Something about C++

First part

Author / **Dmitriy Dyakov, Igor Sadchenko**

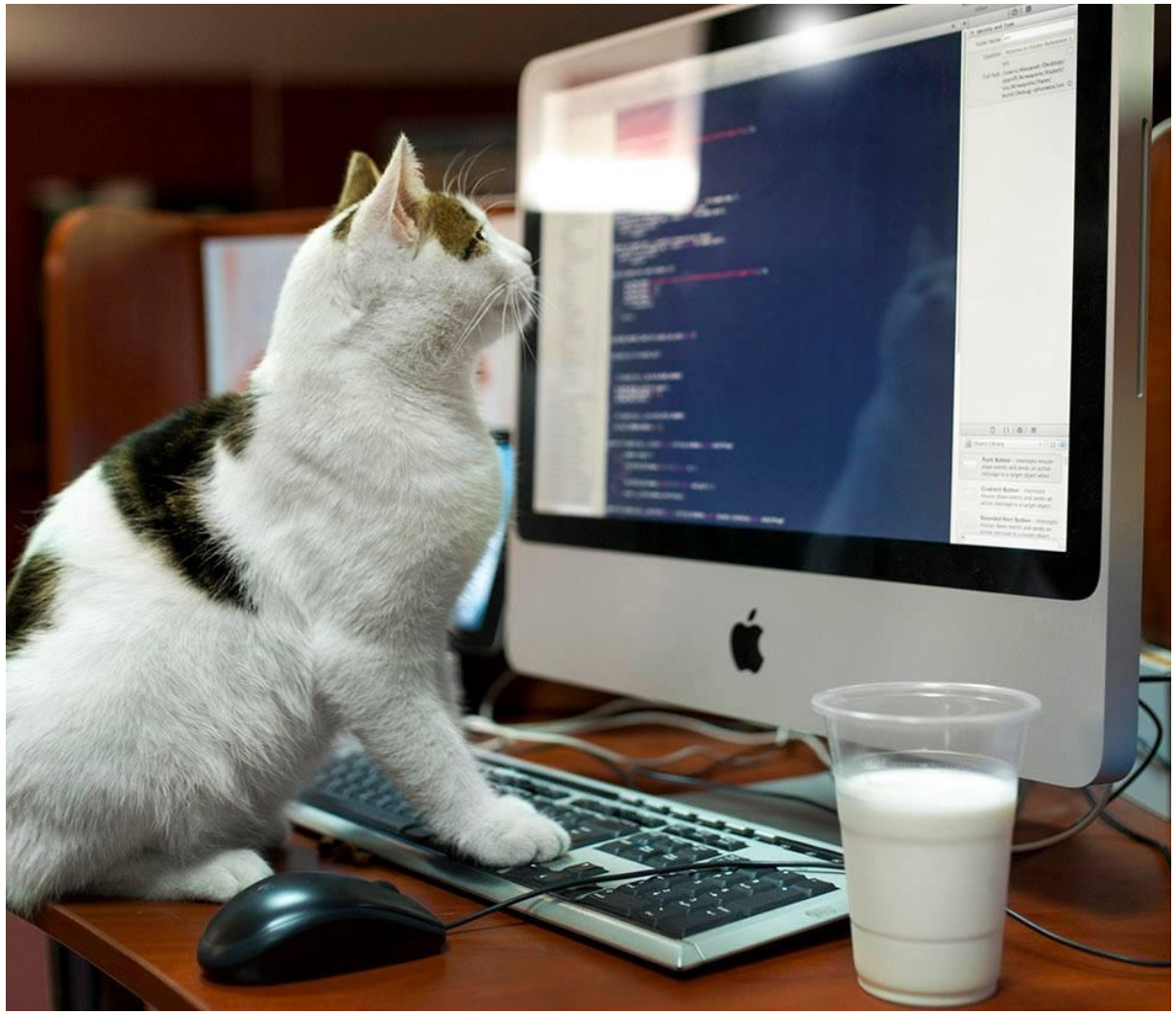
Department / WoT Tools

Contacts /d_dyakov@wargaming.net

2

WARGAMING.NET
LET'S BATTLE

Почему C++?



