Arguments for C++ in bare-metal embedded

Demonstrated with STM32

Matej Blagšič

June 25, 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`

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

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

```
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`

```
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`?

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

void bar(int);
```

```cpp
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

```cpp
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.

```
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);
```

Overloading

Function overloading

In C, functions are distinguished by their name only.

In C++, functions are distinguished also by their arguments.

This includes different types and/or number of arguments.

Different return types don’t count.

```
void foo(int a);     // _Z3fooi -> foo(int)
void foo(char c);    // _Z3fooc -> foo(char)
void foo(float f);   // _Z3foof -> foo(float)
```

Function overloading

Example: Arduino Serial.print():

```
size_t print(const __FlashStringHelper *);
size_t print(const String &);
size_t print(const char[]);
size_t print(char);
size_t print(unsigned char, int = DEC);
size_t print(int, int = DEC);
size_t print(unsigned int, int = DEC);
size_t print(long, int = DEC);
size_t print(unsigned long, int = DEC);
size_t print(double, int = 2);
size_t print(const Printable&);
```

Default function arguments

Default function arguments

Less used function arguments can be set to default value.

Can only be used in function declaration (headers).

Only trailing arguments can have a default value.

```cpp
// OK
int calculate(int value, int option = DEC);

// error: default argument missing for parameter 3 of 'int transfer(unsigned int,
int transfer(unsigned data, int waitTime = MAX_WAIT_TIME, int* config);
```

Casting 🪄

Casting

Explicit conversion from one type to another.

We can find three types:

- Value casting: float → int
- ! Pointer casting: float* → int*
- ! Const casting: float const* → float *
- Dynamic cast (runtime validity check, disabled: `-fno-rtti`)

Casting

C style casting for **all** types: `(new type) variable/pointer`

C++ provides each cast type its own ”function”, which also provides compile time check:

- ● Value cast: `static_cast<new val type>(var)`
- ! Pointer cast: `reinterpret_cast<new ptr type>(ptr)`
- ! Const cast: `const_cast<same val/ptr type>(val/ptr)`

Casts can be spotted more easily in the source code.

Value cast example

```cpp
float f = 12.421f;
// error is triggered with -Wconversion flag, C and C++ do the same
int a = f; // warning: conversion from 'float' to 'int' may change value [-Wfloat-
int b = static_cast<int>(f);// OK, b = 12

enum class Config
{
    Val1 = 55,
    Val2 = 2155,
    Val3 = -4521
};
void sendConfig(int value);

Config config = Config::Val1;
sendConfig(config); // error: cannot convert 'Config' to 'int'
sendConfig(static_cast<int>(config)); // OK
```

Reinterpret cast example

```cpp
void send_uart(uint8_t const* data, size_t len);
char const* message = "Hello, world!\n"; // or const char*

send_uart(message, strlen(message)); // error: invalid conversion from
                                     // 'const char*' to 'const uint8_t*'
                                     // {aka 'const unsigned char*'} [-fpermissive

// OK, check type size compatability
send_uart(reinterpret_cast<uint8_t const*>(message), strlen(message));
```

Const cast example

```
void print(char* data, size_t len);
char const* message = "Hello, world!\n"; // or const char*

print(message, strlen(message)); // C = warning: passing argument 1 of
                                 // 'print' discards 'const' qualifier
                                 // from pointer target type
                                 // C++ = error: invalid conversion from
                                 //'const char*' to 'char*' [-fpermissive]

print("Hello, world!\n", 14); // warning: ISO C++ forbids converting
                              // a string constant to 'char*' [-Wwrite-strings]

// OK, verify print!
print(const_cast<char*>(message), strlen(message));
```

`using`

`using`

Same as `typedef` but with nicer syntax and streamlined use.

- using-directives for namespaces and using-declarations for namespace members
- using-declarations for class members
- using-enum-declarations for enumerators (since C++20)
- type alias and alias template declaration (since C++11)

using as type alias

```
using u32 = uint32_t;
typedef uint32_t const c32;

void foo()
{
    u32 u = 55;
    c32 cu = 241;
}
```

Compile time and `const`

`const`

Introduced in C++85, included in C89.

If it can be const, then it should be.

When the argument is `const *`, caller knows that the function will NOT modify the contents pointed to by the pointer.

Use it.

I'm looking at you, embedded library providers!

const

```cpp
// Function declaration
HAL_StatusTypeDef HAL_UART_Transmit(
    UART_HandleTypeDef* huart, uint8_t* pData, uint16_t Size, uint32_t Timeout);
// Example
void writeUart(uint8_t const* const msg, size_t len)
{
    HAL_UART_Transmit(&huart2, (uint8_t *)msg, len, UART_TIMEOUT);
}

// Real code
void writeUart(std::string_view view)
{
    // HAL SUCKS
    auto cbegin = reinterpret_cast<std::uint8_t const*>(view.data());
    auto begin = const_cast<std::uint8_t*>(cbegin);
    HAL_UART_Transmit(apiUartHandle, begin, view.length(), UART_TIMEOUT);
}
```