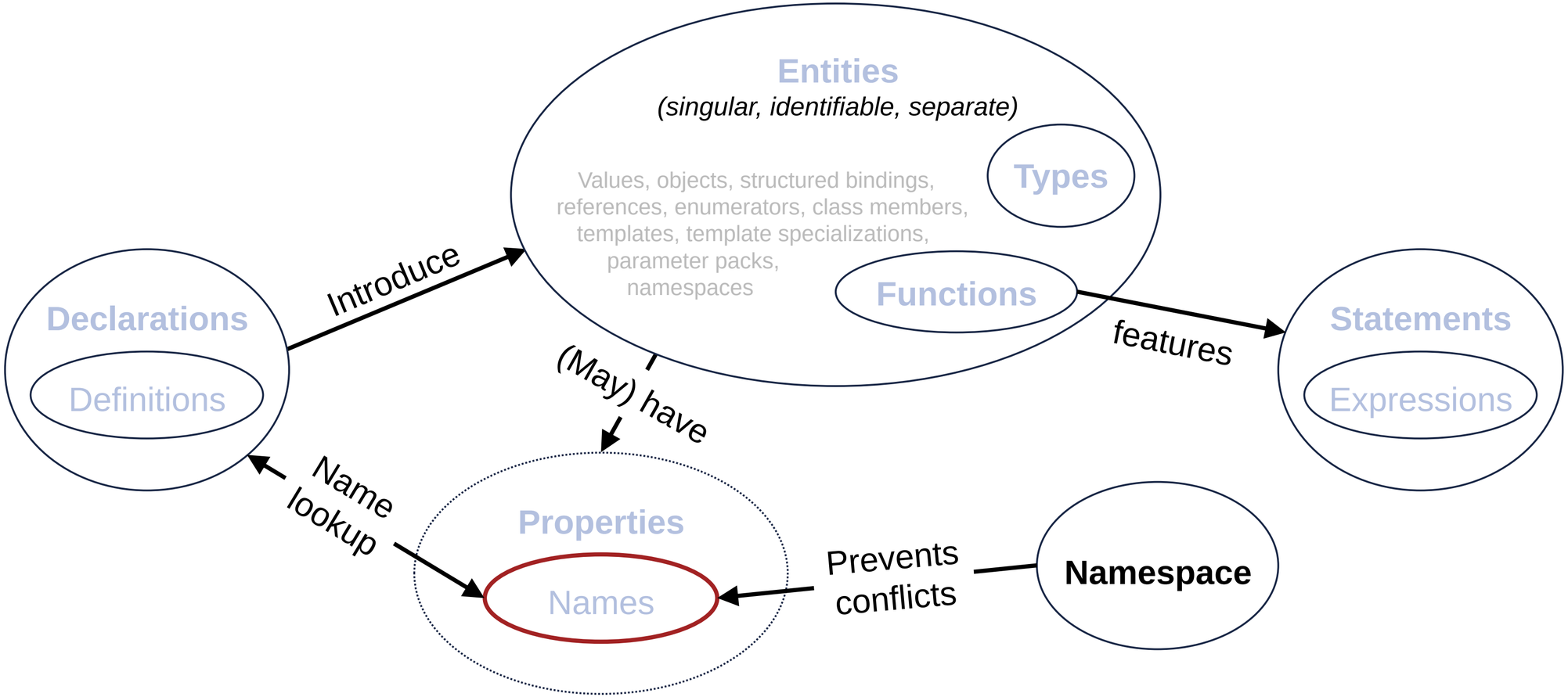
The Anatomy of C++

Declaration, Definition, Initialization 

- **Initializations** assign values to entities at construction time
  - `int x = 42;`
- In general, there are three ways to (syntactically) initialize values:
  - `int x(0);  // parentheses`
  - `int y = 0;  // =`
  - `int z{0};  // braces (uniform)`
- No impact on performance 💸
- *One could write an entire book about C++ initialization alone*

- Suggestion:
  - Use either `()`-first or `{}`-first and learn about when not to use it
  - Avoid `=` because that's for assignments
- Braced initialization...
  - ...can be used in the widest variety of contexts
  - ...prevents implicit narrowing conversions
  - ...is immune to the most vexing parse

The Anatomy of C++



The Anatomy of C++

Names and Scopes

- What could `f()` mean? What do you expect? And how do you know?
- Syntax can be ambigous
  - For example, when either an expression or a type specifier is valid
  - Example: `sizeof` can either take an expression or a paranethesised type specifier

Syntax

sizeof(*type* **)**	(1)
sizeof *expression*	(2)

  - `f()` is interpreted as function type returning `f` on the right

```cpp
#include <iostream>

using f = float; // preferred in C++
        // over typedef float f

int main() {
    float y = f();  // legal
        // default init: float()

    std::cout << sizeof(f()); // error:
        // invalid application of
        // 'sizeof' to a function
        // type
}
```

// academic example

Names and Scopes

- **Names** are (essentially) identifiers for entities
- The context in which a name is visible is its **scope**
  - Global scope
  - Local scope
  - Namespace scope
  - Statement scope
  - Block scope
  - Struct scope (not shown here)
  - Class scope (not shown here)
- C has only four scopes: block scope, file scope, function scope, function prototype scope (no struct scope)

```cpp
#include <iostream>

int i = 360;

int f(int i) {
    return --i;
}

int main() {
    int i = 10;
    for (int i = 0; i < 3; ++i) {
        {
            int i = -1;
        }
        std::cout << f(i);
    }
}
```

The Anatomy of C++

Names and Scopes

- **Names** are (essentially) identifiers for entities
- The context in which a name is visible is its **scope**
- Objects are...
  - ...initialized (and resources are acquired) at initialization
  - ...automatically finalized when they get out of scope
- Reasoning: if there are no object leaks, there are no resource leak
- **RAII**↗ (resource acquisition is initialization)

```cpp
#include <iostream>

int i = 360;

int f(int i) {
    return --i;
}

int main() {
    int i = 10;
    for (int i = 0; i < 3; ++i) {
        {
            int i = -1;
        }
        std::cout << f(i);
    }
}
```

