

Audio Programming Project

CS-E5690



Aalto University
School of Electrical
Engineering

Course Staff

- Responsible Teacher: Lauri Savioja, lauri.savioja@aalto.fi
- Lecturers: Joao Rossi Filho, rossi@complexroot.com
Eero Pekka Damskagg, eero-pekka@complexroot.com
- Course Assistant: Valtteri Kallinen, valtteri.kallinen@aalto.fi



Participants - Pre-Questionnaire

- What is your digital audio background?
 - Ranging from basic DSP knowledge to own effect development
- What is your experience in software development?
 - Ranging from none to professional level, most have strong background in Python/Matlab
- Motivation?

Course Goals



- Learn best practices for modern real-time audio programming from industry experts
- Hands on development of your own real-time audio plugins with C++ and JUCE
- Have fun and network with others interested in audio programming!

Grading

- 4 small home work exercises (25 % grade)
 - No Submission 0 points
 - Submitted but doesn't work correctly 1 point
 - Submitted and works correctly 2 points
 - Any meaningful extra work will be rewarded
 - Intended for pair-wise working (but individual submissions are accepted as well)
- Project work (75 % grade)
 - Code
 - Report
 - Demo

Project Work

- Design and implement a real-time audio effect plugin
- We will provide a list of broad topics for inspiration
- Be as creative as you want!
- Groups of 3 persons





- We will use slack for support during the course and especially for support during the project work phase
- You will receive invite to join the course Slack-channel asap!

Demo Session

- Friday June 6th, at 13:00, Otakaari 5, 3rd floor, Karjalainen
- Each group gives 10 min presentation (+ 5 min Q/A)
 - Describe your plugin
 - Demonstrate the GUI
 - Real-time audio demo ! (audio file / live input)
- Open event (Acoustics lab staff invited)



Course Programme 2025

5.5. – 6.6.

Week	Time	Mon	Tue	Wed	Thu	Fri
19		Intro to C++				
20	9–12	(Real-time software concepts) Intro to JUCE	Homework: Ring modulator	Model Solutions: Ring modulator Lecture: IIR & FIR Filter	Homework: Delay Effect	Model Solutions: Delay Lecture: Oscillators
	13–16	More Juce Workflow with VS Code Debugging		Lecture: Gain Ramping Modulated Delay Line Lecture: Flanger Effect		Event handling Midi - protocol Synthesizer - VCO, VCA, Filter, Envelope

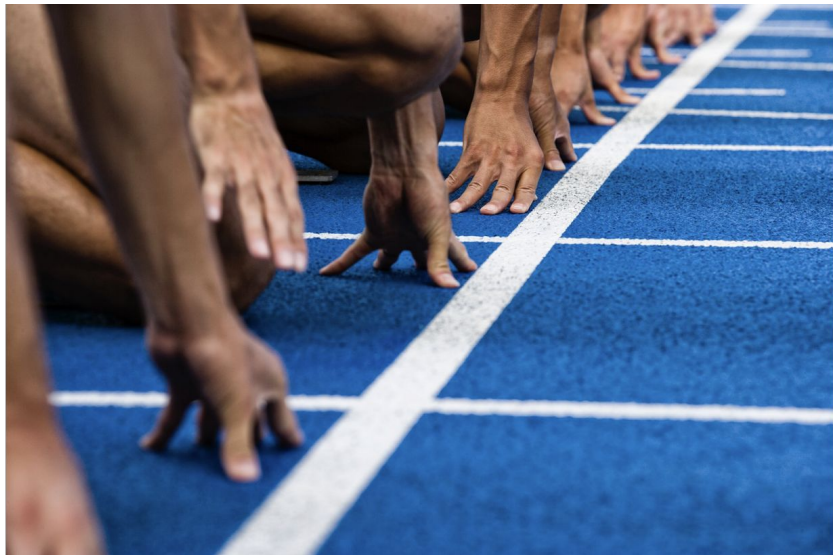
Course Programme Weeks 21–23

21	9–12	GUI elements Widget customization	Homework: Subtractive synthesizer, extra: polyphony	Neural Networks in Real-time Audio	Homework: Inference example	Model solution: inference example Project Development
	13–16	GUI elements Volume meter Project Kickstart		Reverb?		Project Development
22	-	Project Work	Project Work	Project Work	Project Work	Project Work
23	-	Project Work	Project Work	Project Work	Project Work	Demo Day

