Good practices in modern C++ for the purposes of MRS

This guide will attempt to summarize good practices to use and bad practices to avoid when writing C++ code in the context of ROS, robotics and research. The guide is mainly targeted at people who are coming to C++ from C or who are still using old-style C++ (e.g. raw pointers). You don't have to read every last word as some of the sections deal with quite specific topics (e.g. multithreading). I *do* recommend at least skimming through the whole page to check for anything that's new or that might be useful for you. At the minimum, check out the sections related to smart pointers and the general tips.

A list of the main tackled topics is:

- Some useful C++ libraries.
- Avoid raw pointers, `new`, `malloc`, etc. like the devil.
- Take function parameters by constant referece (or constant copy in case of primitive types).
- Use the native multithreading and thread synchronization tools.
- ROS-related coding practices.
- Other tips and remarks.
- Further reading.

If you spot any errors, don't understand something or have ideas for improevments, feel free to contact me at `matous.vrba (at) fel.cvut.cz`.

Useful libraries

Before implementing basically anything, **first check that a suitable implementation doesn't already exist** (this goes for scientific research as well - do your research before you start reinventing the wheel 😀)! Typically, using an already existing and optimized implementation is not only easier and faster than implementing your own, but also the code will be faster and bug-free. A list of useful C++ libraries that you might need with links to their documentation pages follows:

- **The standard C++ library:** Implements many useful algorithms, tools and utilities. Part of the C++ standard. Learn it and learn to use it!
  - https://en.cppreference.com/w/
- **roscpp:** The main ROS C++ API.
  - https://docs.ros.org/en/noetic/api/roscpp/html/
- **tf2_ros:** The ROS tf2 library API, implementing coordinate transformations and related stuff.
  - https://docs.ros.org/en/noetic/api/tf2_ros/html/c++/
- **Eigen:** Linear algebra, basic geometry and other matrix-related stuff (ROS has compatible interfaces).
  - https://eigen.tuxfamily.org/dox/index.html
- **OpenCV:** Computer vision and image processing (ROS has compatible interfaces).
  - https://docs.opencv.org/4.2.0/
- **PCL:** Point cloud processing (ROS has compatible interfaces).
  - https://pointclouds.org/documentation/
- **Boost:** General C++ library implementing *many* tools, algorithms and utilities (used internaly in ROS).
  - https://www.boost.org/doc/libs/1_71_0/
- `mrs_lib`: Our own MRS library implementing some algorithms (e.g. various Kalman filters), ROS wrappers (e.g. for parameter loading) and other utilities (e.g. a 3D geometry library).
  - https://ctu-mrs.github.io/mrs_lib/

Most of these libraries already come pre-installed with ROS or our UAV system and we use them, so we can help you in case you encounter any problems (don't be afraid to ask).

Dynamic memory management

In C, raw pointers are a crucial tool for many tasks, which include management of dynamic memory, passing around large data structures and data ownership management. Many of these problems may be tackled using more focused tools in C++, significantly simplifying and clarifying the code and making it less error-prone.

References

In many cases, pointers may be avoided altogether in C++ by using references, especially when passing function parameters (see the next section). However, references are useful in other cases

as well. Consider the following scenario, where you want to transform the fifth element of the container `cont`:

```
cont.at(102) = 10 + 3*cont.at(102) + 0.1*cont.at(102)*cont.at(102);
```

Here, the `at()` method of the `cont` object is called four times, which may be quite costly e.g. in the case of a linked-list, and is error-prone (a single typo in the index number can break this code). A cleaner version may be obtained using references:

```
auto& cur_el = cont.at(102);
cur_el = 10 + 3*cur_el + 0.1*cur_el*cur_el;
```

The reference also avoids unnecessary copying of the container element (which is important if it's a large data structure). This approach is often employed in range-based for loops, so it's good to understand it well.

Note: Watch out for dangling references (references to variables which went out of scope) These typically happen when returning a reference to a local variable from a function, which is a no-no.

Smart pointers

For dynamic memory and data ownership management in modern C++ is done using the so-called *smart pointers* (and yes, they are pretty clever). **You should not use the keywords `new` nor `delete` (and definitely not `malloc()` nor `free()`) in almost any case in modern C++! This functionality is replaced by smart pointers, which are safer, more user-friendly and less error-prone.** There are three types of smart pointers:

- The unique pointer is the most basic smart pointer. It is the sole and only owner of the memory it points to (hence the name). The memory is allocated on construction of the `std::unique_ptr` object and freed at its destruction, so the user doesn't have to worry about calling `new` nor `delete` (and definitely not `malloc()` nor `free()`). The unique pointer is the most safe and efficient one, but it's quite restrictive as it cannot be copied or copy-constructed (that would break the unique ownership of its data).
- The shared pointer is *the most common smart pointer you will encounter*. It works similarly as the unique pointer, but has a counter which is incremented at each copying of the pointer and decremented at each destructor call. This counter counts how many pointers point to the respective memory and when it reaches zero (the last `std::shared_ptr` pointing to this memory is destroyed), the memory is freed. The thread-safe incrementation/decrementation of the counter makes the `std::shared_ptr` a bit less efficient than the `std::unique_ptr`, but it can be

freely copied, destroyed, shared between threads etc. (although synchronization is still required for accessing the data being pointed to!).