3

WARGAMING.NET
LET'S BATTLE

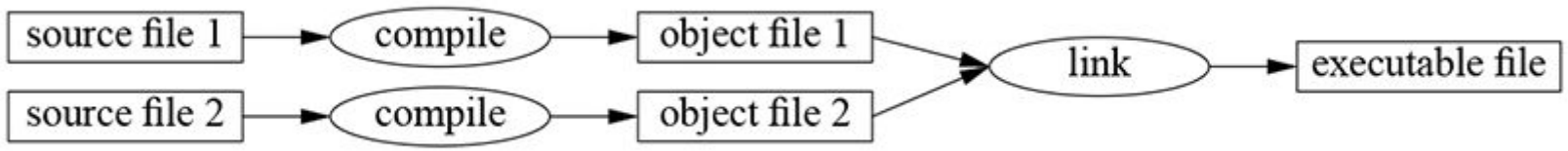
C++

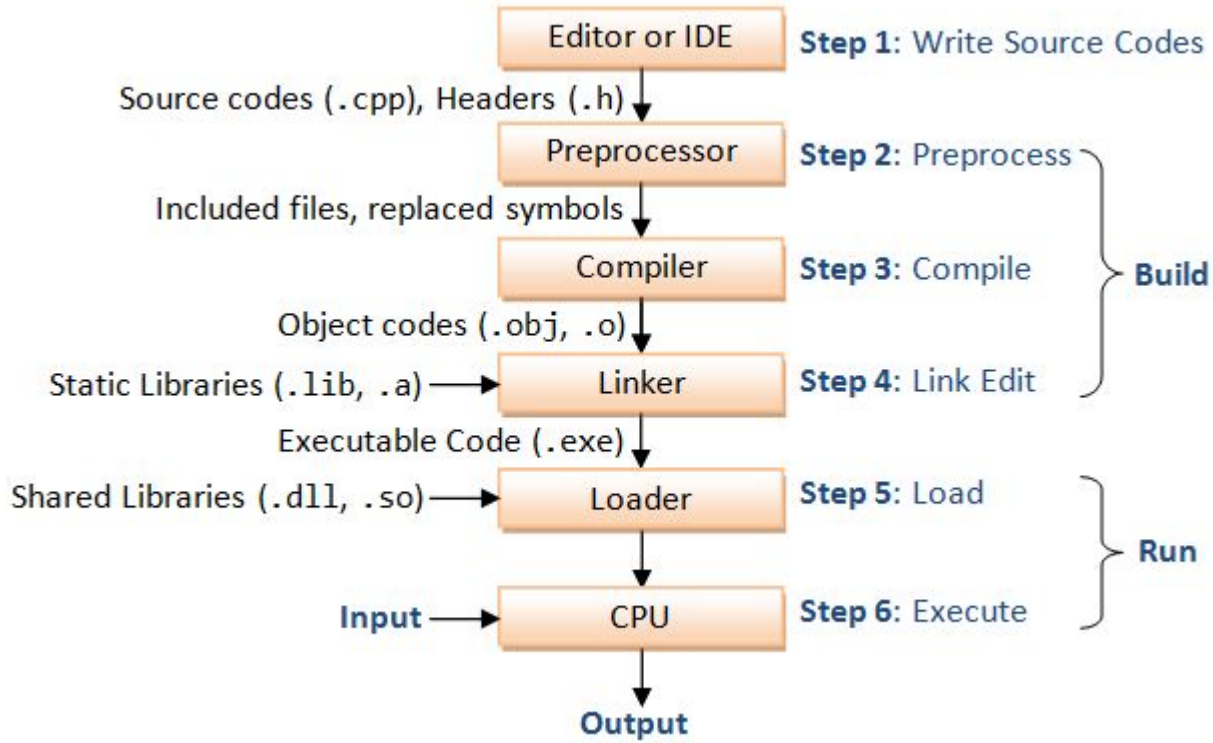
- Author: Bjarne Stroustrup
- First appeared in 1985
- Last standard: C++17 (C++20 in preview)