Contact Sessions

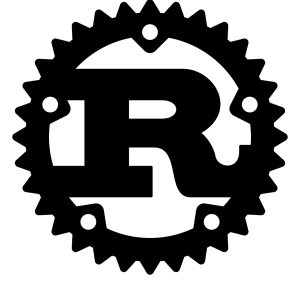
Day 0

- Introduction to C++ Development
- Course environment for assignments
- At the end of the day you should:
 - know why real-time plugins are written in C++
 - be able to read and understand, and modify simple C++ programs
 - have a working environment in which you can compile and run plugins (both standalone and as plugins)



Compiled and interpreted languages

Compiled vs. Interpreted Programming Languages



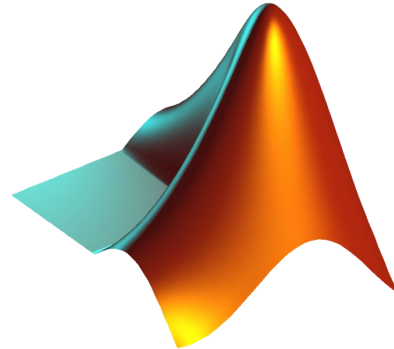
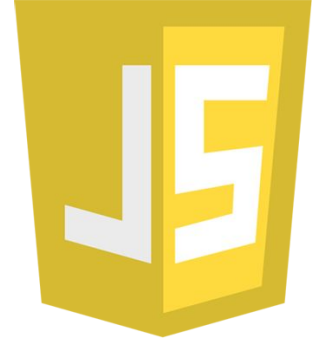
Compiled lang.:

- The source code is digested by a **compiler**.
- Generates machine-level instructions.
- Creates an executable file.

Compiled vs. Interpreted Programming Languages

Interpreted lang.:

- The source code is digested by a **interpreter**.
- Directly executes the instructions.
- The instructions are predefined by the **interpreter**.



Compiled vs. Interpreted Programming Languages (1)

Compiled - C, C++	Interpreted - Python, Matlab
Application is platform dependent. Requires to be compiled for the target CPU architecture and OS.	Application is platform independent. Will execute on any CPU architecture and OS as long as there's a compatible interpreter present.
Restricted syntax. Language offers low level operations. Lots of errors checked at compile-time.	Relaxed syntax. Language offers high level operations. All the errors occur only at run-time.
Low level optimization is directly available on language syntax.	Optimization is restricted by interpreter functionalities.

Compiled vs. Interpreted Programming Languages

Compiled - C, C++	Interpreted - Python, Matlab
Programmer responsible of memory management. Harder to program, and, thus, error-prone.	Automatic memory management. Programmer does not have to care of any memory allocations. Easier to program.
High performance. Executing machine instructions directly is efficient!	Not as efficient. Interpreted languages can be fast, but, basically, the need for interpretation is a source of inefficiency.
Good for real-time systems. It is easy to get guarantees on execution time.	Hard for real-time systems. No guarantees on execution time. Execution time of automatic garbage-collection is hard to predict.