- The weak pointer is a non-owning pointer to a memory. It neither allocates nor destroys memory and before the pointed-to memory may be used, it has to be *locked*, which returns a new `std::shared_ptr`. Otherwise, the memory it points to is owned and allocated/deallocated by a different shared pointer (see the example at cppreference).

For most of our applications, you will utilize only the shared pointers. Note that ROS uses the Boost implementation (`boost::shared_ptr`) instead of the standard library implementation for legacy reasons (this is fixed in ROS2). Luckily, the Boost shared pointer works identically to the standard library (although they cannot be converted to each other's type).

When instantiating a `std::shared_ptr`, use the `std::make_shared<T>()` function, which takes the object `T`'s constructor parameters as arguments - e.g.:

```
class Bar
{
public:
  Bar(const int number, const std::string& text)
   : m_number(number), m_text(text)
  {};
private:
   int m_number;
   std::string m_text;
};
std::shared_ptr<Bar> obj_ptr = std::make_shared<Bar>(666, "foo");
```

Similarly for `std::unique_ptr` and `std::make_unique`.

Function parameters

The rules of thumb when defining function parameters is:

1 If you're taking a primitive type as a parameter (e.g. `int`, `float`, `bool` etc.), use a constant copy:

```
bool foo(const int a, const float b);
```

2 If you're taking a class/struct, use a constant reference:

```
class Bar, Baz;
Bar foo(const int a, const Baz& b);
```

3 If you need to return multiple variables, there are several possibilities:

```
std::tuple<bool, float> foo(const int a)
{
  if (a > 0)
    return {true, 0.1*a};
  else
    return {false, a};
}

// preffered way since it's clearer what is input and what output
// and all can be const, avoiding accidental modification
const int a = 5;
const auto [c, b] = foo(a);
```

or

```
bool foo(const int a, float& ret_b)
{
  if (a > 0)
  {
    ret_b = 0.1*a;
    return true;
  }
  else
  {
    return false;
  }
}

// less elegant and less clear way, but valid
const int a = 5;
float b;
const bool c = foo(a, b);
```

4 If you want to modify a parameter passed to a function (e.g. use the function to update an object's value), use a reference, but make this clear (ideally by naming of the function and the parameters):

```
void append_squared(std::vector<float>& to, const float new_val)
{
  to.push_back(new_val*new_val);
}
```

Thread synchronization

There are three types of synchronization mechanisms for multi-threading in C++:

1 **Atomic variable:** If you have a single primitive-type variable which you want to access and modify from multiple threads in a thread-safe manner (e.g. some counter or a flag that a thread is running), use `std::atomic<T>`. See also the `mrs_lib::AtomicScopeFlag` helper class for automatic atomical setting and unsetting a flag (boolean variable).

2 **Mutex:** For cases where multiple threads modify/read a common resource (e.g. an `std::vector` or other data), use `std::mutex` and `std::lock_guard` to synchronize the access and prevent data races. See an example on cppreference.com.

3 **Condition variable:** The `std::condition_variable` is useful in cases when a thread (or multiple threads) has to wait for another thread to generate a resource to be consumed by the waiting thread (threads). In the context of ROS, this may be waiting until a message on some topic arrives for your thread to process (this is implemented in the `mrs_lib::SubscribeHandler`'s `waitForNew()` method). See an example on cppreference.com.

Other remarks regarding multi-threading in C++:

- In general, do **not** use `volatile` (unless working with a microcontroller where you really need it or other platform-specific cases, which generally don't concern us). Note that `volatile` does *NOT* ensure thread safety - for these cases, use `std::atomic<T>` (which also much better communicates your intention to the compiler as well as to any potential readers of the code)!
- The condition variable may sound very similar to mutex, but actually isn't. A mutex is intended to keep two threads from using the same resource (i.e. a data race), whereas a condition variable is used to suspend a thread until a resource becomes available. You may think of it this way:

- By default, a *mutex* is unlocked and any thread can lock it. Any other thread then has to wait for the first one to release it again.
- By default, a *condition variable* is unavailable and no thread can use it. Any thread may wait for the condition variable (and atomically lock it when it becomes available). Any thread may notify a single or all threads waiting on the condition variable that it has become available (thus waking them).