- C++ has:
  - object-oriented programming features
  - generic programming features
  - functional programming features
  - facilities for low-level memory manipulation

WARGAMING.NET
LET'S BATTLE

Processing C++ program





WARGAMING.NET
LET'S BATTLE

C++ program example

```
#include <iostream>      // include ("import") the declarations for I/O stream library
using namespace std;     // make names from std visible without std::
double square(double x) // square a double precision floating-point number
{
    return x * x;
}

void print_square(double x) // function definition
{
    cout << "the square of" << x << "is" << square(x) << "\n";
}
int main()
{
    print_square(1.234); // print: the square of 1.234 is 1.52276
}
```

6

WARGAMING.NET
LET'S BATTLE

Types

```
bool // Boolean, possible values are true and false

char // character, for example, 'a', 'z', and ‘9’

int // integer, for example, -273, 42, and 1066

double // double-precision floating-point number, for example, -273.15, 3.14, and 6.626e-34

unsigned // non-negative integer, for example, 0, 1, and 999 (use for bitwise logical
operations)
```

7

Types

WARGAMING.NET
LET'S BATTLE

Type	Size in bits	Format	Value range	
			Approximate	Exact
character	8	signed		**-128** to **127**
		unsigned		**0** to **255**
	16	unsigned		**0** to **65535**
	32	unsigned		**0** to **1114111** (**0x10ffff**)
integer	16	signed	**± 3.27 · 10^4**	**-32768** to **32767**
		unsigned	**0** to **6.55 · 10^4**	**0** to **65535**
	32	signed	**± 2.14 · 10^9**	**-2,147,483,648** to **2,147,483,647**
		unsigned	**0** to **4.29 · 10^9**	**0** to **4,294,967,295**
	64	signed	**± 9.22 · 10^{18}**	**-9,223,372,036,854,775,808** to **9,223,372,036,854,775,807**
		unsigned	**0** to **1.84 · 10^{19}**	**0** to **18,446,744,073,709,551,615**
floating point	32		•min subnormal: **± 1.401,298,4 · 10^{-45}** •min normal: **± 1.175,494,3 · 10^{-38}** •max: **± 3.402,823,4 · 10^{38}**	•min subnormal: **±0x1p-149** •min normal: **±0x1p-126** •max: **±0x1.fffffep+127**
	64		•min subnormal: **± 4.940,656,458,412 · 10^{-324}** •min normal: **± 2.225,073,858,507,201,4 · 10^{-308}** •max: **± 1.797,693,134,862,315,7 · 10^{308}**	•min subnormal: **±0x1p-1074** •min normal: **±0x1p-1022** •max: **±0x1.fffffffffffffp+1023**

8

WARGAMING.NET
LET'S BATTLE

Initialization

```
double d1 = 2.3;         // initialize d1 to 2.3
double d2{ 2.3 };        // initialize d2 to 2.3
double d3 = { 2.3 };     // initialize d3 to 2.3 (the = is optional with { ... })
complex<double> z = 1; // a complex number with double-precision floating-point scalars
complex<double> z2{ d1, d2 };
complex<double> z3 = { d1, d2 }; // the = is optional with { ... }

vector<int> v{ 1, 2, 3, 4, 5, 6 }; // a vector of ints
```

WARGAMING.NET
LET'S BATTLE

Constants

- ***const***: meaning roughly "I promise not to change this value."
  This is used primarily to specify interfaces so that data can be passed to functions using pointers and references without fear of it being modified.
  The compiler enforces the promise made by *const*. The value of a *const* can be calculated at run time.

- ***constexpr***: meaning roughly "to be evaluated at compile time." This is used primarily to specify constants, to allow placement of data in read-only memory (where it is unlikely to be corrupted), and for performance. The value of a *constexpr* must be calculated by the compiler.