**There's no better one! It depends on the use case.
But for real-time audio, compilation is the way to go!**

Introduction to C++ Development

The C++ Development Toolchain

Source files:

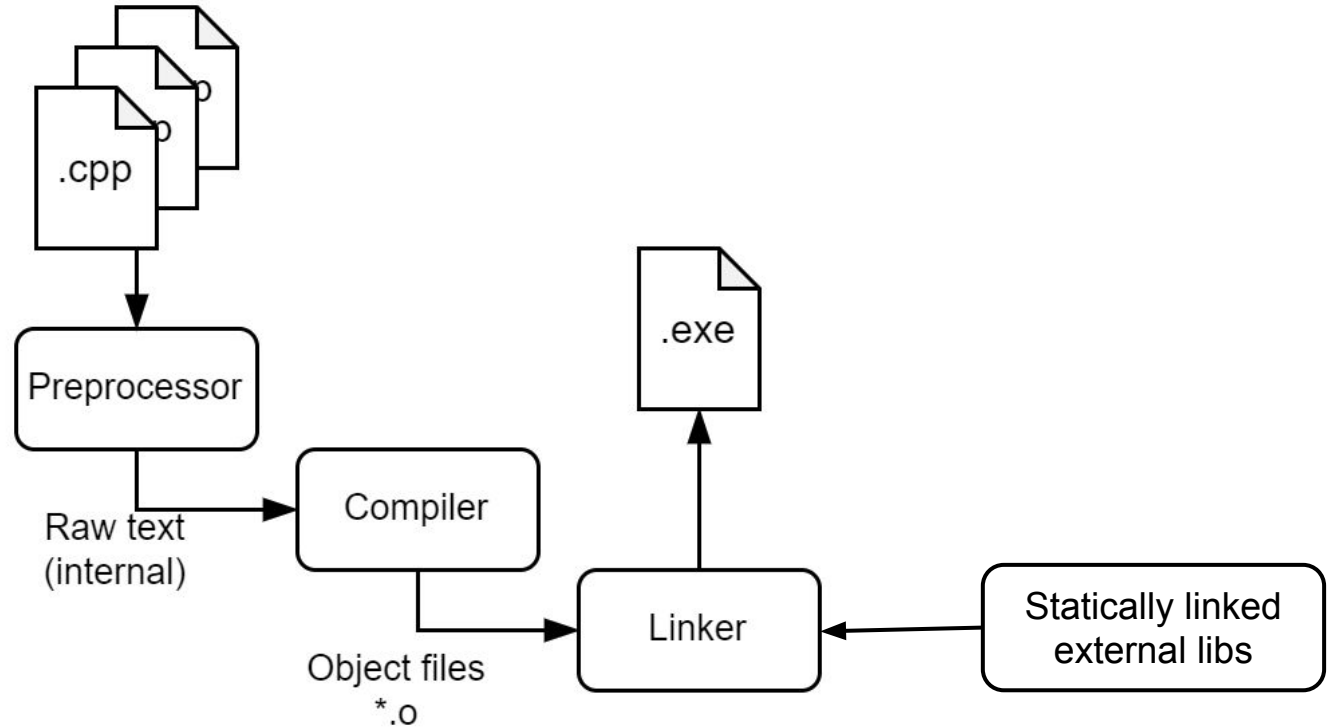
- ***.cpp**: actual program code
- ***.h**: header files that introduce the contents of the corresponding *.cpp files to other source files.
- ***.h**: external libraries provide *.h-files to tell what is inside the library.
- *.cpp-files typically `#include` multiple *.h-files.

The term “**to compile an application**” is a crude simplification of (at least) 3 steps:

- **Preprocessor**: Text processing of the **source files**.
- **Compiler**: Syntax parsing on the code, generates ***object files (*.o)***
- **Linker**: Digest the ***object files***, outputs the ***executable file***.

There can be more steps, but those are the most usual ones.

The C++ Development Toolchain



The C++ Development Toolchain

The Preprocessor:

Text processing on the source files.

- ***#include "another.h"***: Replaces with the contents of the specified file.
- ***#define macro(x)***: Replaces macro definitions with their actual text.
- ***#ifdef/#endif***: Conditionally removes text in between the directives.



The C++ Development Toolchain

The Compiler:

- Syntax parsing of the source code.
 - [Abstract Syntax Tree \(AST\)](#)
- Machine instructions generation.
- Allows outputs in a variety of forms:
 - **Intermediate Representation**
 - **Assembly (*.s)**
 - **Object file (*.o)**
- Optimisation steps.



