

# ECE 3574: Embedded Programming in C++

*Changwoo Min*

# Embedded Programming in C++

- Today we are going to look at what subset of C++ is suitable for embedded programming.
  - What is different about embedded systems programming?
  - C or C++?
  - Memory Management

# Some examples of embedded systems

- Cars
- Cell Phones
- Airplanes
- Space Craft
- Medical Equipment (e.g MRI scanners)
- Elevators
- Appliances (e.g. Washer/Dryer)
- Automated Assembly Systems
- Power Plants

# Some examples of embedded systems

- Typically there are several separate computers within a system, each with one or more CPUs that communicate via buses or private networks.

# What is different about embedded programming?

- reliability is critical: failures can cost lives and millions of dollars
- resource limited: memory, processor cycles, power
- real-time response requirements: control systems have a hard update rate
- long-term operation: e.g satellites
- hands-on maintenance and software updates are difficult or not feasible

# Programs in such systems need to be

- Correct: correct results, at the right time, in the right order with limited resources.
- Fault Tolerant: even including hardware failures (e.g. bad memory blocks)
- Have No Downtime: no-memory leaks!
- Predictable: hard time-constraints restrict the algorithms that can be used.
- Concurrent: programs work together across buses to achieve a result

# Some general design guidelines

- Don't leak resources: memory, locks, file descriptors
- Replicate: voting systems and hot-swap spares
- Self-check: know when your code is misbehaving
- Define a chain of responsibility: failure notifications should propagate
- Monitor: have a separate monitoring system
- And test, Test, TEST, TEST!, \*\*TEST!\*
- Emulation and virtualization can be helpful.

# Example emulation setup

- QEMU
- Board: chipKIT Max32 with PIC32MX7 and bare-metal bootloader
- Compiler: gcc built for cross-compiling to mips (e.g. Microchip xc-g++)
- The standard library is not linked, instead a custom linker script is used to place the code section, entry point (main), and interrupt vector table



# C++ is predictable with three important exceptions:

- heap-allocation (new/delete) – placement new is ok
- exceptions
- dynamic\_cast (RTTI)
- These language features and any libraries that use them are banned or severely restricted in most embedded systems code.
- Note this eliminates `std::string` and the standard containers library with the default allocator, but not all of the standard library.

# To C or not to C

- So, how much of C++ does that leave? Quite a bit actually.
  - front-end syntax (e.g. default parameter values)
  - Classes
  - RAI
  - Templates
  - can still use placement new
- Unfortunately, it is common to see embedded platforms that do not support C++.

# Memory management

- Recall there are three ways to allocate memory in C++
  - static: linker allocates, persists as long a program runs. No concerns.
  - stack: automatically allocated, function call depth (particularly recursive functions) is a concern
  - dynamic (heap/pool): manually allocated by new, deallocated by delete, this has problem with predictability and pool fragmentation
- Dynamic memory allocation is usually either banned, or limited to startup code only. Deallocation is generally banned.

# Example: using static allocation

- Implement a templated stack using statically allocated memory.
- See `static_alloc.cpp`

# Example: using automatic allocation

- Implement a templated stack using automatically allocated memory.
- See `auto_alloc.cpp`

# What is bad about dynamic memory allocation?

- Dynamic memory is allocated in a section of memory called the heap, a pool of memory allocated to the process (if an OS is used) or available to the compiler (no OS).
  - Allocation is not really a problem as long as it does not exceed available space.
  - Deallocation however (delete) creates holes in the memory. This can exhaust memory even without memory leaks.
- See `process_test.cpp`.

# Alternatives to dynamic memory allocation

- The real issue with dynamic allocation is different sized objects.
  - We could use a compacting garbage collector, but when should it run, how to make predictable?

# Alternatives to dynamic memory allocation

- We can get predictability if we use a different data structure:
- stacks: pushes/pops happen at one end so object size does not matter.  
This mimics the way automatic allocation works.
- pool: allocate fixed size blocks that can be recycled without causing holes. This is an allocator for single sized objects.
- This is the basic idea behind custom allocators. We use specialized algorithms for allocation.



# Simple stack example

- Consider a keypad attached to an MCU with a loop polling key presses.
  - Each time a key is pressed it is pushed onto the stack.
  - If a del/back key is pressed a value is pop'd from the stack.
- See keypad\_example.cpp

# Pool example

- Lets look at a simple implementation of a pool based allocator.
- See `ring_buffer.h`, `pool.h`, and `pool_ex.cpp`.

# Next Actions and Reminders

- Reading on FreeRTOS