



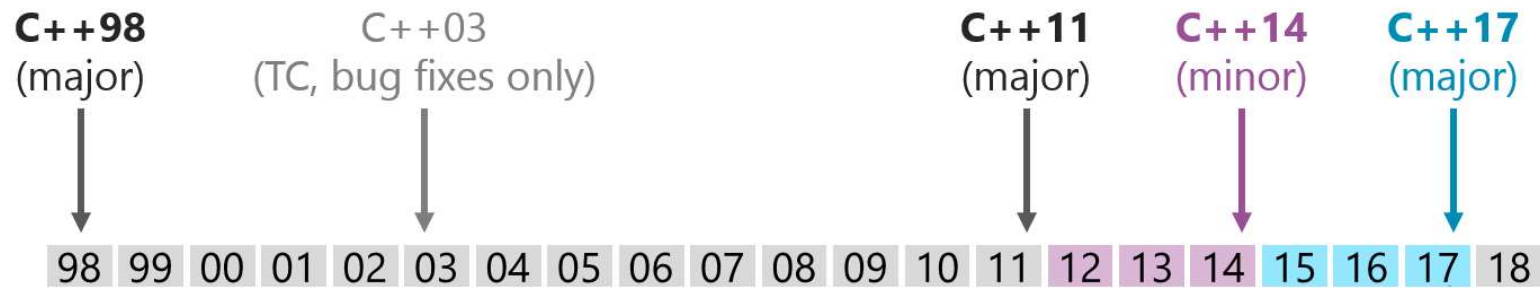
ISTITUTO ITALIANO
DI TECNOLOGIA

Modern C++: core features

For us

Modern C++

It is what is after C++98/03: C++11, C++14, and C++17



Core features

We shall see some of the core features which help us to at least read some modern C++ code:

- nullptr, auto, type aliases, initializer list, uniform initialization, scoped enumerations, range-based loops, constexpr, override and final.

Prerequisites

To fully understand the course it is required:

- Basic knowledge of standard C++98

Prerequisites: C++98

Very basic startup material:

- <http://www.cplusplus.com/doc/oldtutorial/>

Something more:

- Thinking in C++, Bruce Eckel
<https://www.micc.unifi.it/bertini/download/programmazione/TICPP-2nd-ed-Vol-one-printed.pdf>
- Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Scott Meyers

Modern C++

- **nullptr,**
- auto,
- type aliases,
- initializer list,
- uniform initialization,
- range-based loops,
- constexpr,
- scoped enumerations,
- override and final.

Use `nullptr`

Since C++11, we prefer the keyword `nullptr` over 0 or NULL to express a dummy pointer.

Use `nullptr` because:

- The literal 0 (NULL) is an int, not a pointer and could cause ambiguity in overloaded function resolution.
- It improves code clarity.

Use `nullptr`

Example

```
void f(int a);  
void f(char *a);  
void test() {           // on compiler armclang w/ both -std=c++98/c++14/c++17  
    f(0);               // calls f(int a)  
    f(NULL);            // calls f(int a): it is not what we wanted  
    f(nullptr);         // calls f(char *a)  
    auto result = findRecord( /* arguments */ );  
    if(nullptr == result) { // there's no ambiguity:  
        ....              // result must be a pointer type  
    }  
}
```

`nullptr`

Modern C++

- nullptr,
- **auto,**
- type aliases,
- initializer list,
- uniform initialization,
- range-based loops,
- constexpr,
- scoped enumerations,
- override and final.

Use `auto` whenever possible

The new C++ standards allows the use of `auto` as a placeholder for types in various contexts and let the compiler deduce the actual type.

An actual term was coined for this by Andrei Alexandrescu and promoted by Herb Sutter: *almost always auto (AAA)*.

Use **auto** ...

To declare local variables when you do not want to commit to a specific type with the form:

```
auto name = expression
```

Example

```
auto i = 42; // int  
auto d = 42.5; // double  
auto s = "text"; // char const *  
auto v = { 1, 2, 3 }; // std::initializer_list<int>
```

almost always **auto** (AAA).



Use **auto** ...

To declare local variables when you need to commit to a specific type with the form:

```
auto name = type-id { expression }
```

Example

```
auto b = new char[10]{ 0 }; // char*  
auto s1 = std::string {"text"}; // std::string  
auto v1 = std::vector<int> { 1, 2, 3 }; // std::vector<int>  
auto p = std::make_shared<int>(42); // std::shared_ptr<int>
```

almost always **auto** (AAA).



Use **auto** ...