The C++ Development Toolchain

The Linker:

“Link” **object files** and writes a **executable file** or a **dynamic library**.

- **Symbol resolution:** Replace function names with their actual memory address.
- **Link-time optimisation (LTO):** Reduce the amount of function calls by merging and in-lining.



The C++ Development Toolchain

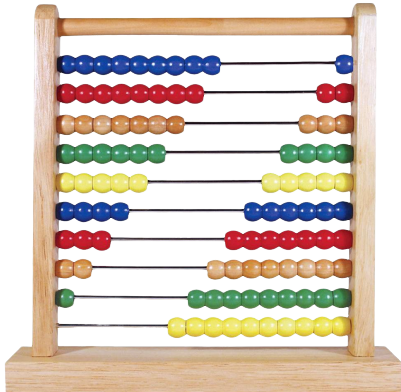
Other C++ toolchain elements:

Shipped with the compiler for advanced analysis and manipulation of executables and libraries.

- **Archiver**: **static library** (*.lib/*.a), which can later be linked to other executables.
- **Strip**: Remove the names from symbol tables.

The C++ Development Toolchain

- **Debug Build:** Used during development for evaluating and proper functionality.
 - + Fast build time
 - + Breakpoints and variables inspection
 - Higher CPU load
 - Larger executable size

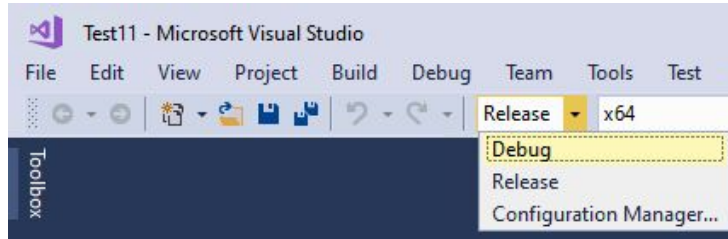


- **Release Build:** Used for “production ready” executable and measuring performance.
 - + Enable compiler optimisations
 - + Harder to reverse-engineer
 - Slow build time
 - Does not allow breakpoints and inspecting

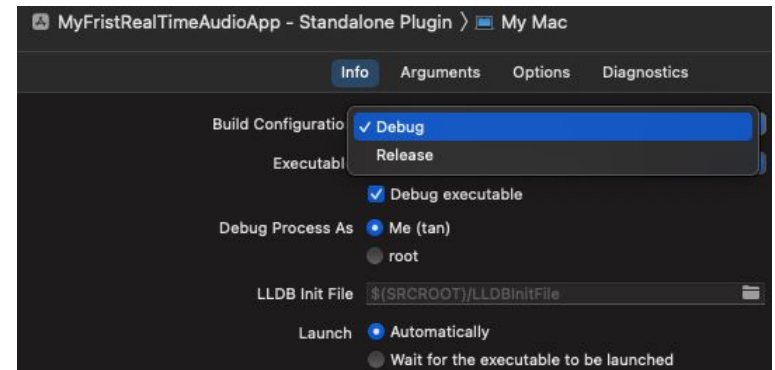
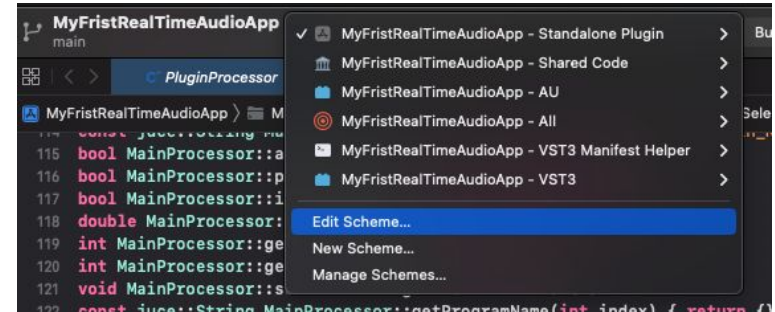


The C++ Development Toolchain

Visual Studio



XCode



Large language models for code development

Several tools available: GitHub Copilot is a popular one, able to use various LLMs such as GPT-4, Gemini (by Google), Claude Sonnet (by Anthropic)

To copilot or not to copilot?