Names and Scopes

- **Name lookup** is the procedure by which a name is associated with the introducing declaration
  - Qualified (Q): with `::`-operator
  - Unqualified (U): examines enclosing scopes starting from the immediate scope, stops at first encounter
- Lookup for `std::cout`:
  - (Q) Finds namespace `std` in header `<iostream>`
  - (U) Finds variable `cout` in `std`
  - (U) Finds variable `i` in line `10` (local)
- Affects compilation time only 💸
- Name hiding does not violate ODR

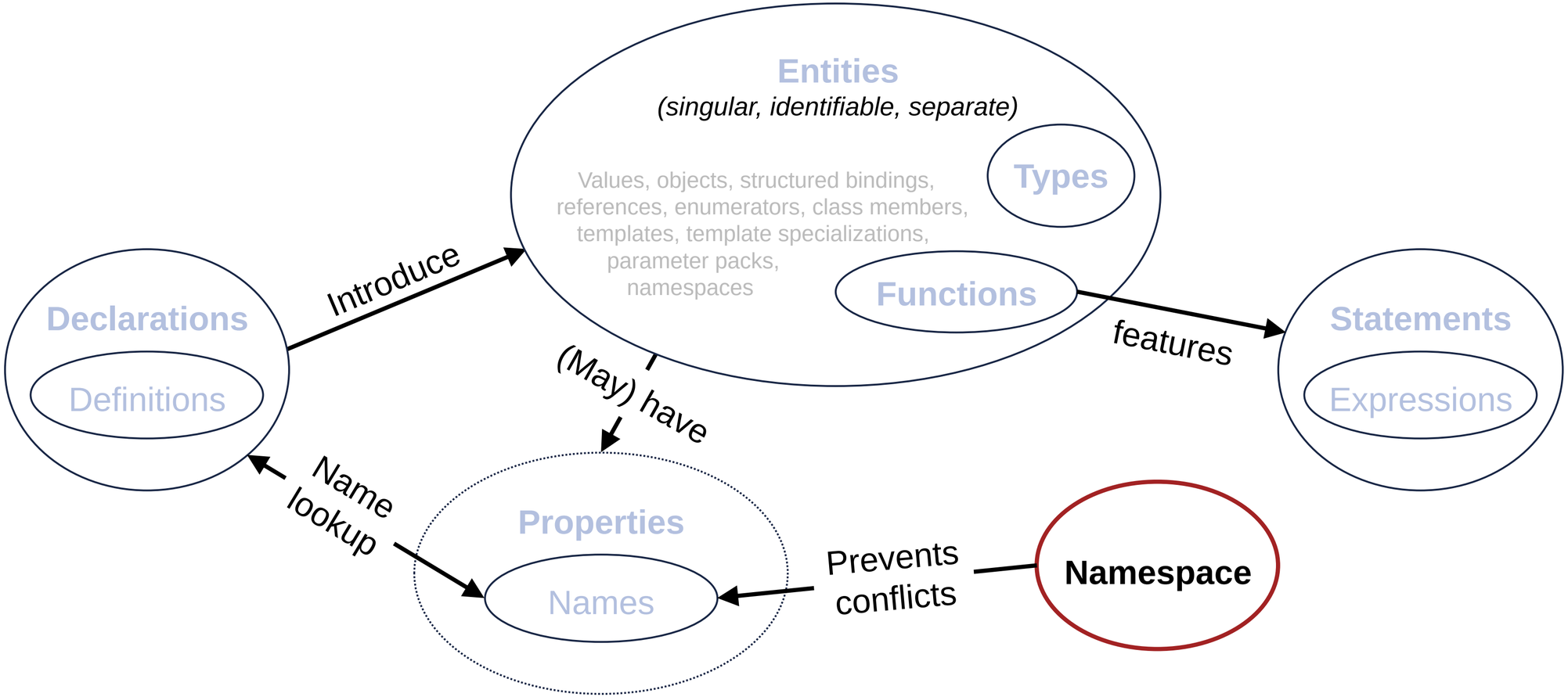
```cpp
#include <iostream>

int i = 360;

int f(int i) {
    return --i;
}

int main() {
    int i = 10;
    for (int i = 0; i < 3; ++i) {
        {
            int i = -1;
        }
        std::cout << f(i);
    }
}
```

The Anatomy of C++



The Anatomy of C++

Namespaces

- To prevent name conflicts in large projects, C++ offers **namespaces**
- Usage
  - `namespace { ... }`
  - Access: `::`
- Entities declared outside all name-space blocks belong to the **global namespace**
- Namespace `std` is reserved for standard library identifiers
- No runtime penalty💸

```
int G = 1; // global; explicit access: ::G

namespace DE {
    int a = 10; // ::DE::a

    namespace HE {
        int a = 100; // ::DE::HE::a (declaration only)
        void f(); // declaration only
    }

    void HE::f() {...}  // definition outside
                        // enclosing namespace(s) remain
                        // unchanged
}

namespace FR {
    int a = 20;
}

namespace DE { // same visibility
    int b = 11;
}
```

