# Arguments for C++ in bare-metal embedded

## Demonstrated with STM32

Matej Blagšič

July 11, 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`

```
1  enum MyEnum
2  {
3      Val1,
4      Val2,
5      Val3
6  };
7
8  void foo(enum MyEnum val);
9  enum MyEnum val = Val2;
```

```
1  typedef enum
2  {
3      Val1,
4      Val2,
5      Val3
6  } MyEnum;
7
8  void foo(MyEnum val);
9  MyEnum val = Val2;
```

# Feature!

C++ drops requirement for `enum` keyword

```
1 enum MyEnum
2 {
3     Val1,
4     Val2,
5     Val3
6 };
```

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

# Enum

Implicit conversion from `int` to `enum`

```
1  int foo(enum MyEnum e);
2
3  int val = foo(Val2); // OK
4  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`

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

void bar(int);
```

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

# Enum

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

```
1  enum class MyEnum
2  {
3      Val1 = 2,
4      Val2 = 115,
5      Val3 = -2
6  };
7
8  void bar(int);
```

```
1  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

```
1 int a = -5; // OK
2 unsigned b = -5; // OK, ooff
3 unsigned c {-512}; // error: narrowing conversion of
4                     // '-512' from 'int' to 'unsigned int'
5 int d = 4294967295; // OK, ooofff
6 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

```
1  int a = 55, b = 66;
2
3  int* const p = &a;
4  *p = 44;
5  p = &b; // error: assignment of read-only variable 'p'
6
7  int& r = a;
8  r = 33;
9  r = b;  // copy value of b into r (a = b)
10 r = &b; // error: invalid conversion from 'int*' to 'int' [-fpermissive]
11
12 int* pp = NULL;
13 pp = &a;
14 pp = &b;
15
16 int& rr; // error: 'rr' declared as reference but not initialized
17 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.

```
1  void foo(int (const) a);// accepts temporary or reference: hard copy
2  void foo(int& a);       // accepts reference to a valid object: pointer const
3  void foo(int const& a); // accepts temporary or reference: const pointer const
4
5  /* (2) error:
6  cannot bind non-const lvalue reference of type 'int&' to an rvalue of type 'int'
7  */
8  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.

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

# Function overloading

Example: Arduino Serial.print():

```
1  size_t print(const __FlashStringHelper *);
2  size_t print(const String &);
3  size_t print(const char[]);
4  size_t print(char);
5  size_t print(unsigned char, int = DEC);
6  size_t print(int, int = DEC);
7  size_t print(unsigned int, int = DEC);
8  size_t print(long, int = DEC);
9  size_t print(unsigned long, int = DEC);
10 size_t print(double, int = 2);
11 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.

```
1  // OK
2  int calculate(int value, int option = DEC);
3
4  // error: default argument missing for parameter 3 of 'int transfer(unsigned int,
5  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 $\rightarrow$ int
- ! Pointer casting: float* $\rightarrow$ int*
- ! Const casting: float const* $\rightarrow$ 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

```
1  float f = 12.421f;
2  // error is triggered with -Wconversion flag, C and C++ do the same
3  int a = f; // warning: conversion from 'float' to 'int' may change value [-Wfloat-
4  int b = static_cast<int>(f);// OK, b = 12
5
6  enum class Config
7  {
8      Val1 = 55,
9      Val2 = 2155,
10     Val3 = -4521
11 };
12 void sendConfig(int value);
13
14 Config config = Config::Val1;
15 sendConfig(config); // error: cannot convert 'Config' to 'int'
16 sendConfig(static_cast<int>(config)); // OK
```

# Reinterpret cast example

```
1  void send_uart(uint8_t const* data, size_t len);
2  char const* message = "Hello, world!\n"; // or const char*
3
4  send_uart(message, strlen(message)); // error: invalid conversion from
5                                       // 'const char*' to 'const uint8_t*'
6                                       // {aka 'const unsigned char*'} [-fpermissive
7
8  // OK, check type size compatability
9  send_uart(reinterpret_cast<uint8_t const*>(message), strlen(message));
```

# Const cast example

```
1  void print(char* data, size_t len);
2  char const* message = "Hello, world!\n"; // or const char*
3
4  print(message, strlen(message)); // C = warning: passing argument 1 of
5                                   // 'print' discards 'const' qualifier
6                                   // from pointer target type
7                                   // C++ = error: invalid conversion from
8                                   //'const char*' to 'char*' [-fpermissive]
9
10 print("Hello, world!\n", 14); // warning: ISO C++ forbids converting
11                              // a string constant to 'char*' [-Wwrite-strings]
12
13 // OK, verify print!
14 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

```
1  using u32 = uint32_t;
2  typedef uint32_t const c32;
3
4  void foo()
5  {
6      u32 u = 55;
7      c32 cu = 241;
8  }
```

# Compile time and `const`

# const

Type qualifier 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