The **auto** can be used to specify the return type from a function. In C++11, this requires a trailing return type in the function declaration. In C++14, the type of the return value is deduced by the compiler.

Example

```
auto add(const int a, const int b) -> int { return a+b; }    // C++11
```

```
auto multiply(const int a, const int b) { return a*b; }      // C++14
```

almost always **auto** (AAA).



Use **auto** ...

To declare named lambda functions, with the form:

`auto name = lambda-expression`

Example

```
auto half = [](const double v) { return v/2.0; };  
auto result = half(8.0);
```

almost always **auto** (AAA).



Use **auto** ...

To declare lambda parameters and return values
(since C++14).

Examples

```
auto half = [](const auto a) { return a/2; };  
constexpr double ninepointzero = 9.0;  
constexpr int eleven = 11;  
auto result1 = half(ninepointzero);           // it is 4.5 (double)  
auto result2 = half(eleven);                   // it is 5 (int)
```

almost always **auto** (AAA).



Benefits of **auto**

- It is not possible to leave a variable uninitialized.

```
auto ihaveavalue = 3; // int
```

```
auto idonthaveavalue; // ERROR: its content is unpredictable
```

- Ensures that you always use the correct type and that implicit conversion will not occur

```
auto v = std::vector<int>{ 1, 2, 3 };
```

```
int size1 = v.size(); // actually size() returns size_t ...
```

```
auto size2 = v.size(); // correct type size_t
```

almost always **auto** (AAA).



Benefits of `auto`

- It means less typing and less concern for types that we don't care about

```
std::map<int, std::string> m; // given this map ...
```

```
// we just want to iterate
```

```
for (auto it = m.cbegin(); it != m.cend(); ++it)
```

```
{ /* ... */ }
```

```
// we don't care about the type of the iterator it
```

```
for(std::map<int, std::string>::const_iterator it = m.cbegin();
```

```
it != m.cend(); ++it)
```

```
{ /* ... */ }
```

almost always `auto` (AAA).



Perils of **auto**

- The user may write obscure code

```
auto var = obtainit();  
useit(var);
```

WHAT is var ????



almost always **auto** (AAA).



Almost always **auto** but

- Use auto wherever you believe it enhances code readability BUT ... use it well and avoid it where it obscures the intent of the code

Modern C++

- nullptr,
- auto,
- **type aliases,**
- initializer list,
- uniform initialization,
- range-based loops,
- constexpr,
- scoped enumerations,
- override and final.

Use using

In C++, we can create synonyms for type name with a `typedef` declaration. However, `typedef` has some limitations (e.g., it cannot be used with templates).

Since C++11, there is a new **using** syntax for type aliases and alias templates.

Use **using**

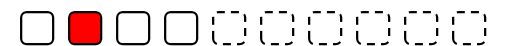
To create type aliases with the form:

Using alias-name = type-id

Example

```
using byte = unsigned char;  
using array10b_t = byte[10];  
using fpWorker = void (*)(void);  
void dosomething() { static array10b_t data = {0}; data[0]++; }  
void caller(fpWorker f) { f(); }  
void f1() { caller(dosomething); }
```

using



Use **using**

To create alias templates with the form :

```
template<template-params-list>  
using alias-name = type-id
```

Example

```
template <typename T>  
using vect_t = std::vector<T>;
```

```
vect_t<std::string> vs;  
vect_t<int> vi;
```

using

Use **using**

Example

```
// FP is a synonym for a pointer to a function taking an int and  
// a const std::string& and returning nothing  
typedef void (*FP)(int, const std::string&); // typedef
```

```
// same meaning as above
```

```
using FP = void (*)(int, const std::string&); // alias declaration
```


Modern C++

- nullptr,
- auto,
- type aliases,
- **initializer list,**
- uniform initialization,
- range-based loops,
- constexpr,
- scoped enumerations,
- override and final.

Initializer list

An object of type `std::initializer_list<T>` is a lightweight proxy that provides access to an array of objects of type `const T`.

Example

```
void afun(const std::initializer_list<int> &l);  
auto ilis = { 10, 20, 30 }; // the type of ilis is initializer_list<int>  
afun(ilis);  
auto ilistr = { "hello", "world" };
```

Initializer list

The `std::initializer_list<T>` is used as a constructor for the std containers. As such, it has wide use in modern C++.