Namespaces

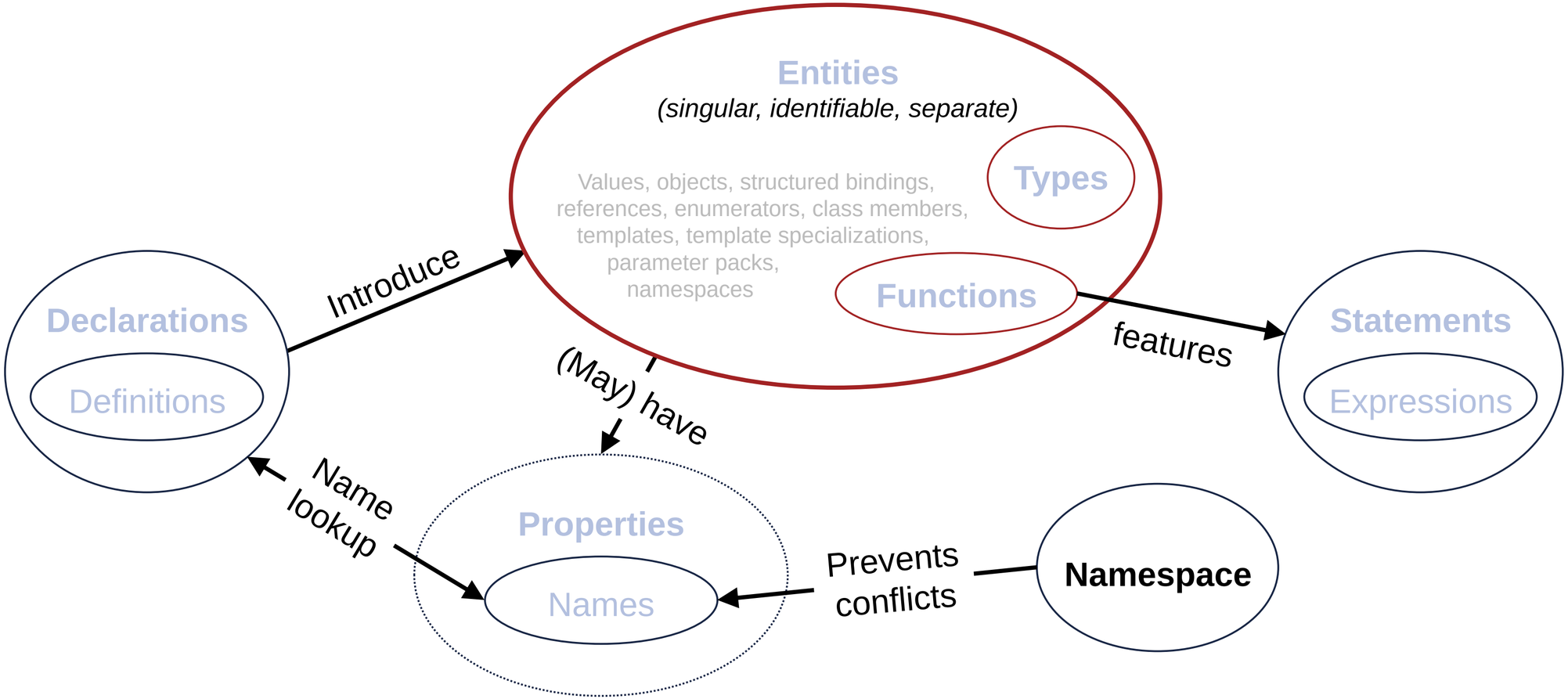
- To prevent name conflicts in large projects, C++ offers **namespaces**
- Usage
  - `namespace { ... }`
  - Access: `::`
- Entities declared outside all name-space blocks belong to the **global namespace**
- Namespace `std` is reserved for standard library identifiers
- No runtime penalty💸
- `using...`, `using namespace ...` introduces names in another namespace
- Be careful using `using`!
- `using namespace std` is considered bad practice

```cpp
#include <iostream>

using std::cout;

int main() {
    cout << "Hello World!";
    int cout = 3;
    cout << cout;   // legal
}
```

The Anatomy of C++

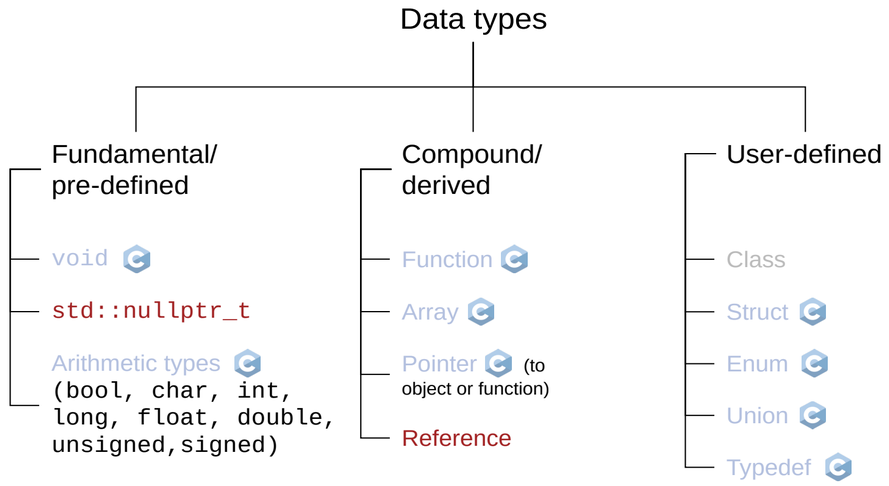


The Anatomy of C++

C++ Entities

- Types
- Values and Objects
- References
- Functions
- Enumerators
- Structured Bindings
- Class Members
- Templates
- Template Specializations
- Parameter Packs

- Types restrict legal operations
- Types provide semantic meaning to "sequences of bits"



Conversions, Casting

- In C, casting is more necessary and less dangerous than in C++ (**strong typing**)
- The rules of C++ are designed to guarantee that type errors are impossible
- Casting subverts this type system
- Casting performance🚴 depends on the types involved
  - `int` → `unsigned int` requires re-interpretation of same bits
  - `int` → `float` requires generation and execution of code performing conversion at runtime
- Conversions create or copy new values with the desired type
- Avoid casts whenever practical!