ROS-related coding practices

To get started with ROS, check out the official *roscpp* tutorials and our example ROS packages:

- example_ros_uav - general ROS package, demonstrating some basic concepts.
- example_ros_vision - a computer vision ROS package, demonstrating some basic CV stuff. Go through the code of these examples and try to understand it (you can skip the vision package if you won't be working on CV). **Read their README** - especially the Coding style and Coding practices parts, which contain useful information related to using C++ in the context of ROS and the *roscpp* API.

Also be sure to check out the available ROS helpers in our `mrs_lib` C++ library. Namely, these helpers are good to use to improve code clarity and robustness:

- `ParamLoader`: Loading of parameters from the `rosparam` server, checking of parameters being loaded correctly, automatic printing of the loaded values.
- `SubscribeHandler`: Subscription to ROS topics with automatic printing when no messages were received for a specified timeout. Threadsafe blocking waiting (with timeout) for new messages or callbacks or flag-checking for new messages.
- `Transformer`: ROS transformations wrapper for easier transformation lookup, one-time or repeated transformation of various types including handling of the special GPS UTM frame (specification of points in lat/lon coordinates).
- `ScopeTimer`: Simple scope-based profiling tool (like `tic-toc` and similar) for timing of duration of various processes.

Other tips and remarks

- Turn on `-Wall` and write your code to emit no warnings. The warnings are there to tell you about potential code smell (not to annoy you), so do not ignore them.
- Do not use the `NULL` macro, use the `nullptr` pointer literal. `NULL` may be defined to be the integer literal `0` according to the standard, which makes some unexpected implicit

conversions possible when using `NULL`. `nullptr` can never be implicitly converted to `int`, making it safer.

- Use `const` whenever possible. This way you will avoid accidentally modifying variables which are not supposed to be modified and enable the compiler to better optimize.
- Use `std::numeric_limits` instead of the `INT_MAX`, `DBL_MAX`, etc. macros. In general, you should avoid macros whenever possible. Watch out for `std::numeric_limits<T>::min` vs. `std::numeric_limits<T>::lowest`! The `::min` function returns the *smallest positive value* (not the lowest - therefore negative - value, which is returned by `::lowest`) for floating types (this behavior is the same for the macros such as `FLT_MIN` by the way).
- Shorten long typenames that you use repeatedly with the `using` aliasing to improve code readability.
- Learn and use the `gdb` debugger (see our short introduction).
- Learn to use the C++ reference documentation and consult it whenever you use a new thing from the standard library.
- Use documentation in general. Do not guess what stuff does or how it's called. Find the documentation of whatever library you're working with, bookmark it, read it and use it. Also learn to use the search tool (the input field on the top-right) in Doxygen-generated pages.

Automatic type deduction

C++ is a typed language, meaning that the type of any variable in the program has to be known at compilation. This has many advantages and enables very powerful compile-time sanitization and error-checking as well as performance improvements and optimizations, but the language can become extremely verbose even to the point of reduced readability (this is especially the case when using templates and the standard library). Enter the `auto` keyword.

`auto` loosely translates to "dear Mr. compiler, please substitute this word with the appropriate deduced type during compilation". For example, instead of writing the whole type of a `std::vector` iterator such as

```
const std::vector<float> cont = init_container();
for (std::vector<float>::const_iterator it = std::cbegin(cont); it != std::cend(cont); it++)
{
  // do stuff with it
}
```

you can simplify this code without loosing expressivity to

```
const std::vector<float> cont = init_container();
for (auto it = std::cbegin(cont); it != std::cend(cont); it++)
{
  // do stuff with it
}
```

Note that `auto` should not be overused at the cost of code readability.

A rule of thumb: If the typename can be automatically deduced by the compiler and it is long, too verbose and the type is clear from the context or variable naming, substitute it with `auto`.

Range-based loops

Since C++11, the range-based `for` loops syntactic sugar is available. Specifically, the following syntax is legal for any container that implements the `begin()` and `end()` methods according to the standard (e.g. `std::vector`, `std::forward_list`, `pcl::PointCloud`, `cv::Mat` etc.):

```
for (const auto& element : container)
{
  // do stuff with element
  if (element > 0)
    sum += element;
}
```

If you want to modify the elements, just drop the `const` keyword:

```
for (auto& element : container)
{
  // do stuff with element
  if (element > 0)
    element += offset;
}
```

I recommend using this syntax whenever applicable as it's more expressive and less verbose and error-prone than classic iteration or C++ iterator-based iteration.

A rule of thumb:

- If you do not need to know the iterator inside the `for` loop and only need to access/modify the elements, use a range-based `for` loop:

```
for (const auto& element : container)
```

- If you need to use the iterator inside the loop body, use an iterator-based `for` loop:

```
for (size_t it = 0; it < container.size(); it++)
```

or

```
// if you need to modify the elements, use std::begin() and std::end() instead
for (auto it = std::cbegin(container); it != std::cend(container); it++)
```

Further reading

- The C++ Best Practices site from Jason Turner are a good general overview of C++ programming.