Example

```
std::array<int, 3> a3 = {1, 2, 3};  
std::map<int, std::string> mm1 = { {1, "one"}, {2, "two"} };  
std::map<int, std::string> mm2{ {3, "three"}, {4, "four"} };
```

Initializer list

Example

```
template <class T>
struct S {
    std::vector<T> v;
    S(std::initializer_list<T> l) : v(l) { std::cout << "size = " << l.size(); }
    void append(std::initializer_list<T> l) { v.insert(v.end(), l.begin(), l.end()); }
};
```

```
S<int> s = {1, 2, 3, 4, 5}; // copy list-initialization
s.append({6, 7, 8}); // list-initialization in function call
```

Initializer list

Actual use in YARP

YARP_OS: add initializer_list constructor for Property.

Example:

```
Property p {{ "one", Value(1) },  
            { "two", Value(2.0) },  
            { "string", Value("foo") }  
};
```

 devel (#2007)



Nicogene committed 29 days ago

```
60      60  
61      +    /**  
62      +    * @brief Initializer list constructor.  
63      +    * @param[in] values, list of std::pair with which initialize the Property.  
64      +    */  
65      +    Property(std::initializer_list<std::pair<std::string, yarp::os::Value>> values);
```

initializer list

□□□■□□□□□□□□

Initializer list

Actual use in embedded software of iCub

```
162
163 static void start_evt_based(void)
164 {
165     // pulse led one at 1 hertz
166     embot::app::theLEDmanager &theleds = embot::app::theLEDmanager::getInstance();
167     theleds.init( { embot::hw::LED::one, embot::hw::LED::two } );
168     theleds.get(embot::hw::LED::one).pulse(embot::common::time1second);
169
class theLEDmanager
{
public:
    static theLEDmanager& getInstance();

    bool init(const std::initializer_list<embot::hw::LED> &leds);
    bool deinit(const std::initializer_list<embot::hw::LED> &leds);
    bool deinit();

    bool initialised(embot::hw::LED led) const;
    LEDhandle & get(embot::hw::LED led) const;

    ~theLEDmanager();

private:
    theLEDmanager();
    struct Impl;
    std::unique_ptr<Impl> pImpl;
};
```

initializer list

□□□□□■□□□□□□

Modern C++

- nullptr,
- auto,
- type aliases,
- initializer list,
- **uniform initialization,**
- range-based loops,
- constexpr,
- scoped enumerations,
- override and final.

Uniform-initialization

C++11 has a uniform syntax that can be used anywhere, the braced-initialization: { }

It can be used in direct initialization and also in copy initialization:

direct initialization

```
int i{2};
```

```
int[] ai{2,3};
```

```
vector<int> vi{1, 2, 3};
```

copy initialization

```
int i = {2};
```

```
int ai[] = {2, 3};
```

```
vector<int> vi = {1, 2, 3};
```


Uniform-initialization

Example w/ built-in types and arrays:

```
int i { 42 }; double d { 1.2 };  
int arr1[3] { 1, 2, 3 }; int* arr2 = new int[3]{ 1, 2, 3 };
```

Example w/ standard containers:

```
std::vector<int> v { 1, 2, 3 };  
std::vector<int> v1 {};  
std::map<int, std::string> m { {1, "one"}, { 2, "two" } };
```

In C++98, it is possible only w/ `std::vector` and integral types:

```
int arr[] = {3, 9, 27, 81};  
std::vector<int> vec98 (arr, arr + sizeof(arr) / sizeof(arr[0]) );
```

}

Uniform-initialization

Example w/ user-defined types:

```
class FAQ {  
    std::string question {"*"};  
    int answer {42};  
public:  
    FAQ() = default;  
    FAQ(const std::string &q, int a) : question(q), answer(a) {}  
};  
void test() {  
    FAQ f0{};  
    FAQ f2{ "what?", 42};  
    {}  
}
```

Uniform-initialization

All standard containers have an additional constructor in C++11 that takes an argument `std::initializer_list<T>`

An initializer list always takes precedence over other constructors where brace-initialization is used.

```
struct fee {  
    std::array<int, 2> a {0, 0};  
    fee() = default;  
    fee(int v1, int v2 = 1) : a{v1, v2} {} // useless  
    fee(std::initializer_list<int> l) { if(l.size() == 2){ ... } }  
};  
fee f{ 1, 2 }; // calls constructor with initializer_list<int>;
```

{ }

□□□■□□□□□□□□

Uniform-initialization

Caveat:

- The precedence of `std::initializer_list<>` may lead to bugs:

```
std::vector<int> v {5, 1}; // a vector of 2 elements: 5 and 1
```

```
std::vector<int> v (5, 1); // a vector of 5 elements with value 1
```

- brace-initialization does not allow narrowing conversion:

```
int i{ 1.2 };           // error
```

```
double d = 47 / 13;
```

```
float f1{ d };          // error
```

```
float f2{47/13};        // OK
```

{ }

□□□□□■□□□□□□

Modern C++

- Hidden track: `std::array<>`

std::array<>

The std::array is a container that encapsulates fixed size arrays.

