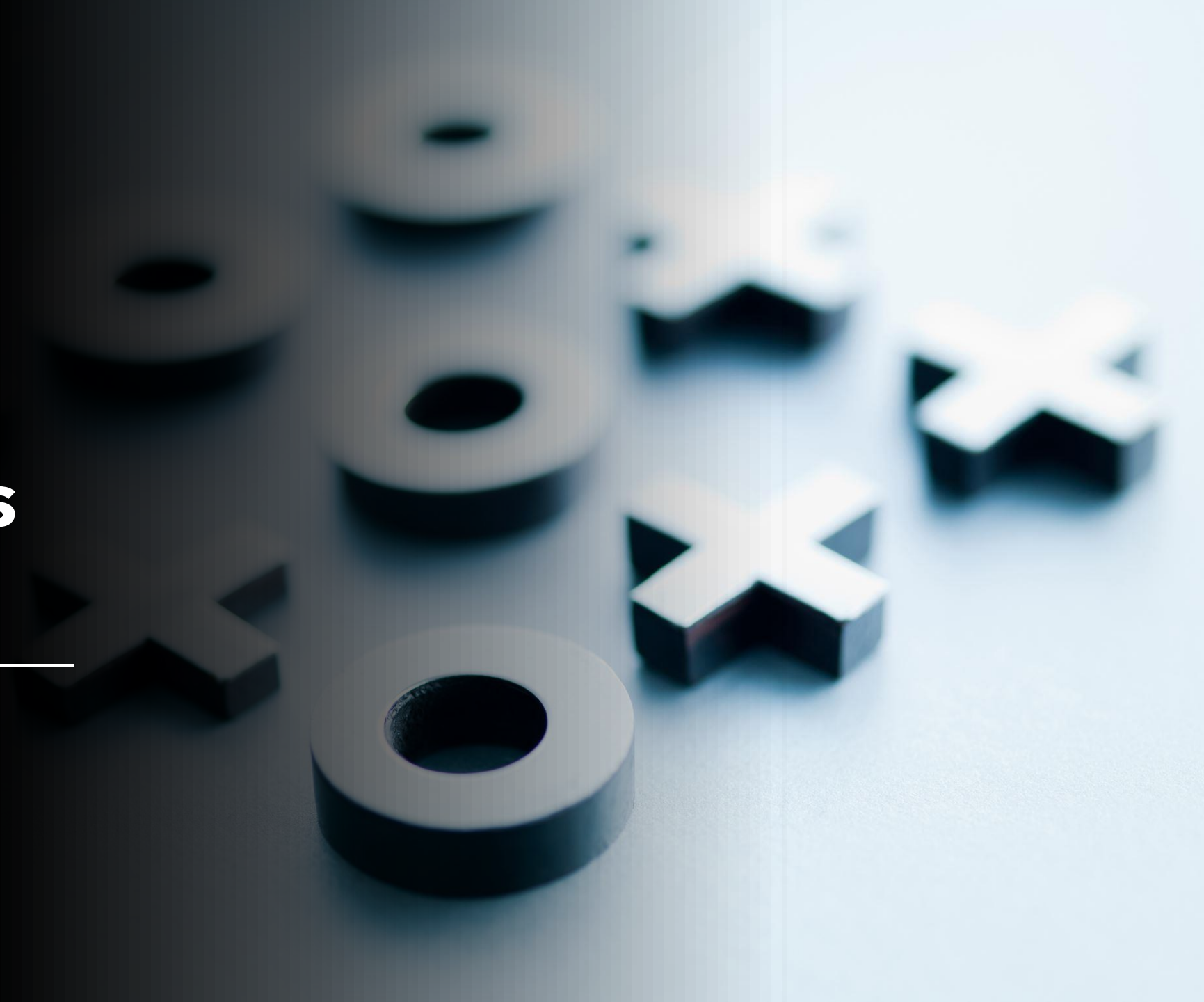
C++ in Constrained Environments



CSE 437 REAL TIME SYSTEM ARCHITECTURES

- Uğur Er
- Muhammed Bedir Uluçay
- Mustafa Karakaş

We do not Live in Alice's Wonderland

- Why do design of programs differ from each other?
- What drives us to design differently?
- Why do we exist, what do engineers do?