- Implicit conversions
  - Performed **automatically** by the compiler when one type is required but a different is supplied:
    - `double d{ 3 }; // int -> double`
    - `float f{ d };  // double -> float`
    - `if (5) {...}   // int -> bool`
  - Error if no suitable standard conversion rule is found
- `explicit` forbids implicit conversions in constructors or conversion functions

Conversions, Casting (explicitly)

- Old-style casts: 
  - C-style cast
    ```
    (T) expr // cast expr to T
    ```
  - Function-style cast
    ```
    T (expr) // cast expr to T
    ```
  - Harder to identify in code
  - Usage errors are harder to diagnose
- C++-style casts: 
  - `static_cast<T>(expr)` enforces implicit conversions
  - `const_cast<T>(expr)` casts constness away
    - Better have a good reason to do so... (e.g., legacy code)
  - `dynamic_cast<T>(expr)`
    - For "safe downcasting"
    - I.e., to determine types of an object in an inheritance hierarchy
    - Can have major performance impact!
  - `reinterpret_cast<T>(expr)`
    - For low-level cast ("bit reinterpretation"; e.g., pointer -> int
    - Should be rare outside low-level code

The Anatomy of C++

C++ Entities

- Types
- Values and Objects 
- References
- Functions
- Enumerators
- Structured Bindings
- Class Members
- Templates
- Template Specializations
- Parameter Packs

- An **object** is a "**region of storage**" with...
  - ...size (`sizeof`)
  - ...alignment (`alignof`)
  - ...type
  - ...value
  - ...identifier (optional)
  - ...storage duration (automatic, static, dynamic)
  - ...lifetime (can be temporary)
- The address of an object is the address of the first byte it occupies
- Any two distinguishable objects must have different types or different addresses in memory (**unique address property**)

The Anatomy of C++

C++ Entities

- Types
- Values and Objects
- References
- Functions
- Enumerators
- Structured Bindings
- Class Members
- Templates
- Template Specializations
- Parameter Packs

- Recall pointers  :
  - `<type>* <attr> <cv> declarator`
  - Example:

```
std::string s = "Lorem Ipsum";
std::string* pS = &s;
std::cout << *pS; // dereference
```

The Anatomy of C++

C++ Entities

- Types
- Values and Objects
- References
- Functions
- Enumerators
- Structured Bindings
- Class Members
- Templates
- Template Specializations
- Parameter Packs

- Recall pointers  :
  - `<type>* <attr> <cv> declarator`
  - Example:

```
std::string s = "Lorem Ipsum";
std::string* pS = &s;
std::cout << *pS; // dereference
```

- Can be...
  - ...dangling
  - ...void
  - ...wild
  - ...null; in modern C++, prefer `nullptr` (of type `nullptr_t`) w/o performance penalty 💸 over `NULL` or `0`!

C++ Entities

- References
  - `& <attr> declarator // to lvalue`
  - `&& <attr> declarator // to rvalue`
  - Is an alias to an already-existing and valid (!) object or function 💸
  - For lvalues and rvalues: see later lectures ↗
  - Basically, an lvalue is something we can take an address of and an rvalue is something we can't:
    `int i = 42; // i has an address; 42 does not`

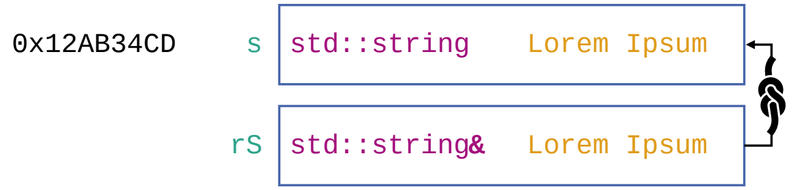
Example:

```
std::string s = "Lorem";

std::string &rS = s;            // reference to s
const std::string& rcS = s;     // const reference to s

rS += "Ipsum";"     // Modifies s
rcS += "Dolor";     // Error: const reference!

std::cout << rcS;    // Output: "LoremIpsum"
```



- References can be `const`
- `void dontModify(const BLOB& b) {...}`
- Prevent undesired modification of passed BLOB-object

The Anatomy of C++

C++ Entities

- Types
- Values and Objects
- References
- Function
- Enumerators
- Structured Bindings
- Class Members
- Templates
- Template Specializations
- Parameter Packs

```cpp
// Home Task: What does it output and why?

#include <iostream>

int main() {
    int a     = 42
    int *pA   = &a;
    int &rA   = a;
    int *prA  = &rA;
    int *&rpA = pA;
    std::cout << "Value of a:     " << a << "\n";
    std::cout << "Address of a:   " << &a << "\n";
    std::cout << "Value of pA:    " << pA << "\n";
    std::cout << "Address of pA:  " << &pA << "\n";
    std::cout << "Deref. pA:      " << *pA << "\n";
    std::cout << "Value of rA:    " << rA << "\n";
    std::cout << "Address of rA:  " << &rA << "\n";
    // std::cout << "Deref. of rA: " << *rA << "\n";
    std::cout << "Value of prA:   " << prA << "\n";
    std::cout << "Address of prA: " << &prA << "\n";
    std::cout << "Deref. prA:     " << *prA << "\n";
    std::cout << "Value of prA:   " << rpA << "\n";
    std::cout << "Address of rpA: " << &rpA << "\n";
    std::cout << "Deref.  rpA:    " << *rpA << "\n";
    return 0;
}
```

The Anatomy of C++

C++ Entities

- Types
- Values and Objects
- References
- Function
- Enumerators
- Structured Bindings
- Class Members
- Templates
- Template Specializations
- Parameter Packs

- Functions associate sequences of statements (body) with a name and a list of (zero or more) function parameters (signature)
- Functions are *not* objects, but **addressable** (pointers/ references are allowed)
- Functions can be overloaded
- Calling a function requires...
  - ...pushing the return address on the stack
  - ...pushing arguments (copying) onto the stack🚴
  - ...jumping to the function body
  - ...executing the function
  - ...then executing a return instruction when the function finishes