10

WARGAMING.NET
LET'S BATTLE

Constants

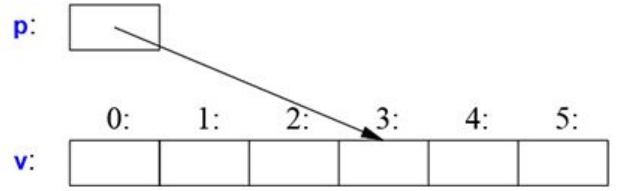
```
constexpr int dmv = 17;         // dmv is a named constant
int var = 17;                   // var is not a constant
const double sqv = sqrt(var); // sqv is a named constant, possibly computed at run
time

double sum(const vector<double>&); // sum will not modify its argument

vector<double> v{ 1.2, 3.4, 4.5 }; // v is not a constant
const double s1 = sum(v);          // OK: sum(v) is evaluated at run time
constexpr double s2 = sum(v);      // error: sum(v) is not a constant expression
```

11

WARGAMING.NET
LET'S BATTLE

Pointers, arrays, and references

```
char* p = &v[3]; // p points to v's fourth element
char x = *p;     // *p is the object that p points to
```




```
T a[n];  // T[n]: a is an array of n Ts
T* p;    // T*: p is a pointer to T
T& r;    // T&: r is a reference to T
T f(A);  // T(A): f is function taking argument of A returning a result of type T

double* pd = nullptr;
Link<Record>* lst = nullptr; // pointer to a Link to a Record
int x = nullptr;             // error: nullptr is a pointer not an integer
```

12

WARGAMING.NET
LET'S BATTLE

Advice

For all the good, against all the bad

- Don’t panic! All will become clear in time
- You don’t have to know every detail of C++ to write good programs.
- Focus on programming techniques, not on language features.
- For the final word on language definition issues, see the ISO C++ standard;
- A function should perform a single logical operation
- Keep functions short
- Use overloading when functions perform conceptually the same task on different types
- If a function may have to be evaluated at compile time, declare it *constexpr*
- Understand how language primitives map to hardware
- Use digit separators to make large literals readable;

13

WARGAMING.NET
LET'S BATTLE

Advice

For all the good, against all the bad

- Avoid complicated expressions
- Avoid narrowing conversions
- Minimize the scope of a variable
- Avoid “magic constants”; use symbolic constants
- Prefer immutable data
- Declare one name (only) per declaration
- Keep common and local names short, and keep uncommon and nonlocal names longer
- Avoid similar-looking names
- Prefer the *{}*-initializer syntax for declarations with a named type
- Use *auto* to avoid repeating type names

14

WARGAMING.NET
LET'S BATTLE

Advice

For all the good, against all the bad

- Avoid uninitialized variables
- Keep scopes small
- When declaring a variable in the condition of an *if*-statement, prefer the version with the implicit test against *0*
- Use unsigned for bit manipulation only
- Keep use of pointers simple and straightforward
- Use *nullptr* rather than *0* or *NULL*
- Don't declare a variable until you have a value to initialize it with
- Don't say in comments what can be clearly stated in code
- State intent in comments
- Maintain a consistent indentation style

15

RAII

WARGAMING.NET
LET'S BATTLE

Resource Acquisition Is Initialization or RAII can be summarized as follows:

- encapsulate each resource into a class, where
  - the constructor acquires the resource and establishes all class invariants or throws an exception if that cannot be done,
  - the destructor releases the resource and never throws exceptions;

- always use the resource via an instance of a RAII-class that either
  - has automatic storage duration or temporary lifetime itself, or
  - has lifetime that is bounded by the lifetime of an automatic or temporary object

16

WARGAMING.NET
LET'S BATTLE

Structs & classes

```
class Vector {
public:
    Vector(int s) : elem{new double[s]}, sz{s} { } // construct a Vector
    double& operator[](int i) { return elem[i]; }  // element access: subscripting
    int size() { return sz; }
private:
    double* elem; // pointer to the elements
    int sz;       // the number of elements
};
```