- OK for boiler-plate code, but we aim to provide you with all such code in any case
- Not OK for the actual problems. Code it yourself such that you will learn!

After you have learnt coding here, you are in much better position to understand and debug code by a co-pilot!

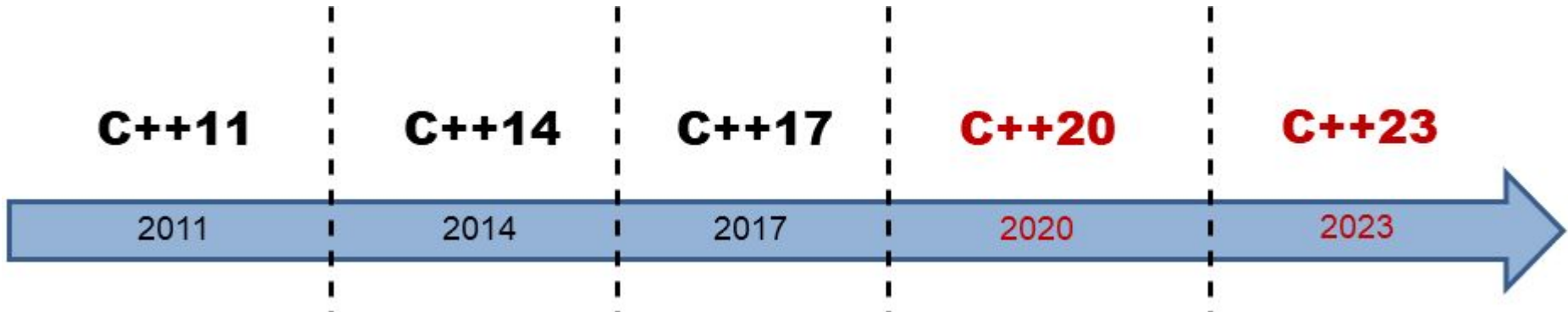
Introduction to C++

C++ Syntax Overview

The **C++** language is [frequently referred](#) as a **superset of C**. Both languages share a considerable amount of keywords and syntax.

[C++](#) standards have been evolving at a fast pace, increasing a amount of feature on every version.

Let's check a few of them...



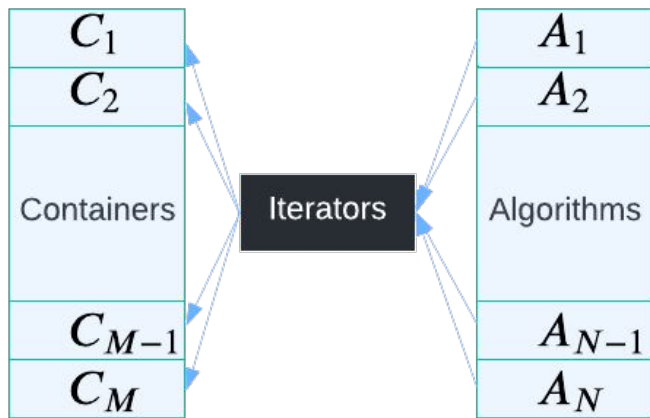
C++ Syntax Overview

- **Object Oriented Paradigm (OOP)**: Most discerning feature of C++, characterized by the [class](#) keyword.
- [new/delete](#): Dynamic memory allocation, replacing the C *malloc/free*.
- [Namespaces](#): Allows variables, function and class names management.
- [Templates](#): Type deduction, first [generic programming](#) feature of C++.
- [Lambda expressions](#): Possibly the most popular C++ [functional programming](#) feature, allows for higher-order functions.