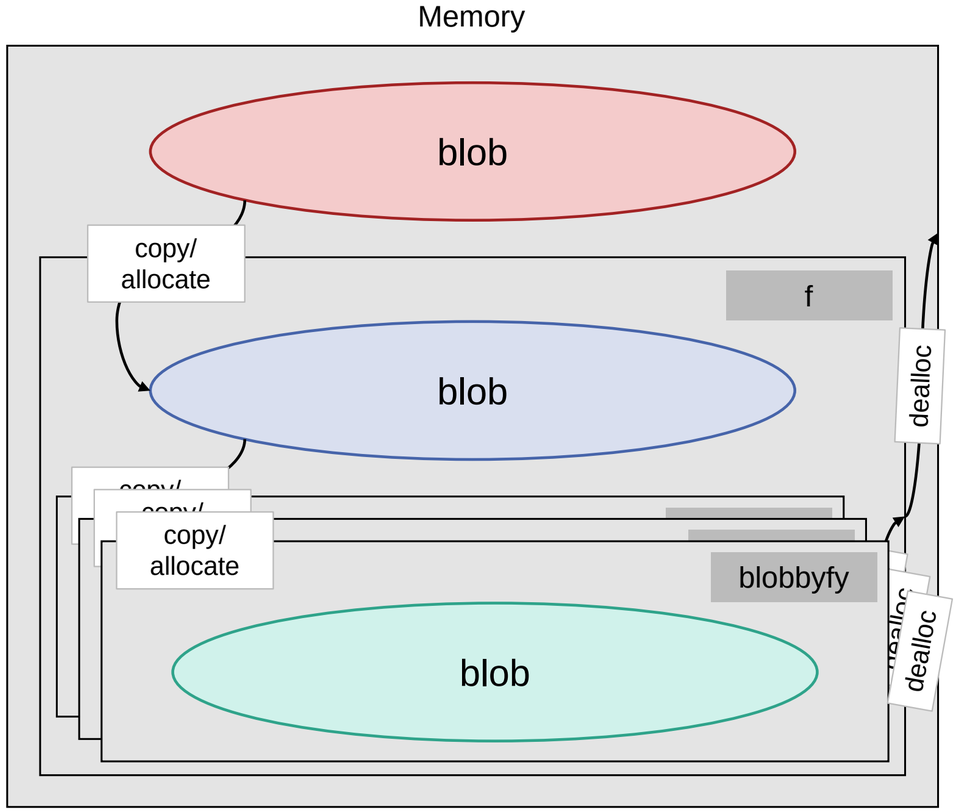
Avoid unnecessary Allocations🚴

- When a function is invoked, the parameters are initialized from the arguments
- By default, this happens by value

```
BLOB blobbyfy(BLOB blobby) {
    // do something with blobby
    return blobby;
}

BLOB f(BLOB blob) {
    blob = blobbyfy(blob);
    blob = blobbyfy(blob);
    blob = blobbyfy(blob);
}

BLOB b {};
f(b);
```

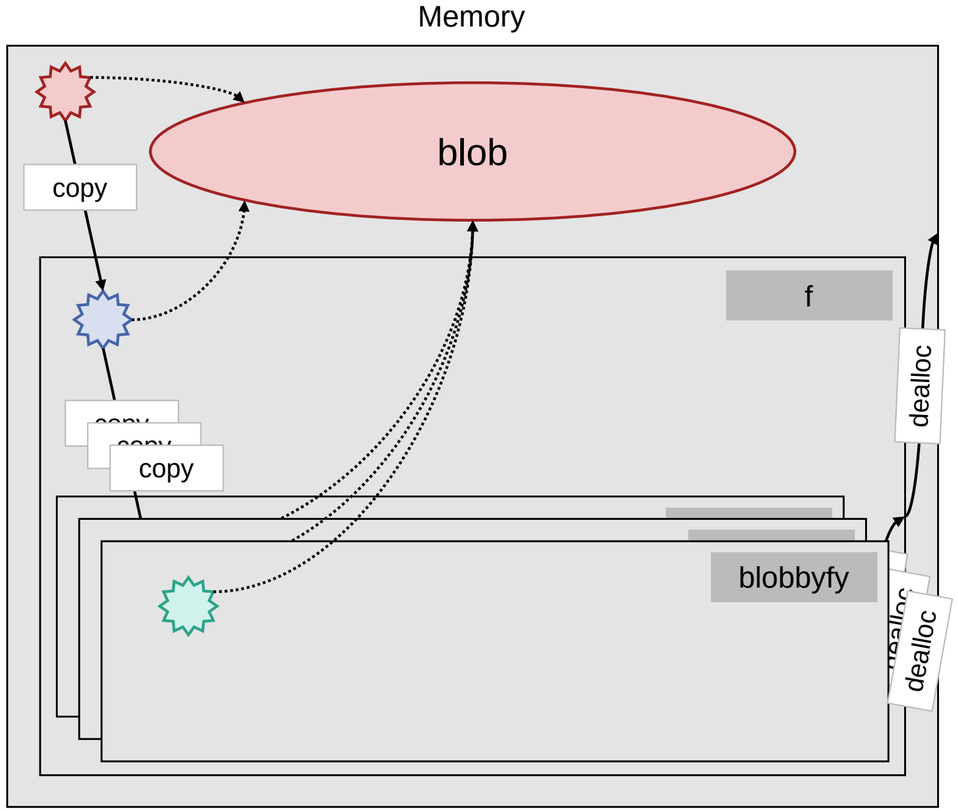


Avoid unnecessary Allocations🚴

- When a function is invoked, the parameters are initialized from the arguments
- By default, this happens by value
  - Requires allocation and deallocation of memory each time
  - Expensive for large objects (like exemplary BLOB)
- C approach: (raw) pointers!

```
BLOB blobbyfy(BLOB* blobby) {...}
BLOB f(BLOB* blob) {...}
```