17

Unions

WARGAMING.NET
LET'S BATTLE

```
// a Type can hold values ptr and num
enum Type { ptr, num };

struct Entry {
    string name; // string is a std type
    Type t;
    Node* p; // use p if t==ptr
    int i;   // use i if t==num
};

void f(Entry* pe)
{
    if (pe->t == num)
        cout << pe->i;
    // ...
}
```

```
union Value {
    Node* p;
    int i;
};

struct Entry {
    string name;
    Type t;
    Value v; // use v.p if t==ptr or v.i if t==num
};

void f(Entry* pe)
{
    if (pe->t == num)
        cout << pe->v.i;
    // ...
}
```

18

WARGAMING.NET
LET'S BATTLE

Enumerations

```
enum class Color { red, blue, green };
enum class Traffic_light { green, yellow, red };

Color col = Color::red;
Traffic_light light = Traffic_light::red;

Color x = red;                // error: which red?
Color y = Traffic_light::red; // error: that red is not a Color
Color z = Color::red;         // OK


int i = Color::red; // error: Color::red is not an int
Color c = 2; // initialization error: 2 is not a Color

Color x = Color{ 5 }; // OK, but verbose
Color y{ 6 };         // also OK
```

```
enum Color { red, green, blue };
int col = green;
// The enumerators from a “plain” enum
// are entered into the same scope as the
// name of their enum and implicitly
// converts to their integer value
```

WARGAMING.NET
LET'S BATTLE

Advice

For all the good, against all the bad

- Prefer well-defined user-defined types over built-in types when the built-in types are too low-level
- Organize related data into structures (*struct*s or *class*es)
- Represent the distinction between an interface and an implementation using a *class*
- A *struct* is simply a *class* with its members public by default
- Define constructors to guarantee and simplify initialization of *class*es
- Avoid “naked” *union*s; wrap them in a class together with a type field
- Use *enum*erations to represent sets of named constants
- Prefer *class enums* over “plain” *enums* to minimize surprises
- Define operations on enumerations for safe and simple use

Separate compilation

WARGAMING.NET
LET'S BATTLE

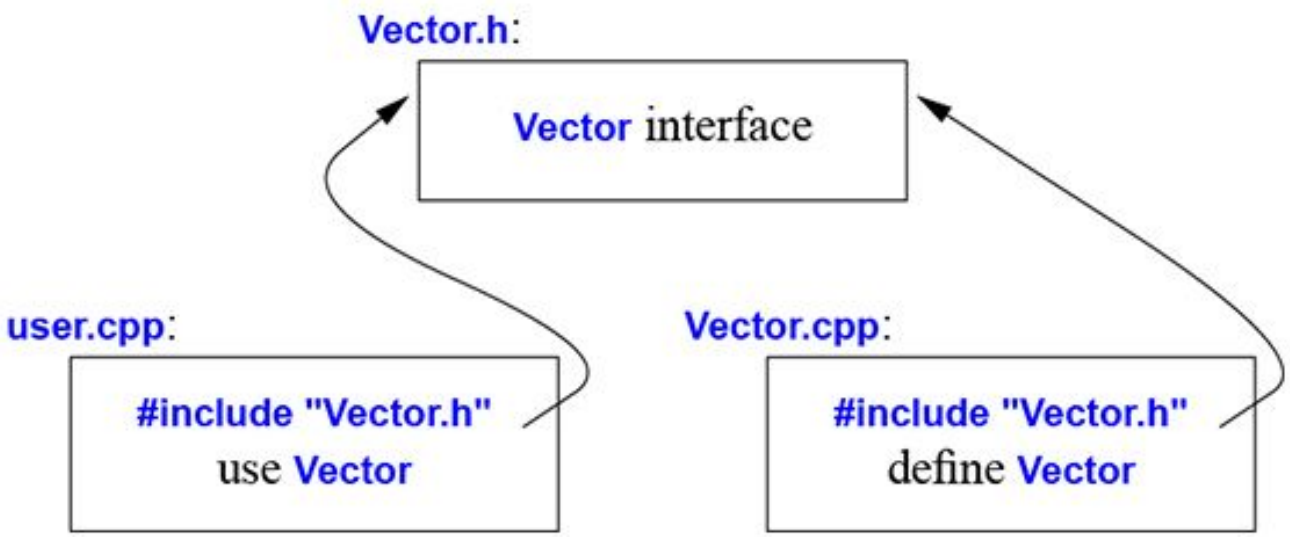
```
// Vector.h:

class Vector
{
public:
    Vector(int s);
    double& operator[](int i);
    int size();

private:
    double* elem;
    int sz;
};
```

```
// user.cpp:

#include "Vector.h" // get Vector's interface

#include <cmath> // get the standard-library math function
interface including sqrt()

double sqrt_sum(Vector& v)
{
    double sum = 0;
    for (int i = 0; i != v.size(); ++i)
        sum += std::sqrt(v[i]); // sum of square roots
    return sum;
}
```

```
// Vector.cpp:

#include "Vector.h" // get Vector's interface

Vector::Vector(int s) : elem{ new double[s] }, sz{ s }
{ }

double& Vector::operator[](int i)
{ return elem[i]; }

int Vector::size()
{ return sz; }
```