- std::array<T, N> an aggregate type with the same semantics as a struct holding a C-style array T[N] as its only non-static data member.

```
std::array<int, 2> data0 {1, 2};
```

```
struct ARRAY {  
    int memory[2];  
    // other code  
};
```

```
ARRAY data1 {1, 2};
```

```
int value = data0[1];
```

```
int value = data1.memory[1];
```

std::array<>

- the struct combines the performance and accessibility of a C-style array with the benefits of a standard container, such as knowing its own size, supporting assignment, random access iterators, etc.

```
std::array<int, 4> data0;
```

```
size_t pos = getpositionsomewhere();  
if(pos < data0.size())  
{  
    data0[pos]++;  
}
```

std::array<>

```
std::array<int, 4> data0 {};
```

```
Data0.fill(41);
```

```
for(auto &v : data0)
{
    v++;
}
```

hidden track: std::array<>

cppreference.com

Page Discussion

C++ Containers library **std::array**

std::array

Defined in header <array>

```
template<
    class T,
    std::size_t N
> struct array;
```

std::array is a container. This container is an aggregate. It has a static data member. Unlike other containers, it is not initialized with aggregate values.

```
std::array<int, 3> a;
```

The struct combines the features of an array, such as knowing its own size, with those of a container. std::array satisfies the requirements of a container, is empty and that the container is not empty and partially satisfies the requirements of a container. There is a special case for some unique value. The container can also be used as a container.

Member functions

Element access

- array::at
- array::operator[]
- array::front
- array::back
- array::data

Iterators

- array::begin
- array::cbegin
- array::end
- array::cend
- array::rbegin
- array::crbegin
- array::rend
- array::crend

Capacity

- array::empty
- array::size
- array::max_size

Modifiers

- array::fill
- array::swap

Non-member functions

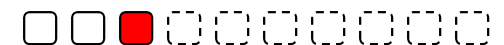
- get
- swap
- operator==
- operator!=
- operator<
- operator>
- operator<=
- operator>=

Helper classes

- tuple_size
- tuple_element

Deduction guides (C++17)

As a rule, iterators to an array are invalidated by swap, however, that during swap, the iterator will continue to point to the same memory location.



Modern C++

- nullptr,
- auto,
- type aliases,
- initializer list,
- uniform initialization,
- **range-based loops,**
- constexpr,
- scoped enumerations,
- override and final.

Range-based loops

In C++11, a range-based for loop has the following general syntax:

```
for ( decl : expr ) stmt;
```

Example

```
std::vector<int> fibo {0, 1, 1, 2, 3, 5, 8, 13};  
std::vector<std::string> names {};  
for(const auto &a : fibo) {  
    names.push_back(std::to_string(a));  
}
```

```
for ( decl : expr ) stmt;
```



Range-based loops

A loop can be non mutating:

```
for (int i : fibo) std::cout << i;  
for (const int i : fibo) std::cout << i; // It is OK, but prefer the const &  
for (const std::string &s : names) std::cout << s;
```

Or mutating:

```
for (int &i : fibo) i++;
```

Also, it can automatically infer the type:

```
for (auto i : fibo) std::cout << i;  
for (const auto i : fibo) std::cout << i; // It is OK, but prefer the const &  
for (const auto &s : names) std::cout << s;  
for (auto &i : fibo) i++;
```

for (decl : expr) stmt;

Range-based loops

It works on any *expr* whose type, or `decltype(expr)`:

- has `begin()` / `end()` member functions (as the STL containers);

```
std::vector<int> fibo{0, 1, 1}; for(const auto &a : fibo) { ... }
```

- or: is of array type;

```
int set[] = {1, 2, 3, 4};  
for(auto &item : set) item++;
```

- or: has existing `begin(decltype(expr))` and `end(decltype(expr))` functions, found exclusively via argument-dependent lookup.

```
MySet set;  
for(const auto &item : set) { /* use the item set */ }  
// it works if we have MySet::begin() / ::end() / related operators ++ and !=
```

`for (decl : expr) stmt;`

□□■□□□□□□□

Range-based loops

The expression `for (decl : expr) stmt;` expands to:

```
{  
    auto&& __range = expr;           // it is a universal reference  
    auto __b = begin(__range);  
    auto __e = end(__range);  
    for ( ; __b != __e; ++__b ) {  
        decl = *__b;  
        stmt;  
    }  
}
```