- When you examine lectures given in college, people discussing problems, you'll often find system constraints ignored. It does not look like a big problem, but it is.
- Many systems come with certain restrictions when writing our programs. This requires that we act with these constraints in mind when designing our program. Therefore, these constraints greatly affect the design of our program.

What is a Constraint?

- A constraint is anything that slows a system down or prevents it from achieving its goal. You could think of a constraint as a bottleneck in your processes that impedes your progress. There are many, many different types of constraint.
    - Limited space for data
    - Limited processing power
    - Zero or very limited downtime
    - Long service life
    - And more...
- Not every system has same constraints.
- You can not solve every problem with just one language, language feature, library etc.

Common Misconceptions?

- Learning C is prerequisite for learning C++.
- Good C++ relies on a lot of dynamic memory use and large class hierarchies with run time resolution of all function calls.
- To be reliable, code must be littered with run-time tests
- Embed system must avoid exceptions.
  - True only for some tiny systems and some hard-real-time systems.
- To write good code you must use the latest features

Who writes "constrained systems"? Why constraints occur?

- Professional programmers
  - Sometimes with limited domain knowledge
- Engineers with no programming training
  - Often with extensive domain knowledge
- Everything in between

About C++

- Technically, C++ rests on two pillars:
  - A direct map to hardware
  - Zero-overhead abstraction in production code
- Key foundation of the C++ systems:
  - Static type system
  - Systematic and general resource management
  - Efficient OOP
  - Flexible and efficient generic programming
  - Compile-time programming
  - Direct use of machine and operating system resources

```
#version 330 core
out vec4 color;

uniform float uMoveX;

void main(){
    color = vec4(abs(1.0f + uMoveX), abs(1.0f - uMoveX), 1.0f, 1.0f);
}
```

```
glShaderSource(shaderId, 1, &shader, nullptr);
glCompileShader(shaderId);

int errorResult;
glGetShaderiv(shaderId, GL_COMPILE_STATUS, &errorResult);
if(errorResult == GL_FALSE){
    int length;
    glGetShaderiv(shaderId, GL_INFO_LOG_LENGTH, &length);
    char* message = (char*)alloca(length);      // stack de dinamik olr
    glGetShaderInfoLog(shaderId, length, &length, message);

    std::cerr << "Failed to compile " <<
        ((type == GL_VERTEX_SHADER) ? ("Vertex") : ("Fragment")) << " s
    std::cerr << message << std::endl;
    glDeleteShader(shaderId);
}

glAttachShader(pId, shaderId);
```

Express Intent

- Make every construct checkable
- You may not always be able to check
  - Static (compile-time) checking can be difficult or impossible. But it's the ideal: guaranteed and no run-time cost
  - Run-time check can be expensive. But if it is checkable, we can check in test runs.

- Traditional (C-style)

```
union U { int x, string y};        // &U::x is the same as &U::y
U u;
u.x = 7;
string s = u.y;                    // unsafe! (crash, if you are lucky)
```

- Variants (C++ standard library)

```
variant<int,string> v;
v = 7;
bool b = has_alternative<string>(v);     // knows its type
string s = get<string>(v);               // run-time error (exception)
```

Ideally, the better version also has the cleanest notation and zero-overhead

Protect The Data

- Use immutable data
  - You can not have a race condition on a constant
  - It's hard to be faster than a simple read of a value
  - **const** values, **const** T* and **const** T& interfaces
  - **constexpr** and **consteval** functions
- Initialization
  - Variables should almost always be initialized
- Bit manipulations are messy, error-prone, often essantial.

