Advanced Control Flow

Arnav Gupta

September 4, 2024

Contents

1 Basics of Advanced Control Flow

Within a routine, basic and advanced control structures allow any control flow.

`while` and `for` are interchangeable for predicate only loops. When using loop indexes, use `for` only.

Multi-exit loops have 1+ exit locations occurring within the body of the loop (such as `break`). With multi-exit loops, exit conditions are reversed from `while` and outdented. These eliminate priming (duplicated) code.

Multi-exit loops should not be used to simulate `while` or `for` and loop exits never need an `else` clause. Multi-exit loops allow multiple exit conditions.

Flag variables that only affect control flow should not be used, since they are equivalent to a `goto`.

1.1 Static Multi-Level Exit

Static multi-level exit exits multiple control structures where the exit point is known at compile time, done with labeled exit (`break`, `continue`).

Labeled exits helps eliminate all flag variables and can help remove duplicated code.

Normal and labeled `break` are `goto` with limitations:

- cannot loop (only forward branch) so only loop constructs branch back
- cannot branch into a control structure

Only use `goto` to perform static multi-level exit (simulate labeled `break` and `continue`).

1.2 Dynamic Memory Allocation

Stack allocation preferred over heap allocation (eliminates storage-management and more efficient).

Situations where dynamic heap allocation is necessary:

1. when storage must outlive the block in which it is allocated (ownership change)
2. when the amount of data read is unknown
3. when an array of objects must be initialized via the object's constructor and each element has a different value
   (a) use stack with `uArray` if possible (allocates without constructing and proves subscript checking)
   (b) calls to `()` or `make_unique<T>()` initialize array elements
   (c) allocation is $O(1)$, so better than unique pointer and vector
4. when large local variables are allocated on a small stack
   (a) `uArrayPtr` dynamically allocates the array in heap and implicitly frees it at the end of the block (like `unique_ptr`)
   (b) alternatives are large stacks (waste virtual space) or dynamic stack growth (complex and pauses)