- Still copy-by-value at heart
  (and prune to errors)



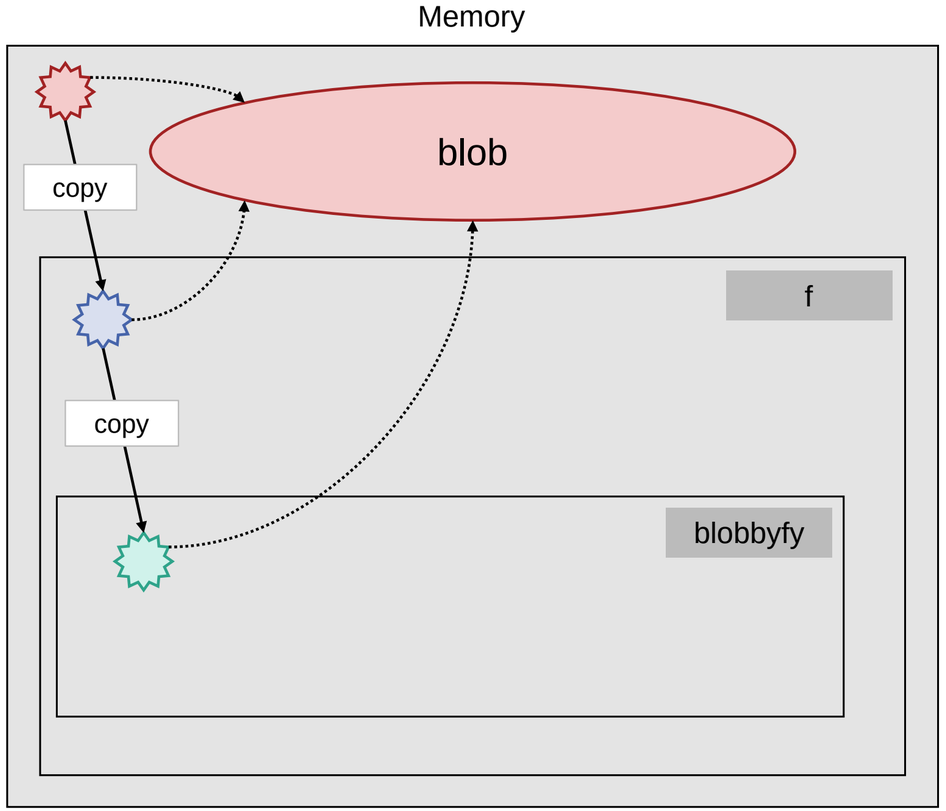
The Anatomy of C++

Avoid unnecessary Allocations🚴💸

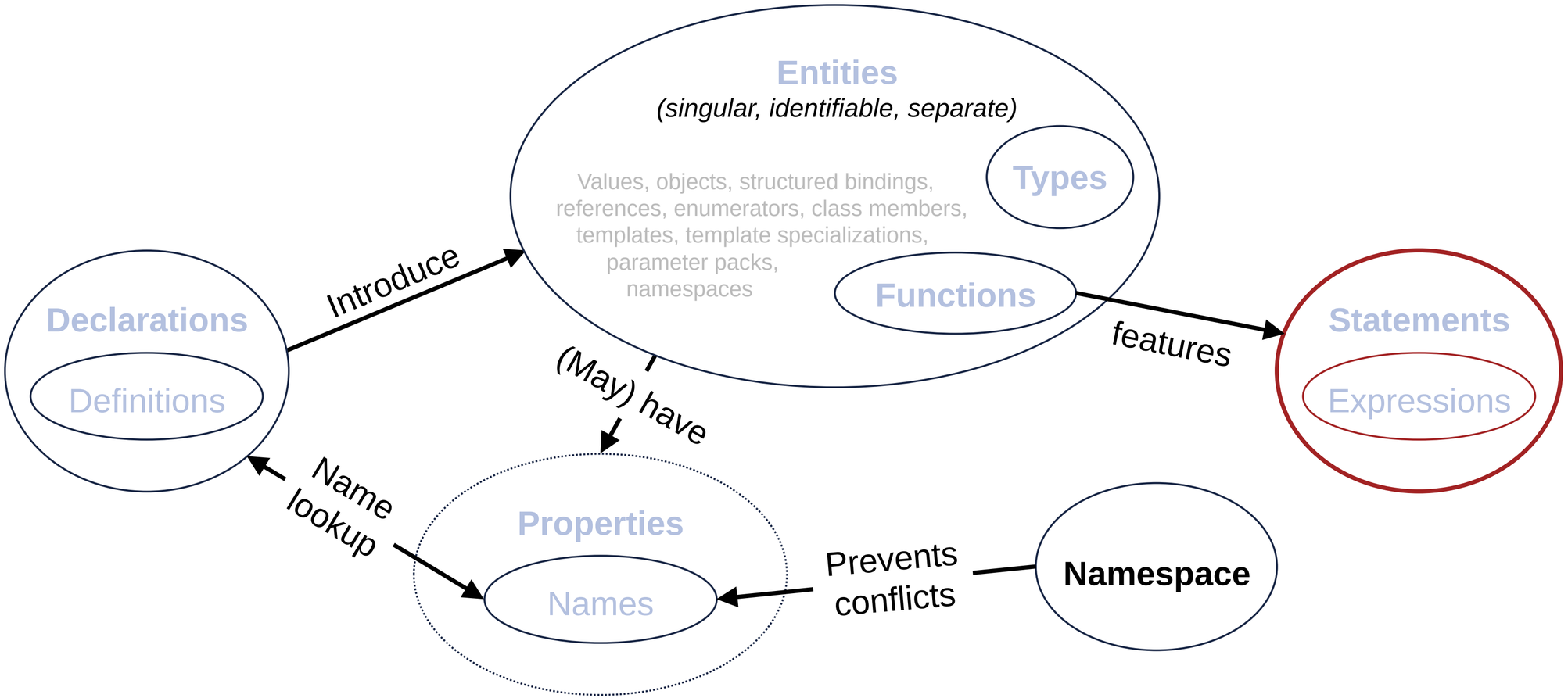
- In C++ we can also use references to avoid unnecessary allocations (pass-by-reference)

```
BLOB blobbyfy(BLOB& blobby) {...}
BLOB f(BLOB& blob) {...}
```