`for (decl : expr) stmt;`

Range-based loops

Example (compiled w/ armclang and executed)

```
volatile int ciao[] = {1, 2, 3, 4, 5, 6, 7, 8}; // placed in RAM @ 0x2000A9B4
// for(auto &item : ciao) item++; // is equivalent to following code:
{
    auto && __range = ciao;    // it is a ref to int[8] and points 0x2000A9B4
    auto __b = std::begin(__range); // it is a int * = 0x2000A9B4
    auto __e = std::end(__range); // it is a int * = 0x2000A9D4
    for ( ; __b != __e; ++__b ) {
        auto &item = *__b; // is the reference to each value inside ciao[]
        item++;
    }
}
```

for (decl : expr) stmt;

□□□□□■□□□□□□□□

Range-based loops

Example (yarp vector **TODO**)

```
volatile int ciao[] = {1, 2, 3, 4, 5, 6, 7, 8}; // placed in RAM @ 0x2000A9B4
// for(auto &item : ciao) item++; // is equivalent to following code:
{
    auto && __range = ciao;    // it is a ref to int[8] and points 0x2000A9B4
    auto __b = std::begin(__range); // it is a int * = 0x2000A9B4
    auto __e = std::end(__range); // it is a int * = 0x2000A9D4
    for ( ; __b != __e; ++__b ) {
        auto &item = *__b; // is the reference to each value inside ciao[]
        item++;
    }
}
```

for (decl : expr) stmt;

□□□□■□□□□□□□

Modern C++

- nullptr,
- auto,
- type aliases,
- initializer list,
- uniform initialization,
- range-based loops,
- **constexpr**,
- scoped enumerations,
- override and final.

constexpr

The keyword `constexpr` (short for constant expression) can be used to declare compile-time constant objects and functions.

The possibility to evaluate expressions at compile time improves runtime execution because there is less code to run and the compiler can perform additional optimizations.

constexpr

Use **constexpr** to define non-member functions that can be evaluated at compile time.

```
constexpr unsigned int factorial(const unsigned int n)
{
    return n > 1 ? n * factorial(n-1) : 1;
}
```

constexpr

Use **constexpr** to define constructors that can be executed at compile time to initialize constexpr objects.

```
struct P3D {  
    double const x_; double const y_; double const z_;  
    constexpr P3D(const double x = 0, const double y = 0,  
                  const double z = 0) : x_{x}, y_{y}, z_{z} {}  
    constexpr double get_x() const {return x_;}  
    constexpr double get_y() const {return y_;}  
    constexpr double get_z() const {return z_;}  
};
```

constexpr

Use **constexpr** to define variables that can have their values evaluated at compile time.

```
constexpr P3D origin { 1.3, 1.5, 1.7 };  
constexpr unsigned int size = factorial(6);  
char buffer[size] {0};  
constexpr P3D p {0, 1, 2};  
constexpr auto x = p.get_x();
```

constexpr

Use in embedded software in iCub

```

69 namespace embot { namespace hw { namespace bsp {
70
71     struct SUPP
72     {
73         std::uint32_t      supportedmask;
74
75         constexpr SUPP(std::uint32_t m) : supportedmask(m) {}
76
77         template <typename T>
78         constexpr bool supported(T v) const
79         {
80             return embot::binary::bit::check(supportedmask, embot::common::tointegral(v));
81         }
82     };
83
84 }}} // namespace embot
85
86 namespace embot { namespace hw { namespace bsp { namespace pga308 {
87
88     struct PROP
89     {
90         embot::hw::GPIO poweron;
91     };
92
93     struct BSP : public embot::hw::bsp::SUPP
94     {
95         constexpr static std::uint8_t maxnumberof = embot::common::tointegral(embot::hw::PGA308);
96         constexpr BSP(std::uint32_t msk, std::array<const PROP*, maxnumberof> pro) : SUPP(msk),
97         constexpr BSP() : SUPP(0), properties({0}) {}
98
99         std::array<const PROP*, maxnumberof> properties;
100         constexpr const PROP * getPROP(embot::hw::PGA308 h) const { return supported(h) ? properties[h] : 0; }
101         void init(embot::hw::PGA308 h) const;
102     };
103
104     const BSP& getBSP();