  There is support in C++ with std::bitset
- We cant ignore problems just becasue their solutions are hard and messy

```
const int data_1 = 2;
const int* data_2 = &data_1;
const int& data_3 = data_1;
constexpr int multiply(const int a,const int b){
    return a*b;
}
constinit/consteval

int data_un;
int data_in = 0;
X x{};          // default
string s;       // ""
vector<int> c; // {}
Struct S {int s = 0; ...};
```

Abstraction

- Zero-overhead principle
  - What you don't , you don't pay for
- Zero-overhead does not mean "zero cost"
  - You'll always pay for what you use
- User-defined types
  - User-defined types are just like built-in types

Resource Management

- A resource is something that must be acquired and released
  - Memory, file handles, locks, sockets…
- Resource release must be
  - Implicit
  - Guaranteed
  - Predictable
  - Not dependent on significant run-time support
- Scope-based resource management (RAII)
  - In RAII, holding a resource is a class invariant, and is tied to object lifetime: resource allocation (or acquisition) is done during object creation (specifically initialization), by the constructor, while resource deallocation (release) is done during object destruction (specifically finalization), by the destructor.

Run-time to Compile-time

- Inlining of function calls
- Copy elision and move constructors
  - Avoid copying (e.g., copying of function results)
- Avoid virtual member functions when possible
- Constant expression evaluation and constexpr functions
- If you can move work to compile-time, there is:
  - Less code to write
  - Less code to store in memory
  - Less code to execute

Compile-time Computation

- Type-rich programming at compile time
  - Usually much smaller and faster
  - No run-time error-handling needed
- Not just built-in types
  - Examples from <chrono>
- Compile-time computation tends to be invisible

Examples from <chrono>

```
cout << weekday{June/21/2016};                    // Tue
static_assert( weekday{June/21/2016}==Tuesday );  // At compile time

static_assert(2020y/February/Tuesday[last] == February/25/2020);  // true


auto tp = system_clock::now();
cout << tp;                                       // 2019-11-14 10:13:40.785346
cout << zoned_time tp{current_zone(),tp};         // 2019-11-14 11:13:40.785346 CET
```

Examples from auto

```
auto z = sqrt(3+2.7i);                    // call sqrt(complex<double>)
auto d = 5min+10s+200us+300ns;            // a duration
auto s = "This is not a pointer to char"s;// a string

// implementations:
constexpr complex<double> operator""i(long double d) { return {0,d}; }
constexpr seconds operator""s(unsigned long long s) {return s; }
constexpr string operator""s(const char* str, size_t len) { return {str, len}; }
```

Compile-time Computation and Static Polymorphism

- Use overloading or compile-time **if**

```
template<typename T>
void writer2(T* p, unsigned n)
{
      if constexpr(is_same_v<T,char>)
            Character_device::write(p,n);
      else
            Block_device::write(p,n);
}

void user2(Packet* pp, int ps, char* pc, int cs)
{
   writer2(pp,ps);
   writer2(pc,cs);
}
```

```
template<typename T>
void writer(span<T> s)
{
      if constexpr(is_same_v<T,char>)
            Character_device::write(s.data(),s.size());
        else
            Block_device::write(s.data(),s.size());
}

void user1(span<Packet> sp, span<char> sc)
{
        writer(sp);
        writer(sc);
}
```

Exceptions & Errors

- Throw an exception when?
- A guarantee that all errors are handled is needed(often forgotten).
- An error must propagate back up the call chain to the "ultimate caller" before being caught. (often forgotten).
- No suitable return path for errors codes are available.
- Avoid overusing try/catch statements in your code.
  - Expensive
  - Replicates the problem of using return codes.
- Return an indicator?
- A failure is normal an expected.
- Caller can handle the Calle's returned error.
- When the memory has a so little memory that the run-time exceptions would crowd out essential functionality.

- Rely on RAII handle resources systematically

Thanks For Listening