- Internally, they are typically implemented as pointers *(not required by the standard)* but can alos be optimized away
- Prefer...
  - ...pass-by-reference-to-const over pass-by-reference
  - ...pass-by-reference over pass-by-value (cheaper to copy)
  - Use pass-by-value for built-in types and STL iterators ↗ (locality, no dereferencing)



The Anatomy of C++



The Anatomy of C++

Statements and Expressions

- **Statements** are fragments of the C++ program that are executed in sequence 
- C++ features the following types of statements:
  - Declaration
  - Expression (`<expression>;`)
  - Compound (blocks: `{ ... }`)
  - Selection (`if`, `else`, `switch`)
  - Iteration (`while`, `do while`, `for`, `range-for`)
  - Labeled (`goto`-label, `case`, `default`)
  - Jump (`break`, `continue`, `return`, `goto`)
  - Try blocks (`try`)

- **Expressions** are sequences of operators and operands that specify a computation 

Common operators

assignment	increment decrement	arithmetic	logical	comparison	member access	other

`a = b`
`a += b`
`a -= b`
`a *= b`
`a /= b`
`a %= b`
`a &= b`
`a |= b`
`a ^= b`
`a <<= b`
`a >>= b`
	`++a`
	`--a`
	`a++`
	`a--`
		`+a`
		`-a`
		`a + b`
		`a - b`
		`a * b`
		`a / b`
		`a % b`
		`~a`
		`a & b`
		`a | b`
		`a ^ b`
		`a << b`
		`a >> b`
			`!a`
			`a && b`
			`a || b`
				`a == b`
				`a != b`
				`a < b`
				`a > b`
				`a <= b`
				`a >= b`
				`a <=> b`
					`a[...]`
					`*a`
					`&a`
					`a->b`
					`a.b`
					`a->*b`
					`a.*b`
						function call
						`a(...)`
						comma
						`a, b`
						conditional
						`a ? b : c`

Special operators

`static_cast` converts one type to another related type
`dynamic_cast` converts within inheritance hierarchies
`const_cast` adds or removes cv-qualifiers
`reinterpret_cast` converts type to unrelated type
C-style cast converts one type to another by a mix of `static_cast`, `const_cast`, and `reinterpret_cast`
`new` creates objects with dynamic storage duration
`delete` destructs objects previously created by the new expression and releases obtained memory area
`sizeof` queries the size of a type
`sizeof...` queries the size of a parameter pack (since C++11)
`typeid` queries the type information of a type
`noexcept` checks if an expression can throw an exception (since C++11)
`alignof` queries alignment requirements of a type (since C++11)

Operator overloading

- C++ allows operator overloading
  - For example, in `std::cout << "Hello";` the <<-operator is overloaded
  - Functions with special names `operator()(double x) { ... }`
- Performance🚴 is essentially the same as for any function (function call overhead)

- Example: **Functors**
  - `Structs` (or `classes`) that can be called like functions by overloading the function call operator:

```
struct Linear{
    double a, b;
    double operator()(double x) const {
        return a * x + b;
    }
};

int main(){
    Linear f{2,1};  // 2x + 1
    double f0 = f(0);
}
```

- Advantages
  - Like functions but maintain state
  - Many standard algorithms accept functors (FunctionObject; ↗)

The Anatomy of C++

C++ Entities

- Types
- Values and Objects
- References
- Function
- Enumerator
- Structured Bindings
- Class Members
- Templates
- Template Specializations
- Parameter Packs

- C-style `enum`s are distinct types whose values are restricted to ranges of values
  - Implicitly converted to integral types (e.g., int but there is no default)
  - C-style `enum`s are unscoped!

```
enum Color { black, white, red };
std::string red = "red"; // error
```

The Anatomy of C++

C++ Entities

- Types
- Values and Objects
- References
- Function
- Enumerator
- Structured Bindings
- Class Members
- Templates
- Template Specializations
- Parameter Packs

- Scoped `enum`s are distinct types whose values are restricted to ranges of values
  ⇒ Reduced namespace pollution 💸

```
enum class Color { black, white, red };
Color r = Color::red;
switch (r){ ... }
```

- Default underlying type is `int` (can be overridden)
  ⇒ Reasonable default 💸
- They convert to other types only via casting
  ⇒ Strong(er) typed 💸
- Prefer scoped `enum`s to unscoped `enum`s

The Anatomy of C++

C++ Entities

- Types
- Values and Objects
- References
- Function
- Enumerator
- Structured Bindings (C++17) 💸
- Class Members
- Templates
- Template Specializations
- Parameter Packs

- Binds specified names to subobjects or elements
- Only alias/ binding to existing type (similar to references)