```

constexpr

Use in embedded software in iCub

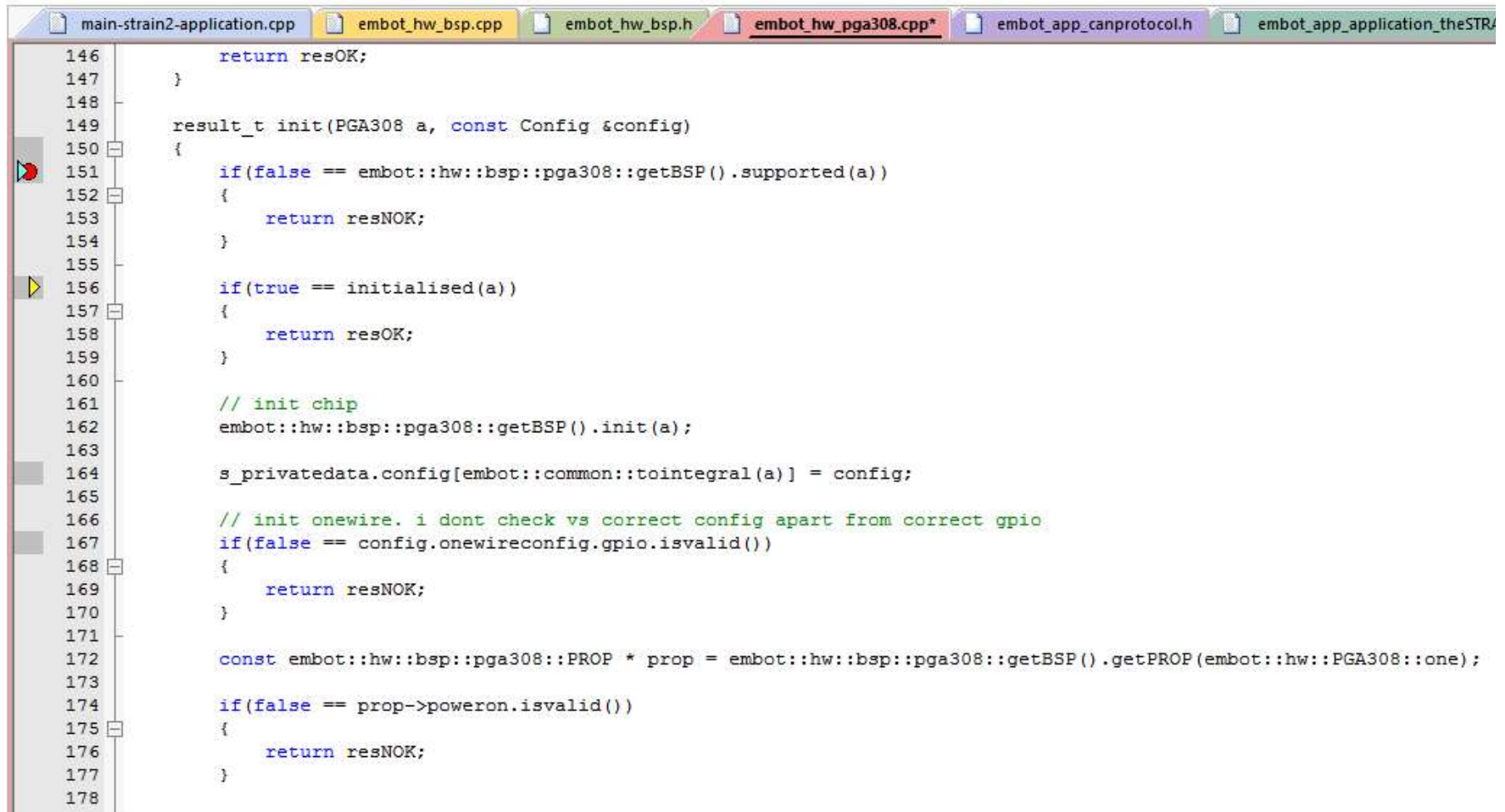
```

578 namespace embot { namespace hw { namespace bsp { namespace pga308 {
579
580     #if defined(STM32HAL_BOARD_STRAIN2)
581
582     constexpr PROP prop { .poweron = {embot::hw::GPIO::PORT::B, embot::hw::GPIO::PIN::fifteen} };
583
584     constexpr BSP thebsp {
585         // maskofsupported
586         mask::pos2mask<uint32_t>(PGA308::one) | mask::pos2mask<uint32_t>(PGA308::two) |
587         mask::pos2mask<uint32_t>(PGA308::three) | mask::pos2mask<uint32_t>(PGA308::four) |
588         mask::pos2mask<uint32_t>(PGA308::five) | mask::pos2mask<uint32_t>(PGA308::six),
589         // properties
590         {{
591             &prop, &prop, &prop, &prop, &prop, &prop
592         }}
593     };
594
595     void BSP::init(embot::hw::PGA308 h) const {}
596
597     #else
598
599     constexpr BSP thebsp { };
600     void BSP::init(embot::hw::PGA308 h) const {}
601
602     #endif
603
604     const BSP& getBSP()
605     {
606         return thebsp;
607     }
608
609 }}} // namespace embot { namespace hw { namespace bsp { namespace pga308 {
610

