# C++ Keywords

## Sections of a binary

### `.text`

Assembly instructions are placed within this section. When loaded onto a board, this will section will be placed into the flash memory (ROM) of the board. The binary file (.bin) that you load onto you board includes all of this information in it. The more code you write, the bigger the binary size gets.

### `.data`

All global initialized variables are placed in this section. When the binary is created, in order to know what those global variables were at compile time, they are put into the binary which takes space on the ROM. At runtime, when the embedded system turns on, it moves the .data contents from the ROM to RAM so it can be used and modified by the application.

### `.bss`

Section contains information about all uninitialized global variables. This will take up a small section in ROM, in that it only includes the start position in RAM and its length. The embedded platform will write zeros to to the start of that RAM location, extending its whole length. The `.bss` section must be cleared before using any **newlib** (stdclib and stdc++lib) libraries:

Reference: https://www.embecosm.com/appnotes/ean9/ean9-howto-newlib-1.0.html#id2717944

# Const

Out of all of the discussed keywords, **const** is probably the most commonly known keyword because of its simplicity as well as its ubiquity across many languages. But does it REALLY mean for a variable, class, or structure to be **const**?

Resource: https://en.cppreference.com/w/cpp/language/cv

## As a Global Variable on an Embedded Platform

Take the following code:

```
const uint32_t kConstantVariable = 5;

int main(void)

{

  // ...

  return 0;}
```

### What effect does the const in front of the variable type change the variable?

1. Does not allow the variable to be modified. Will throw a compiler error.
2. **Will be place the variable in ROM (or .text section).**
3. Removing the const will place

## As a Local Variable on an Embedded Platform

1. Compiler will not allow modification of the variable.
2. Will be placed in the **STACK** as per a typical function call.

## Cheating the system

```
const uint32_t kConstVariable = 5;

void AttemptToModifyConst()

{

  uint32_t * const_pointer = const_cast<uint32_t*>(&kConstVariable);

  *const_pointer = 10;  // Should cause system fault. Do not try this.

  const uint32_t kLocalConstVariable = 15;


  uint32_t * local_const_pointer = const_cast<uint32_t*>(&kLocalConstVariable);

  *local_const_pointer = 10;  // Should NOT cause system fault, but defeats the purpose of using const

}
```

If you cast away the const of a variable, you can attempt to change it. If the variable is global and placed in ROM, this will attempt a write access to the ROM which will cause a system fault. Doing so to local variables, which exists on the STACK which is within ram does not cause any faults, because the memory was always mutable. In the case of local variables, const is a means to keep you from compiling code that changes a constant.

## Benefits of using Const in Embedded

If you evaluate many **MCUs**, you will see that most of them have a decent amount of flash but small amounts of RAM. If you have information that does not need to be modified at run time, you can shift the information into

the ROM by placing it in global space (or make it static, see static section). This is why you should make you character strings, lookup tables, bitmaps and anything else const.

```
// Examples of good use cases for
const char kIpAddress = "192.168.1.5";
const char kUrl = "http://example.com/index.html";
const uint32_t kBitMasks[] = { 0xF0F0F0F, 0x55555555, 0xAAAAAAAA };
const uint8_t kMaximumRetries = 10;
```

# Volatile

Every access to volatile variable will be treated as a visible side-effect. This means that:

1. The compiler CANNOT do optimizations such as out-of-order instruction reordering, link time garbage collection of the object or as-if transformations.
2. Access to object WILL bypass CPU cache and generate a bus cycle.

```
volatile uint32_t * pin_address = &LPC_GPIO0->PIN;
// Pin address will be loaded from the system bus and written back to the system bus.
// No caching will take place.*pin_address |= (1 << 5);
```

## Benefits of using Volatile in Embedded

- Required in order to insure that access of registers is not optimized out.
- Prevents the register information from being access through the cache. To change a register, you need to write to it, but if you make the changes to cache memory, the actual hardware isn't read or written to.

# Inline

Many people make the mistake in C++ in thinking that the **inline** keyword means that the contents of a function call will be inlined at the call site. This is not correct. In order to make this happen you must use the `always_inline` compiler attribute.