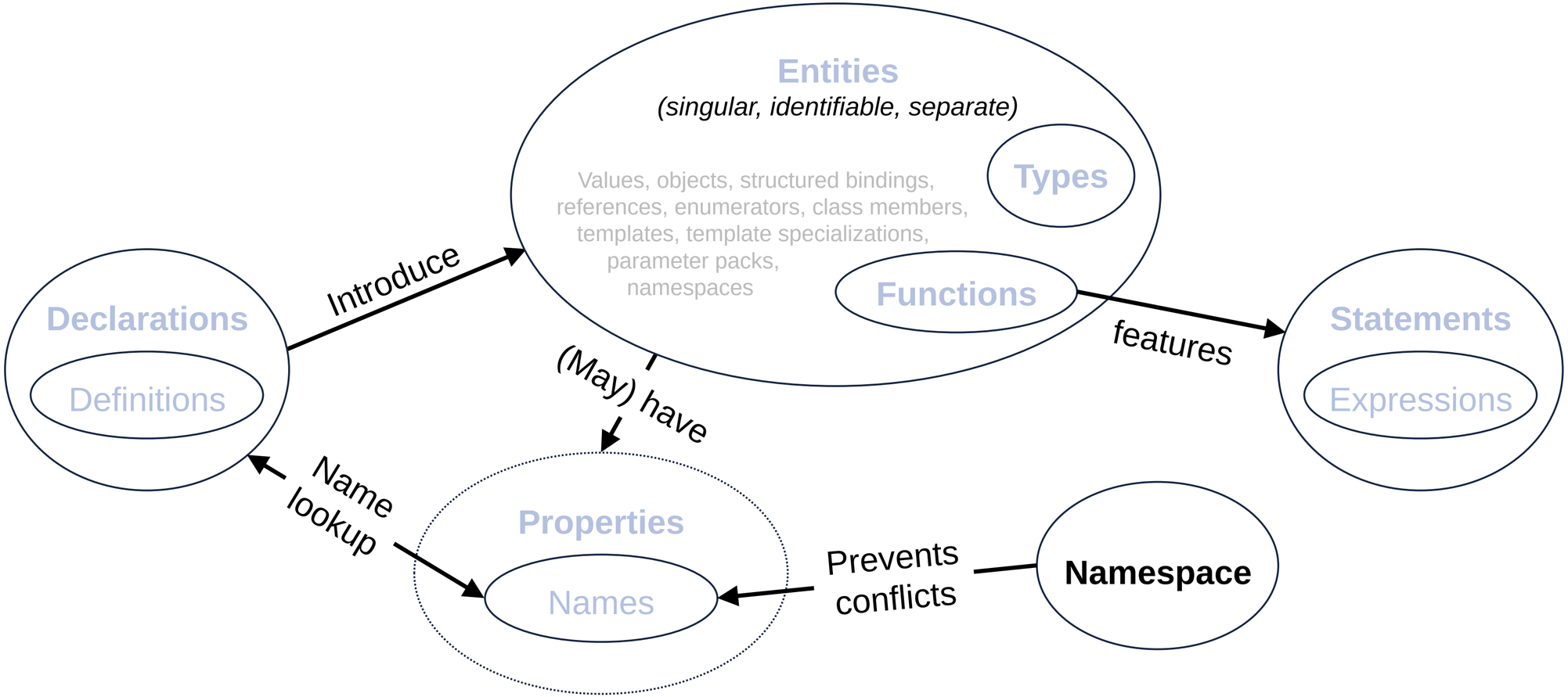
```
std::pair p(666, std::string("Beast"));
// access 666:      std::first;
// access "Beast": std::second
const auto& [nr, name] = p; // structured binding
std::cout << nr << " " << name;
```

- The compiler introduces identifiers (`nr, name`) as names in the surrounding scope and binds them to sub-objects:

```
auto hidden = p; // only visible to compiler
using nr = hidden.first;
using name = hidden.second;
```

- Qualifiers (such as `const` are applied to the entire (hidden) variable, not the identifiers!

The Anatomy of C++



The Anatomy of C++

C++ MAKES IT HARDER TO SHOOT YOURSELF IN THE FOOT…

- Types
- Values and Objects
- References
- Function
- Enumerator
- Structured Bindings
- Class Members
- Templates
- Template Specializations
- Parameter Packs

- C is considered efficient because it is a very low overhead language
- C++ offers zero-cost abstractions💸 on top of this efficiency
  - Reasonable defaults, strong types, no namespace pollution
  - Initialization, scoping, namespaces, scoped enums, references, structured bindings

The Anatomy of C++

...BUT WHEN YOU DO, IT BLOWS YOUR WHOLE LEG OFF!

- Example

```
std::unordered_map<std::string, int> m;
...
for (const std::pair<std::string, int>& p : m){ ... }
```

- Looks alright, however...

Member types

Member type	Definition
key_type	Key
mapped_type	T
value_type	std::pair<const Key, T>

- Compilers will create temporary objects of the type `p` wants
  - each element in `m` is copied, bound to and the copy is destroyed at the end of the loop
- You probably wanted to only bind:
  `for (const auto& p : m) { ... }`

- `auto` variables have their type deduces from their initializers

```
int x1; // maybe unitialized

auto x2; // error: cannot deduce type
         // (requires initialization)

auto x3 = 42; // well-defined; int
```

- Type deduction works similar to templates ↗
- Advantages
  - Prevents uninitialization mistakes
  - Saves typing
  - Can represent types known to compilers only
  - Still strong types

Takeaways

Takeaways

- C is not perfect
- C++ is not C and C is not C++
- Differences include
  - Namespaces and scoping
  - References
  - Operator overloading
  - auto
  - ...
- Many basic concepts in C++ are handled similarly to or in the same way as in C
- Basic concepts include declarations, entities, names, namespaces, statements, expressions
- Reading and understanding cppreference.com
- C++ *can* be nice
  - `void f(const BLOB& b) { ... }`
  - `for (const auto& p : m) { ... }`
- Performance-wise
  - Zero-cost abstraction
  - Avoid unnecessary allocations
  - Prefer pass-by-reference *(where practical)*
  - Avoid conversions *(where practical)*
  - Efficient code does not need to look ugly
- Consider the human factor
  - Debugging pointers
  - Spotting (implicit) type conversions when reading code
  - Keeping track of multiple objects and types
  - Annoyance caused by syntactic contortions
  - ...

Thank You!