```

constexpr

Use in embedded software in iCub



The screenshot shows a C++ code editor with several tabs open: `main-strain2-application.cpp`, `embot_hw_bsp.cpp`, `embot_hw_bsp.h`, `embot_hw_pga308.cpp*` (active), `embot_app_canprotocol.h`, and `embot_app_application_theSTR/`. The code in the active tab is as follows:

```
146     return resOK;
147 }
148
149 result_t init(PGA308 a, const Config &config)
150 {
151     if(false == embot::hw::bsp::pga308::getBSP().supported(a))
152     {
153         return resNOK;
154     }
155
156     if(true == initialised(a))
157     {
158         return resOK;
159     }
160
161     // init chip
162     embot::hw::bsp::pga308::getBSP().init(a);
163
164     s_privatedata.config[embot::common::tointegral(a)] = config;
165
166     // init onewire. i dont check vs correct config apart from correct gpio
167     if(false == config.onewireconfig.gpio.isvalid())
168     {
169         return resNOK;
170     }
171
172     const embot::hw::bsp::pga308::PROP * prop = embot::hw::bsp::pga308::getBSP().getPROP(embot::hw::PGA308::one);
173
174     if(false == prop->poweron.isvalid())
175     {
176         return resNOK;
177     }
178 }
```

constexpr



Modern C++

- nullptr,
- auto,
- type aliases,
- initializer list,
- uniform initialization,
- range-based loops,
- constexpr,
- **scoped enumerations,**
- override and final.

Scoped enumerators

Enumerators declared with `enum class` or `enum struct` are called *scoped enumerators*.

```
enum color { Black, White, Red }; // unscoped, C++98
```

```
enum class Color { black, white, red }; // scoped, C++11
```

From Effective Modern C++, Scott Meyers:

[item 10]: Prefer scoped enums to unscoped enums

OK, but why enums at all?

From Effective C++, Scott Meyers:

[item 2]: Prefer consts, enums, and inlines to `#defines`

Or: "prefer the compiler to the preprocessor," because `#define` may be treated as if it's not part of the language per se.

- The enum can be scoped by a namespace, the macro (`#define`) cannot.
- The macro can be redefined, the enum cannot
- The enum can be used as a type for an argument passing.

Why not just unscoped enum?

The unscoped enums suffer two main problems that the scoped enum have solved:

- nothing else in their scope may have the same name, hence they do scope pollution;
- they implicitly convert to integral types and, from there, to floating-point types.

Reasons for scoped enum

They don't do scope pollution.

```
enum color { Black, White, Red }; // they are in same scope as color  
enum class Color { black, white, red }; // they are scoped by Color
```

```
auto White = false; // error! White already declared in this scope  
auto white = false; // fine, no other white in scope  
Color c = white; // error! no enumerator named white in this scope  
Color c = Color::white; // fine  
auto c = Color::white; // also fine  
enum class LED { one, two, three };  
enum class SPI { one, two, three };
```

enum class

□□■□□□□□□□

Reasons for scoped enum

They don't convert to integral types:

```
enum color { Black, White, Red = 2};  
enum class Color { black, white, red = 2 };  
std::vector<std::size_t> primeFactors(std::size_t x);  
color c = Red; // Red but also = 2  
if (c < 14.5) { // compare Color to double (!)  
    auto factors = primeFactors(c); // SIC! prime factors of a color  
}  
Color c = Color::red;  
if (c < 14.5) { // error! can't compare Color and double  
    auto factors = primeFactors(c); // error! can't pass Color to size_t  
}
```

enum class

□□□■□□□□□□□□

Underlying types of enum

Since C++11, we can specify an underlying type for enums so that they occupy a given size:

```
enum color : std::uint8_t { Black, White, Red = 1000 }; // error!  
enum class Color : std::uint8_t { black, white, red = 2 };  
static_assert(sizeof(Color) == 1, "underlying type is not 8 bits");
```