The C++ Standard Template Library

Designed by [Alexander Stepanov](#), introduced with C++98 standard. Built around 3 basic concepts:

- **Containers**: Data structures such as: lists, array, maps, queue, etc...
- **Algorithms**: Various algorithms such as: sort, search, shuffle, etc..
- **Iterators**: Logic glue between *containers* and *algorithms*.



The C++ Standard Template Library

Smart pointers:

- Safer memory management.
- Built around the RAII concept.
- Primordial type of “garbage collection”

Example code

```
static const std::vector<mrta::ParameterInfo> ParameterInfos
{
    { Param::ID::Enabled,    Param::Name::Enabled,    "Off", "On", true },
    { Param::ID::Drive,      Param::Name::Drive,      "", 1.f, 1.f, 10.f, 0.1f, 1.f },
    { Param::ID::Frequency,  Param::Name::Frequency, "Hz", 1000.f, 20.f, 20000.f, 1.f,
0.3f },
    { Param::ID::Resonance,  Param::Name::Resonance, "", 0.f, 0.f, 1.f, 0.001f, 1.f },
    { Param::ID::Mode,       Param::Name::Mode,       { "LPF12", "HPF12", "BPF12",
"LPF24", "HPF24", "BPF24" }, 3 },
    { Param::ID::PostGain,   Param::Name::PostGain,   "dB", 0.0f, -60.f, 12.f, 0.1f,
3.8018f },
};
```


Example code 2

```
class MainProcessor : public juce::AudioProcessor {
Public:
    MainProcessor();
    ~MainProcessor() override;

    void prepareToPlay(double sampleRate, int samplesPerBlock) override;
    void processBlock(juce::AudioBuffer<float>&, juce::MidiBuffer&) override;

...

Private:
    mrt::ParameterManager parameterManager;
    juce::dsp::LadderFilter<float> filter;
    juce::SmoothedValue<float> outputGain;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(MainProcessor)
};
```


Audio and Real-Time Computing

What is a Real-Time Computing System?

- Not only the correctness of the compute is considered, but also the time that it happens.
- Computations should take place inside a **deadline** or **time-frame**.
- Designed to **react to a event**, usually originated by a timely hardware measurement.



Audio as Streaming Data

- Events are usually generated by a ***analog-to-digital*** and/or ***digital-to-analog converter*** (ADC/DAC). Commonly referred as ***audio interface***.
 - Those events are handled by the OS as [interrupts](#).
 - Interrupts results on a ***callback*** function call on the application responsible for the audio processing.
- 



Audio as Streaming Data

- Usual audio sampling rates goes from 32kHz to 192kHz.
- Would be impractical to handle a **callback** for every sample.
- Drivers bundle an amount of samples within a single *buffer* interrupt.
- The application needs to be able to process the buffer within the amount of time required to sample the audio buffer.

$$T_{\text{DEADLINE}} = N_{\text{BUFFER}} \times T_{\text{S}} > T_{\text{EXECUTION}}$$

Real-Time Audio Applications on Modern OS

- Modern operating systems offer simple ways to deal with audio interface drivers.
- Some of the most popular ones are:
 - [CoreAudio on MacOS](#)
 - [ASIO on Windows](#)
- JUCE makes it even easier!

Real-Time Audio and Concurrency

Audio Thread: Driven by HW interrupts.

- High priority execution in the OS
- Avoid unbounded time operations:
 - Memory allocation
 - Mutex locking
 - File I/O
- Avoid race conditions:
 - Lock-free structures
 - Atomic operations

Real-time Audio Application Distribution

Standalone Application

- The application is responsible for connecting to the audio drivers and processing the audio.



Real-time Audio Application Distribution

Plugin Application

- Used in a **Digital Audio Workstation (DAW)**, which is dynamically loaded and receives the audio buffers.
- There are [many standards for plugin distribution](#):
 - **VST2/3**: Available for Windows, MacOS and Linux.
 - **AudioUnit/v3**: Available only for MacOS.
 - **AAX**: Supported only by **ProTools** DAW, available for MacOS and Windows.



Audio Units

