Arguments for C++ in bare-metal embedded

Demonstrated with STM32

Matej Blagšič

June 17, 2022

Prerequisites

Contents apply to non-bare-metal use cases as well

Compiler: `gcc-arm-none-embedded v10.3.1 (2021.10)`

Options: `-std=c++20 -Og -mcpu=cortex-m4 -mfloat-abi=hard -Wall -Wextra -Wpedantic -Wconversion`

Compiler explorer: `https://godbolt.org/z/KP3K5Woxq`

C/C++ bible: `https://en.cppreference.com`

Motivations

C++ can replace C compiler for (almost) all C code:

- Better compile time checks
- Improvements on existing C features
- More tools
- Standard library

Keywords

C: 32
C++: 97 (total in history)
C++ shares all C keywords, some were changed or deprecated

`export`

- until C++11: templates
- until C++20: unused
- since C++20: modules

`auto`

- until C++11: storage specifier
- since C++17: placeholder type

`register`

- until C++17: storage specifier
- since C++17: unused

Right!
C++ versions matter!

Versions

Standardized in c++98
Biggest changes in C++11
Compile time improvements in C++14 and C++17
Standard library improvements in C++20

ENUM

Enum

Each enumeration-constant that appears in the body of an enumeration specifier becomes an integer constant with type int.

```
enum MyEnum
{
    Val1, // 0
    Val2, // 1
    Val3  // 2
};
```

```
enum MyEnum
{
    Val1 = 5,// 5
    Val2,    // 6
    Val3     // 7
};
```

```
enum MyEnum
{
    Val1 = 2,
    Val2 = 115,
    Val3 = -2
};
```

Enum

If you want to omit `enum` from declarations, use `typedef`

```c
enum MyEnum
{
	Val1,
	Val2,
	Val3
};

void foo(enum MyEnum val);
enum MyEnum val = Val2;
```

```c
typedef enum
{
	Val1,
	Val2,
	Val3
} MyEnum;

void foo(MyEnum val);
MyEnum val = Val2;
```

Feature!

C++ drops requirement for `enum` keyword

```
enum MyEnum
{
    Val1,
    Val2,
    Val3
};
```

```
void foo(MyEnum val);
MyEnum val = Val2;
```

Enum

Implicit conversion from `int` to `enum`

```c
int foo(enum MyEnum e);

int val = foo(Val2); // OK
int val = foo(55);   // OK in C, Error in C++
```

Enum

Implicit conversion from `int` to `enum`
No extra compiler settings needed!

```
<source>: In function 'int main()':
<source>:16:20: error: invalid conversion from 'int' to 'MyEnum' [-fpermissive]
16 |      int test = foo(55);
   |                     ^
   |                     |
   |                     int
<source>:9:21: note:   initializing argument 1 of 'int foo(MyEnum)'
9 | int foo(enum MyEnum e)
  |         ~~~~~~~~~~~~~^
Compiler returned: 1
```

Enum

Implicit conversion from `enum` to `int`

```cpp
enum MyEnum
{
        Val1 = 2,
        Val2 = 115,
        Val3 = -2
};

void bar(int);
```

```cpp
bar(Val2); // OK
```

Enum

But what if I don't want my `enum` to implicitly convert to `int`?

```
enum class MyEnum
{
    Val1 = 2,
    Val2 = 115,
    Val3 = -2
};

void bar(int);
```

```
bar(MyEnum::Val2); // Error
```

Now we have to use scope `MyEnum::<member>`

Enum

```
<source>: In function 'int main()':
<source>:17:17: error: cannot convert 'MyEnum' to 'int'
17 |     bar(MyEnum::Val2);
   |         ~~~~~~~~~~~~
   |                |
   |                MyEnum
<source>:10:14: note:   initializing argument 1 of 'void bar(int)'
10 | void bar(int a)
   |          ~~~~~
Compiler returned: 1
```

Enum size

Default enum type is `int`. Size on 32bit arm is 4 bytes.
We can change the size of enum to **any** integer type.

```
enum MyEnum
{
    Val1, // 0
    Val2, // 1
    Val3  // 2
};
```

```
enum MyEnum : int8_t
{
    Val1 = 5,// 5
    Val2,    // 6
    Val3 = -24
};
```

```
enum MyEnum : char
{
    Val1 = 'a',
    Val2 = '4',
    Val3 = 'z'
};
```

Variable initialization/assignment

Yeah ... we have *multiple* ways : |

- ”=” assignment initialization (as in C)
- ”” curly bracket initialization: prevents narrowing

```
int a = -5; // OK
unsigned b = -5; // OK, ooff
unsigned c {-512}; // error: narrowing conversion of
                   // '-512' from 'int' to 'unsigned int'
int d = 4294967295; // OK, oooffff
int e {4294967295}; // same error as before
```

More coming in classes and structs!

Reference&

Reference

Acts as a constant pointer, non-reassignable (`* const`).

Name alias.

Has to be initialized.

Use it as a de-referenced pointer.

Reference

```
int a = 55, b = 66;

int* const p = &a;
*p = 44;
p = &b; // error: assignment of read-only variable 'p'

int& r = a;
r = 33;
r = b;  // copy value of b into r (a = b)
r = &b; // error: invalid conversion from 'int*' to 'int' [-fpermissive]

int* pp = NULL;
pp = &a;
pp = &b;

int& rr; // error: 'rr' declared as reference but not initialized
rr = &a; // error: invalid conversion from 'int*' to 'int' [-fpermissive]
```

Watch out!

Function argument of reference cannot accept a temporary or compile time constant (global const or constexpr).

Function argument constant reference accepts constants and temporaries as well.

```cpp
void foo(int (const) a);// accepts temporary or reference: hard copy
void foo(int& a);        // accepts reference to a valid object: pointer const
void foo(int const& a); // accepts temporary or reference: const pointer const

/* (2) error:
cannot bind non-const lvalue reference of type 'int&' to an rvalue of type 'int'
*/
foo(5);
```