To retrieve the value of the underlying type, use:

```
template<typename E> // // C++14 [Meyers, pag. 73]  
constexpr auto toUType(E enumerator) noexcept {  
    return static_cast<std::underlying_type_t<E>>(enumerator);  
}  
primeFactors(toUType(Color::red)); // Color -> std::uint8_t -> std::size_t
```

enum class

□□□□■□□□□□

Example of scoped enum + constexpr

Example

```
enum class Led : std::uint8_t { red = 0, green = 1, blue = 63, max = blue };

template<typename E, E max>
struct eMap {
    static_assert(toUType(max) < (8*sizeof(unsigned long long)), "size error");
    std::bitset<toUType(max)+1> map;
    constexpr eMap(const std::initializer_list<E> &li) {
        unsigned long long m = 0;
        for(auto &e : li) m |= (static_cast<unsigned long long>(1) << toUType(e));
        map = m;
    }
    constexpr bool available(E e) const { return map[toUType(e)]; }
};
```

enum class

□□□□□■□□□□□□

Example of scoped enum + constexpr

Example (continued)

```
enum class Led : std::uint8_t { red = 0, green = 1, blue = 2, white = 63, max = white };  
constexpr eMap<Led, Led::max> theLEDs {Led::red, Led::blue}; // resolved at compile...
```

```
void init()  
{  
    if(theLEDs.available(Led::green) { // resolved at compile time.  
        dosomething();                // on armclang compiler with 0 optimization ...  
    }                                  // no code is generated.  
  
    if(theLEDs.available(Led::red) {   // resolved at compile time.  
        doit();                       // on armclang compiler with 0 optimization ...  
    }                                  // only code for doit() is generated.  
}
```

enum class

□□□□□□□■□□□□

Modern C++

- nullptr,
- auto,
- type aliases,
- initializer list,
- uniform initialization,
- range-based loops,
- constexpr,
- scoped enumerations,
- **override and final.**

Use override

Use the keyword **override** to prevent mistakes in the design of virtual functions.

I will use Scott Meyers's words from his book *Effective Modern C++* (Item 12: Declare overriding functions **override**):

‘The world of object-oriented programming in C++ revolves around classes, inheritance, and virtual functions. Among the most fundamental ideas in this world is that virtual function implementations in derived classes *override* the implementations of their base class counterparts. It’s disheartening, then, to realize just how easily virtual function overriding can go wrong ‘

Use override

Example (bad design, but OK so far)

```
class Base {  
public:  
    virtual int get() { return b; }           // base class virtual function  
    ...  
};  
class Derived: public Base {  
public:  
    virtual int get() { return d*d; }         // overrides Base::get()  
    ...                                       // ("virtual" is optional here)  
};  
  
Base * p = new Derived;  
p->get();      // OK: called Derived::get() through a Base *
```

override



Use override

Example (bad design, incorrect behaviour)

```
class Base {  
public:  
    virtual int get() const { return b; }    // someone changes signature  
    ...  
};  
class Derived: public Base {                // but forgets the Derived class  
public:  
    virtual int get() { return d*d; }        // it DOES NOT override Base::get()  
    ...                                     // anymore  
};  
  
Base * p = new Derived;  
p->get();    // ERROR: called Base::get() now
```

override



Use override

Example (better design)

```
class Base {  
public:  
    virtual int get() { return b; }           // base class virtual function  
    ...  
};  
class Derived: public Base {  
public:  
    virtual int get() override { return d*d; } // overrides Base::get()  
    ...                                       // ("virtual" is optional here)  
};  
  
Base * p = new Derived;  
p->get();      // OK: called Derived::get() through a Base *
```

override

□□□■□□□□□□□□

Use override

Example (the error is caught at compile time)

```
class Base {  
    public:  
        virtual int get() const { return b; }    // someone changes signature  
        ...  
};  
class Derived: public Base {                    // but forgets the Derived class  
    public:  
        virtual int get() override { return d*d; } // it DOES NOT compile  
        ...  
};  
  
Base * p = new Derived;  
p->get();    // OK: always calls Base::get()
```

override

□□□□□■□□□□□□

There is also final

In C++11 we have the keyword **final** which is used to terminate inheritance.

```
struct Point final { int x, y; };  
struct IperPoint : public Point { int i; };           // does not compile  
class Polygon final : public std::vector<Point> { ... }; // it compiles  
  
class Derived: public Base {                          // but forgets the Derived class  
public:  
    int get() const override final { return d*d; } // cannot further derive get()  
};  
  
class Derived2 : public Derived { // I can further derive the class as long as  
...                               // I don't derive get() any further  
};
```

final



EMPTY

Bla bla bla **typedef** sssss.
dece, **typedef** cercwercver.

- Some etxt.
Soem code // given this map ...

using