WARGAMING.NET
LET'S BATTLE

Separate compilation



22

Modules(C++20)

WARGAMING.NET
LET'S BATTLE

```
// file Vector.cpp:
module; // this compilation will define a module
// ... here we put stuff that Vector need for implementation...

export module Vector; // defining the module called "Vector"

export class Vector
{
public:
    Vector(int s);
    double& operator[](int i);
    int size();
private:
    double* elem;
    int sz;
};

// ...vector implementation...

export int size(const Vector& v)
{ return v.size(); }
```

```
// file user.cpp:

import Vector;   // get Vector's interface
#include <cmath> // get the std math function interface

double sqrt_sum(Vector& v)
{
    double sum = 0;
    for (int i = 0; i != v.size(); ++i)
        sum += std::sqrt(v[i]); // sum of square roots
    return sum;
}
```

23

WARGAMING.NET
LET'S BATTLE

Modules(C++20)

- The differences between headers and modules are not just syntactic.
  - A module is compiled once only (rather than in each translation unit in which it is used).
  - Two modules can be imported in either order without changing their meaning.
  - If you import something into a module, users of your module do not implicitly gain access to (and are not bothered by) what you imported: import is not transitive.
- The effects on maintainability and compile-time performance can be spectacular.

24

Namespaces

WARGAMING.NET
LET'S BATTLE

```
namespace My_code {
class complex
{ /* ... */ };
complex sqrt(complex);
// ...
int main();
}

int My_code::main()
{
    complex z{ 1, 2 };
    auto z2 = sqrt(z);
    std::cout << '{' << z2.real() << ',' << z2.imag() << "}\n";
    // ...
}

int main()
{
    return My_code::main();
}
```

25

WARGAMING.NET
LET'S BATTLE

Exceptions

```
#include <exception>

Vector::Vector(int s)
{
    if (s < 0)
        throw std::length_error{ "Vector constructor: negative
size" };
    elem = new double[s];
    sz = s;
}
```

```
#include <exception>

void test()
{
    try
    {
        Vector v(-27);
    }
    catch (std::length_error& err)
    {
        // handle negative size
    }
    catch (std::bad_alloc& err)
    {
        // handle memory exhaustion
    }
    catch (...)
    {
        // handle all exceptions
    }

}
```

WARGAMING.NET
LET'S BATTLE

Error-Handling Alternatives

- Function can indicate that it cannot perform its allotted task by:
  - throwing an exception
  - somehow return a value indicating failure
  - terminating the program (by invoking a function like *terminate()*, *exit()*, or *abort()*).

