richardwu.ca

CS 343 Course Notes

Concurrency

Peter Buhr • Fall 2018 • University of Waterloo

Last Revision: October 10, 2018

Table of Contents

Abstract

These notes are intended as a resource for myself; past, present, or future students of this course, and anyone interested in the material. The goal is to provide an end-to-end resource that covers all material discussed in the course displayed in an organized manner. These notes are my interpretation and transcription of the content covered in lectures. The instructor has not verified or confirmed the accuracy of these notes, and any discrepancies, misunderstandings, typos, etc. as these notes relate to course's content is not the responsibility of the instructor. If you spot any errors or would like to contribute, please contact me directly.

1 September 11, 2018

1.1 Advanced control flow

Everything herein pertains to control flow *within* routines:

- Use break guard clauses (early breaks)
- **Avoid** flag variables: instead using an infinite loop with break statements.

  However, in some cases one may use flag variables if absolutely necessary (e.g. memoizing some status that occurs much later and would be hard to modify).
- Use nested control structures with multi-level breaks (with labels)

Rules for gotos:

- No backward breaks/gotos: use a loop's inherent looping capabilities
- No jumping into the middle of code

tl;dr: use gotos for *static* multi-level exit (to simulate labelled breaks/continues).

1.2 Dynamic allocation

Use stack allocation over dynamic allocation whenver possible: e.g. `int arr[size]` as opposed to `int *arr = new int[size]` and `delete [] arr` (although variable-length stack arrays are not part of the C++ standard, use it whenever possible).
However, heap allocation may be necessary if:

- memory needs to persist outside of the scope memory was initialized in
- unbounded input size (e.g. initializing values from `STDIN` into a `vector`)
- array of objects with variable initialization parameters
- when allocation would overflow a small stack

1.3 Control-flow between routines

For *dynamic* multi-level exit (call/return semantics between routines where exit points are not known at compile time) use a global label variable which is referred to inside subroutines with `goto`s for jumping between multiple function stack frames. Assigning label literals to the variable at various points in time can alter where the subroutines end up jumping to.
`jump_buf`, `setjmp`, `longjmp` initialize, set, and jump to a label variable, respectively, in C.
Traditional approaches to what we described include:

- Return codes. Disadvantage: mixes exception and normal results, and checking code or flag is optional.
- Status flags via global variable (e.g. `errno` in UNIX). Disadvantage: may be modified by other routines (mixed out).
- Fixup routines (or callbacks). Disadvantage: adds overhead with additional function calls.
- **`return union`**: returning union types (e.g. result or return code). Disadvantage: must check return type on every call (optional). Multiple values must be returned to higher-level calls (intermediate function frames need to *forward* nested return codes).

1.4 Static vs dynamic multi-level exit

Static multi-level exit occurs when exit points are known at compile time (e.g. with literal break labels that are in the code).
Dynamic multi-level exit occurs when there can be multiple outcomes depending on run-time conditions (i.e. invoking routines and exits between routines depending on the current execution stack, which is dynamic).

2 September 13, 2018

2.1 Exception handling

Complex control-flow *among routines* is called **exception handling** (it is more than just error handling).
While it may be simulated using simpler control structures (as described above), it is difficult in general and more messy.
Depending on the execution environment (e.g. object-oriented vs non-object-oriented where we may have `finally` destructors and inherited destructors; concurrent vs sequential where we may have multiple execution stacks), the exception handling mechanism (EHM) implemented by the language/compiler must be adapted accordingly.

2.2 Static vs dynamic call/return

Similar to static/dynamic multi-level exit static calls/returns can be statically inferred from the code itself whereas dynamic calls/returns depend on the current execution stack (i.e. what function frames are on the stack).
Normal routines (e.g. `foo()` method definition and a call to `foo()` with no virtual methods) is a **static** call with a **dynamic** return (dynamic because it returns to the block that invoked `foo()`, which depends on the stack).

	call/raise	
return/handled	static	dynamic
static	1) sequel	3) termination exception
dynamic	2) routine	4) routine pointer, virtual routine, resumption

Figure 2.1: Chart summarizing the classifications of each call-return static/dynamic pairs.

Summary of why these are static/dynamic calls/returns:

Sequel A named routine that can be invoked statically and statically returns to the *end* of block in which it was declared.

Disadvantage: the declaration and invocation must be statically compiled together, i.e. invocations cannot be compiled separately from the declaration.

Note that blocks (section of code enclosed in `{ }`) are pushed onto the stack (think of local variables declared in block being pushed onto/popped off stock), thus sequel's will need to *unwind* the stack when returning to the end of its declaring block.

Termination An exception is thrown and some arbitrary handler (which is a routine itself) handles it (dynamic call). Since control cannot be returned to the raise point (i.e. **termination**), it finishes executing the handler routine and **statically** returns to the line after the handler's definition.

Virtual routine/resumption Virtual routine is calling a function pointer (dynamic call) where the virtual routine returns to the invocation point (dynamic return, depends on current execution).