```
1  // Function declaration
2  HAL_StatusTypeDef HAL_UART_Transmit(
3      UART_HandleTypeDef* huart, uint8_t* pData, uint16_t Size, uint32_t Timeout);
4  // Example
5  void writeUart(uint8_t const* const msg, size_t len)
6  {
7      HAL_UART_Transmit(&huart2, (uint8_t *)msg, len, UART_TIMEOUT);
8  }
9
10 // Real code
11 void writeUart(std::string_view view)
12 {
13     // HAL SUCKS
14     auto cbegin = reinterpret_cast<std::uint8_t const*>(view.data());
15     auto begin = const_cast<std::uint8_t*>(cbegin);
16     HAL_UART_Transmit(apiUartHandle, begin, view.length(), UART_TIMEOUT);
17 }
```

# constexpr

Declaration specifier (like `inline`).

It declares that **it is possible** to evaluate the value of the function or variable at compile time.

Such variables and functions can then be used where only **compile time** constant expressions are allowed. This is a requirement for function arguments as well!.

Public functions are required to have their implementation **in headers!** (no declaration-definition separation).

# constexpr

Object declaration or `non-static` member function **implies** `const`.

Used in a function or `static` data member **implies** `inline`.

`constexpr` variable has to be of type `Literal Type` (Scalar, reference, array of literal type ...)

Functions have to be pure and don't have side effects.

Read up on restrictions:
`https://en.cppreference.com/w/cpp/language/constexpr`

# Examples of constexpr

```
1  // Inlined even with -Og (holds true even for much larger functions)
2  constexpr int pow2(int a)
3  {
4      return a * a;
5  }
6
7  // non constexpr function
8  int pow2_non(int a);
9
10 int val1 = pow2(5); // may be evaluated at compile time
11 int val2 = pow2_non(5); // pow_non is always generated, evaluated at run time
12 constexpr int val3 = pow2(5); // evaluated at compile time, also can be optimized
13 constexpr int val4 = pow2(pow2_non(5)); // evaluated at compile time, also can be
14 constexpr int val5 = pow2_non(5); // error - pow2_non is not constexpr
15
16 // Compile time checks
17 static_assert(val3 > 25 && val3 < 30, "val3 not in bounds!");
18 static_assert(pow2(5) == 25, "pow2 error!");
```

# Examples of constexpr

## Constants

```
1  struct MyStruct;
2
3  constexpr bool defaultStruct(MyStruct const& s)
4  {
5      constexpr MyStruct defaultStruct = MyStruct{};
6      if (s == defaultStruct)
7      {
8          return true;
9      }
10     return false;
11 }
12
13 constexpr bool isStructDefault = foo(MyStruct{}); // calculated at compile time
```

`defaultStruct` will not occupy memory, unless declared with static

`bool`

# boolean

C++ has built-in **type** `bool`.
In C you bool as a **macro** with `stdbool.h`.

It can hold a value of `true` or `false`.

Size is left to compiler implementation, but is usually 1**B**.

It is implicitly converted to and from `int`, but can be prevented with {brace initialization and assignment}.

`bool` is resulting type of boolean comparison between objects:
`&&`, `||`, `<`, `<=`, `>`, `>=`, `==`, `!=`

# >>operators

# operator

Operator performs an operation on variable(s).

Built-in types have defined operators for arithmetics and comparisons (just like in C).

Math: `+`, `++`, `+=`, `-`, `--`, `-=`, `/`, `/=`, `*`, `*=`, `%`, `%=`
Bitwise math: `&`, `&=`, `|`, `|=`, `^`, `^=`, `~`, `<<`, `>>`
Boolean comparison: `&&`, `||`, `<`, `<=`, `>`, `>=`, `==`, `!`, `!=`
Pointer arrow operator: `->`, `->*`
Call and bracket operator: `( )`, `[ ]`

# Feature!

C++ treats operators as **functions**.

operator overloading is used to accommodate same operator for multiple (number of) types.

This means we can add our own implementations!

Some additional operators: `>>=`, `<<=`, `<=>`, `","`

Read more:
`https://en.cppreference.com/w/cpp/language/operators`

# operators

Some keywords are actually ”special” operators!
Some can be overloaded.

- `static_cast`
- `reinterpret_cast`
- `const_cast`
- `sizeof`
- `new`
- `delete`

More on operators later!

# Operator for custom type

```cpp
struct Complex { double re = 0.0, im = 0.0; }; // Complex a{}, b{1.0, 2.0};

Complex operator+(Complex const& lhs, Complex const& rhs)
{
    return Complex{lhs.re + rhs.re, lhs.im + rhs.im};
} // Complex c = a + b; // c = {1.0, 2.0}

Complex operator*(Complex const& lhs, double d)
{
    return Complex{lhs.re * d, lhs.im * d};
} // Complex d = b * 55.0; // d = {55.0, 110.0}

void operator/=(Complex& lhs, double d)
{
    lhs.re /= d;
    lhs.im /= d;
} // b /= 2.0; // b = {0.5, 1.0}
```