Interview Review Chart

Karl Solomon

December 17, 2024

1 Concepts

1.1 Concurrency vs Parallelism

Concurrency When two or more tasks can start/run/complete in overlapping time periods. This does not necessarily mean they'll be running in overlapping time periods. Examples include:
RTOS

Parallelism When tasks run at the same time (e.g. on a multi-core CPU)

Multithreading When multiple tasks are running on a CPU. This can be implemented truly parallel where each task has access to separate HW/core. However, more common in desktop applications is SMT.

SMT (Simultaneous Multithreading) Multiple threads share 1 core. The thread instructions are pipelines s.t. they run mostly in-parallel, and when one is waiting for I/O the other can run uninhibited, however since only one thread can access a dedicated HW block at any given time they are not truly parallel.

2 Operating Systems

3 C

3.1 Preprocessor

```c
# // stringizes the macro parameter
#define stringify(x) #x
#define foo 1
stringify(foo) // --> evaluates to "foo", NOT "1"
```

```c
## // concatenates the macro parameter
#define COMMAND(NAME)  {#NAME, NAME ## _command}
struct command commands[] = {
  COMMAND(quit), // equivalent to {quit_command}
  COMMAND(help), // equivalent to {help_command}
}
```

- predefined macros
  - __FILE__
  - __LINE__
  - __DATE__
  - __TIME__

  - __STDC_VERSION__
  - __cplusplus
- item2
- item3
- item4

3.2 Peripherals

- **I2C**
  SDA is data, SCL is clock. PURs typically in the 1-4.7k range. Too weak = slow comm and errors. Clocks are usually 100k-1MHz. Addr can be 7 or 10 bit. This is rate-limiter for number of slaves, though line impedance would increase for each slave. Here are some use usage examples:

  1. Master sends START and slave Addr
  2. Master sends data to slave
  3. Master terminates with a STOP

  1. Master sends START and slave Addr
  2. Master sends data to slave
  3. Master sends repeatedSTART and either sends more data to slave or receives data from slave.
  4. Master sends STOP

- **SPI**
- **UART**
- **USB**
- **HDMI**

4 C++

4.1 Classes

4.2 Containers

- Sequence
  - **array**

```
std::array<int, 3> arr; // uninitialized (whatever was in
    memory before)
std::array<int, 3> arr = {}; // initialized as 0s
std::array<int, 3> arr1 = {1, 2, 3};
std::array<int, 3> arr2{1, 2, 4};
arr1.fill(0); // fills array with 0s
arr1.swap(arr2); // swaps contents of arr1 and arr2
```

  - **vector**

```cpp
std::vector<int> v;
v.capacity(); // size of currently allocated memory
v.shrink_to_fit(); // releases unused memory
v.reserve(100); // pre-allocates 100 elements
v.clear(); // erases all elements
v.erase(v.begin()); // erases first element
v.push_back(1); // adds 1 to the end
v.rbegin(); // reverses iterator
std::erase_if(v, [](int x) { return x > 10; }); // removes all elements > 10
std::vector<Pair<int,int>> classV;
classV.emplace_back(10,1); // create Pair object and push to back
```

  - **inplace_vector**
  - **deque**

- Associative
  - **Set**
  - **Map**
  - **Multiset**
  - **Multimap**

- Unordered Associative
  - **unordered_set**
  - **unordered_map**
  - **unordered_multiset**
  - **unordered_multimap**

- Adaptors
  - **stack**
  - **queue**
  - **priority_queue**
  - **flat_set**
  - **flat_map**
  - **flat_multiset**
  - **flat_multimap**

4.3 Modern C++

- C++11
  - **Alias Templates**
  - **atomic**
    Well-defined behavior in the event of RMW race contition. Accesses to atomics may establish inter-thread synchronization and order non-atomic accesses.

```cpp
atomic_bool b; // same as std::atomic<bool> b;
```

  - **auto**
  - **constexpr**
  - **final**

* Specifies that a class cannot be inherited from.
* When used in a virtual function, specifies that the function cannot be overridden by a derived class.
* final is also a legal variable/function name. Only has special meaning in member function declaration or class head.

```
struct Base
{
    virtual void foo();
};
struct A : Base
{
    void foo() final; // Base::foo is overridden and A::foo is the final override
    void bar() final; // Error: bar cannot be final as it is non-virtual
};

struct B final : A // struct B is final
{
    void foo() override; // Error: foo cannot be overridden as it is final in A
};

struct C : B {}; // Error: B is final
```

- **initializer list**
- **iota**

```
void iota(ForwardIterator begin, ForwardIterator end, T v); // fills range [first-last] with sequentially increasing values starting at v in begin
```

- **lambdas**

capture comma-separated list of variables which are captured/modified by the lambda. Captures cannot have same name as input parameters.

Capture list

* & = capture all used variables by reference
* = = capture all used variables by copy
* varName = by-copy
* varName... = by-copy pack-expansion
* varName initializer = by-copy w/ initializer
* &varName = by-reference
* &varName... = by-reference pack-expansion
* &varName initializer = by-reference w/ initializer
* this = by-reference capture of current object
* *this = by-copy capture of current object
* ... = by-copy capture of all objects w/ pack expansion
* ... initializer = by-reference w/ initializer and pack expansion

```
// If the capture-default is &, subsequent simple captures must not begin with &.
[&] {};           // OK: by-reference capture default
[&, i] {};        // OK: by-reference capture, except i is captured by copy
[&, &i] {};       // Error: by-reference capture when by-reference is the default
[&, this] {};     // OK, equivalent to [&]
[&, this, i] {}; // OK, equivalent to [&, i]
```

```cpp
// If the capture-default is =, subsequent simple captures
   must begin with & or be *this(since C++17) or this(since C
   ++20).
[=] {};          // OK: by-copy capture default
[=, &i] {};      // OK: by-copy capture, except i is captured by
    reference
[=, *this] {}; // until C++17: Error: invalid syntax
               // since C++17: OK: captures the enclosing S2
                  by copy
[=, this] {};  // until C++20: Error: this when = is the
   default
                    // since C++20: OK, same as [=]
```

- **mutex**
- **override**
- **random**

```cpp
            #include <stdlib>
            int rand(); // returns integer in [0, RAND_MAX]
```

```cpp
            #include <random>
            // default_random_engine
            // philox4x64 -> philox_engine
            // random_device = non-deterministic generator
               based on hardware entropy
            std::random_device rd;
            rd.entropy(); // estimate of random number device
               entropy. Deterministic entropy = 0.
            std::uniform_real_distribution<double> dist(0.0,
               1.0);
```

Distribution list

* uniform
  - int
  - real (double)
* bernoulli
  - bernoulli
  - binomial
  - negative binomial
  - geometric
* Poisson
  - poisson
  - exponential
  - gamma
  - weibull
  - extreme_value
* Normal
  - normal
  - lognormal
  - chi_squared
  - cauchy
  - fisher_f
  - student_t
* Sampling

        - discrete
        - piecewise_constant
        - piecewise_linear
        - item4
    - **range-based for**
    - **thread**
    - **trailing return type**
- C++14
    - **Variable Templates**
    - **Generic Lambdas**
- C++17
    - **tuple**
    - **execution policies**
- C++20
    - **Modules**
    - **Coroutines**
    - **Ranges**
    - **Midpoint**
    - **using enum**
    - **constinit**
    - **string formatting**
    - **template concepts**
- C++23
    - **print/println**
    - **byteswap**
    - **flat_map/flat_set**

4.4 C++23