Resumption is a mechanism where something like an exception is raised and propagation occurs to the handler (dynamic call), then the handler returns back or resumes to the raise block (dynamic return).

3 September 18, 2018

3.1 `_Resume` vs `_Throw` in μC++

In uC++, when a `_Resume` is "thrown", it looks for a handler to `_CatchResume` and perform fix up (which subsequently returns to the point of where the event was raised i.e. at the beginning of the `_Enable`). If no `_CatchResume`s can be found up the stack, then by default (`defaultResume`) the exception is **thrown** via `_Throw`.
Notice that `_Throw` will cause all blocks between the raise block and the guarded block that catches the exception to unwind from the stack, whereas `_Resume` does not. This is why `_Throw` does not have an `_At` clause (otherwise it'll cause the coroutine targetted to unwind its stack, which is not good practice).

3.2 Multiple catch clauses

In most programming languages, multiple catch clauses will not be re-evaluated, even if the exception handler in the first catch clauses throws an exception that could be caught by subsequent catch clause handlers.
In μC++, this is different for `_CatchResume` followed by `catch`: since `_CatchResume` *does not* unwind the stack, thus the `catch` clause in the guarded block stack frame can still be observed.

4 September 20, 2018

4.1 Caveat with running off co-routines

When a `resume()` to a co-routine causes the co-routine to terminate at the end of `main()`, it will actually `resume()` to the **starter** i.e. the first caller of `resume()`.
So if the first `resume()` was invoked in the constructor of the coroutine object, subsequent calls to `resume()` are made by a different block of code, and some other caller who invokes `resume()` causes the coroutine to terminate, the code will actually resume to the starter (first coroutine that called `resume()`), which is `main()`. So `main()` will continue executing from its last stack point.

5 September 25, 2018

5.1 Uncaught local co-routine exception

If a local `_Throw` is thrown and not caught inside a coroutine, then a `_Resume` event `uBaseCoutine::UnhandledException` is propagated to the **last resumer** (it hooks onto the last resumer). Therefore if the **last resumer** invoked a `resume()` which caused the local exception, it must always check if there are any exceptions hooked on.

If there are, the resumer will check its handler first for `_CatchResume` and then (by the default logic for `_Resume` events) for `catch`.
If `_CatchResume` catches the unhandled exception, it will do a dynamic return back to the `_Enable` that surfaced the non-local exception.
If `catch` catches the unhandled exception, it will do a static return to the code after the `catch` handler since the stack will unroll.

5.2 Formal definition of `resume()` and `suspend()`

When a `resume()` is invoked, it *inactivates* `uThisCoroutine()`, activates `this` (which is a context switch from `uThisCoutine()` to `this`).
Therefore, `this` must be a coroutine object itself (i.e. one must invoke `resume()` inside a coroutine's member function).
When a `suspend()` is invoked, `this` context switches back to the *last resumer*.
So for example, if `int main()` invokes `routine.foo()` where `foo()` contains a `resume()`: just before `resume()` is invoked `uThisCoroutine() = int main()` and `this = routine`.
Caveat: note that if we decide to do recursive `resume()`s inside a coroutine, it will overwrite the last resumer: thus we may lose track of `int main()` i.e. we do not keep a stack of resumers. Therefore, a `suspend()` after a recursive `resume()` will suspend back to itself. This is why when a coroutine terminates, it returns to its **starter**: this allows us to get back to `int main()`.
This also prevents our stack from overflowing if we are using full-coroutines and end up doing many `resume()`s.
Note that `resume()` and `suspend()` are complementary: if `resume()` is initiated and an arrow is drawn in one direction, `suspend()` traverses the arrow in the opposite direction.

6 September 27, 2018

6.1 Notes on `_Coroutine` in μC++

- A `_Coroutine` that has not been started is an object: only when it has started and not terminated is it a "coroutine"
- Member/class variables for a `_Coroutine` initialized on the some `int main()` thread lives on `int main()`'s stack.

  Any local variables initialized inside `void main()` (coroutine main function) is created on the *coroutine's stack* (which may be context switched during `resume()`s and `suspend()`s).

  Therefore coroutines actually have a reference to member variables on the `int main()` stack.

6.2 Dichotomy between semi- and full-coroutines

In semi-coroutines, we never `resume()` another co-routine while in a co-routine (other than `int main()`). However, full-coroutines can `resume()` inside another coroutine's member function.
This implies that full coroutines may be `suspend()`ed back to or "woken up" inside another coroutine's member function, whereas semi-coroutines are always `suspend()`ed back to inside itself.

6.3 `_Enable` (and `_Disable`)

Note that `_Enable` is not required to throw (i.e. `_Resume ... _At`) a nonlocal event at a different coroutine.
It is also **not required** to always enclose everything with `_Enable` in the receiving coroutine (i.e. when the receiving coroutine is inactive).

One can have a try-catch surrounding an empty `_Enable{}` to receive any queued up events.