## How define a function that will be inlined at its call site

```
// Modern C++17 and above attribute
[[gnu::always_inline]]
```

```
void CallsiteInlinedFunction(int a, int b)
{
 return a + b;
}
// Older version of GCC will require this
// Must seperate the attributes from the declaration
__attribute__((always_inline))
void CallsiteInlinedFunctionOld(int a, int b);
void CallsiteInlinedFunctionOld(int a, int b)
{
 return a + b;}
```

## Inline functions and variables within header files

Inline functions and variables can be defined within a header file without the need to define them in a .cpp file. The will keep linker errors from appearing

> TODO: Add more detail as the above lacks it.

## For member variables in class/structs

Static variables can be defined within a class if their declaration is preceded by the `inline` keyword.

# Static

The **static** keyword has a load of different meanings depending on where it is used.

?

## Static global variable or function

```
// Objects below are not visible outside of the .cpp file.
// Techincally works in .h/.hpp files but it defeats the purpose of putting it in there.
static uint8_t kHiddenBuffer[256];
static void FunctionPrivateToThisFile()
{
  // ...}
```

Stay away from using static this way in C++. If you would like to make a variable, object, type, etc private to a

file you can use an anonymous namespace.

```
namespace
{
// Objects below are not visible outside of the .cpp file.
// Techincally works in .h/.hpp files but it defeats the purpose of putting it in there.
uint8_t kHiddenBuffer[256];
void FunctionPrivateToThisFile()
{
  // ...
}
}  // namespace
```

## Static local function/method variable

Static variables within a function are actually apart of the `.data` section and not the stack. The also retain values across calls. This variable can be considered the state of the function.

```
uint32_t FunctionWithInternalStateVariable()
{
  static uint32_t call_count = 0;
  return ++call_count;}
```

## Static class/struct member variable

```
class ClassWithStaticVariable
{
 public:
  ClassWithStaticVariable()
  {
    // NOTE: that this is NOT thread safe!
    id = next_id++;
  }
  // ...
 private:
  // This variable is common and accessable by all objects of this class
```

```
  // So if one class alters it, each class will see that change.
  inline static uint32_t next_id = 0;
  uint32_t id;};
```

# Enum Class

You may be familiar with enumerations in C and C++. What enumeration class does is make enumerations strong types.

```
// A clever way to force the user to your enumeration
// vs them potentially putting in an invalid value.
enum class TransferSpeed : uint32_t
{
  kHigh = 0b011,
  kFast = 0b010,
  kLow = 0b001,
  kDisabled = 0b111,
};
void SetTransferSpeed(TransferSpeed speed)
{
  // Because speed is not a integer type, you need to cast it into that type.
  *transfer_speed_register = static_cast<uint32_t>(speed);
}
// Usage ...
SetTransferSpeed(TransferSpeed::kHigh);  // OK
SetTransferSpeed(TransferSpeed::kFast);  // OK
SetTransferSpeed(0b00);  // Compiler Error!
```

# Constexpr

Variables and function declared with the `constexpr` keyword exist only at compile time.

```
// Should return a mask like so:
//   AlternatingPatternMask(1) => 0b...0101'0101'0101'0101
//   AlternatingPatternMask(2) => 0b...1011'0110'1101'1011
//   AlternatingPatternMask(3) => 0b...0111'0111'0111'0111
//   etc ...
constexpr uint32_t AlternatingPatternMask(uint8_t number_of_ones_in_sequence)
{
  uint32_t result = 0;
  for (int i = 0; i < sizeof(uint16_t)*8; i++)
  {
    uint32_t set_this_bit = ((i % number_of_ones_in_sequence) == 0) ? 0 : 1;
    result |= set_this_bit << i;
  }
  return result;
}
// ...
// This value of this global variable is figured out at compile
// time and not at runtime.
// NOTE: this depends on the situation in which it is used.
uint32_t three_ones_in_sequence_mask = AlternatingPatternMask(3);
```

# Using

> TODO: Fill this out later

?

Revision #4
Created 5 years ago by Khalil Estell
Updated 5 years ago by Khalil Estell