27

Assertions

WARGAMING.NET
LET'S BATTLE

```
void f(const char* p)
{
    // run-time checking
    assert(p != nullptr); // p must not be the nullptr
                          // ...
}

constexpr double C = 299792.458; // km/s

void f(double speed)
{
    constexpr double local_max = 160.0 / (60 * 60); // 160 km/h == 160.0/(60*60) km/s
    // compile-time checking
    static_assert(speed < C, "can't go that fast");     // error: speed must be a constant
    static_assert(local_max < C, "can't go that fast"); // OK

    // ...
}
```

WARGAMING.NET
LET'S BATTLE

Function Argument Passing

```
void test(vector<int> v, vector<int>& rv)
// v is passed by value; rv is passed by reference
{
    v[1] = 99;  // modify v (a local variable)
    rv[2] = 66; // modify whatever rv refers to
}

int main()
{
    vector<int> fib = { 1, 2, 3, 5, 8, 13, 21 };
    test(fib, fib);
    cout << fib[1] << ' ' << fib[2] << '\n';
    // prints 2 66
}
```

Usually pass small values by-value and larger ones by-reference. Here "small" means "something that's really cheap to copy." Exactly what "small" means depends on machine architecture, but "the size of two or three pointers or less" is a good rule of thumb.

Function Value Return


```
class Vector
{
public:
    // ...
    // return reference to ith element
    double& operator[](int i) { return elem[i]; }
private:
    double* elem; // elem points to an array of sz
};

int& bad()
{
    int x;
    // ...
    // bad: return a reference to local variable x
    return x;
}
```

The default for value return is to copy and for small objects that's ideal. Return "by reference" only when we want to grant a caller access to something that is not local to the function

30

WARGAMING.NET
LET'S BATTLE

Structure Binding

```
struct Entry
{
    string name;
    int value;
};

Entry read_entry(istream& is) // naive read
function (for a better version, see §10.5)
{
    string s;
    int i;
    is >> s >> i;
    return { s, i };
}
```

```
auto e = read_entry(cin);
cout << "{ " << e.name << "," << e.value << " }\n";

auto [name, value] = read_entry(is);
cout << "{ " << name << "," << value << " }\n";
```

31

WARGAMING.NET
LET'S BATTLE

Structure Binding

```
map<string, int> m;
// ... fill m ...
for (const auto [key, value] : m)
    cout << "{" << key "," << value << "}\n";
```

```
void incr(map<string, int>& m)
// increment the value of each element of m
{
    for (auto& [key, value] : m)
        ++value;
}
```

WARGAMING.NET
LET'S BATTLE

Advice

For all the good, against all the bad

- Distinguish between declarations (used as interfaces) and definitions (used as implementations)
- Use header files to represent interfaces and to emphasize logical structure
- *#include* a header in the source file that implements its functions
- Avoid non-inline function definitions in headers
- Prefer modules over headers (where modules are supported)
- Use namespaces to express logical structure
- Use *using*-directives for transition, for foundational libraries (such as *std*), or within a local scope
- Don't put a using-directive in a header file

WARGAMING.NET
LET'S BATTLE

Advice

For all the good, against all the bad

- Use error codes when an immediate caller is expected to handle the error
- Throw an exception if the error is expected to percolate up through many function calls
- If in doubt whether to use an exception or an error code, prefer exceptions
- Develop an error-handling strategy early in a design
- Use purpose-designed user-defined types as exceptions (not built-in types)
- Don't try to catch every exception in every function
- Prefer RAII to explicit *try*-blocks

WARGAMING.NET
LET'S BATTLE

Advice

For all the good, against all the bad

- What can be checked at compile time is usually best checked at compile time
- Pass “small” values by value and “large“ values by references
- Prefer pass-by-*const*-reference over plain pass-by-reference;
- Return values as function-return values (rather than by out-parameters)
- Don’t overuse return-type deduction
- Don’t overuse structured binding; using a named return type is often clearer documentation

WARGAMING.NET
LET'S BATTLE

Links

-
-
-