Main topics of the week:
- Control Flow Constructs
- Sequencing and expression evaluation
- Blocks

Recall that imperative languages are action oriented – their execution is modeled after the abstraction of the machine, which executes instructions sequentially, and at each step has a current state of the machine. So the essential characteristics of an imperative language are assignment and control flow. The computation in an imperative language always has a current state – this can be changed as variables get assignment. Likewise, the current control point can change via control flow constructs. The basic model is that of a Random Access Machine. In assembly language, control flow is determined by jump statements, basically unconstrained gotos. One of the design goals of a language is to make the program more understandable, so that the text of the program helps us to understand the program. This is the basic idea of structured programming – the program is organized into procedural blocks, which are further decomposed into control flow blocks. Also, an imperative programming language should allow the underlying assignment oriented machine to be used directly and efficiently. To these ends, we focus on four concepts that drive the design of an imperative language.

Sequencing. To be able to program effectively with the underlying machine model in mind, it is important that the language reflect the same sequencing of instructions. Typically, this means that the order of constructs in the program text corresponds to the order in which they are executed. Note that in some languages, notably logic languages, sequencing is unimportant – the program text reflects logical constraints and the program model determines an order of evaluation that results in finding satisfying values for the constraints. However, in the imperative model, sequencing is very important. Thus, when faced with understanding a C program, the first thing to do is to locate the main procedure since we know execution will begin there, and we can follow the execution by looking for the next statement. Even in an event driven environment (like a GUI), the execution is still purely sequential. Libraries may hide the linkage of the sequences, but from the language standpoint, statement execution is still sequential.

Subtle issues of sequencing arise when we start dissecting statements themselves, particularly statements that consist of an assignment. In a language like C, a statement may consist of just an expression, and the assignment expression commonly appears as an expression statement for its side effect. The assignment expression is truly an expression, and has a value (the value assigned), but the programmer is typically thinking about the side effect of modifying a memory location. In fact, one could almost characterize C (or most imperative languages) as “programming by side effect”. That is, as the sequential execution unfolds, memory locations have their values changed by assignment, which in turn affects control flow. Because expressions can have side effects, the sequencing within expression evaluation becomes important. We can look at expressions in C to understand the importance of sequencing in expression evaluation.

First, expressions often consist of operators and operands, and may use parentheses for grouping or not. In the absence of parentheses for controlling grouping, the language
must define **precedence rules**. So for example, an important reference for C programming is the precedence table, which lists the many operators (arithmetic, logical, relational, bit, increment, member, assignment), and these operators have a precedence hierarchy that specifies which operators have higher precedence than others. E.g., multiplication has higher precedence than addition, meaning that a+b*c groups as a+(b*c). In addition to operator precedence, the language specifies the **associativity** of operators – whether it is left to right or right to left. Thus, since addition associates left to right, a+b+c groups as (a+b)+c. Ideally, precedence and associativity rules will reflect some “natural” or intuitive semantics of the expressions, so that the language “does the right thing” when the absence of parentheses requires interpretation by the compiler.

However, beyond the grouping implied by the precedence and associativity, the language may not be specific about order, and this is the case with the C language. That is, the **order of evaluation** is a distinct issue from the grouping order in the expression. When we see an example like a+b*c, we tend to think of evaluation order (the b*c is done first), but all that the language specification says is that b*c is an operand to plus. Naturally, you would guess that it must be done first before it can be added to a, but this is kind of like an “implementation detail” and not required explicitly by C. The important thing here is to understand what the language does guarantee about evaluation order. In C, order is only guaranteed to be left to right for the logical **and** and **or** operators, for the ternary operator, and for the comma operator. The prefix/postfix increment operators imply a sense of order of evaluation in that prefix means the value is the incremented value, and postfix means the pre-incremented value, so one would guess that the implementation has to sequence the incrementing accordingly. The C language leaves a lot undefined here, ostensibly to allow the compiler implementer to make efficient choices. The cost to program understanding is that the program may behave differently for different compilers, thus reducing portability. Good programming practice dictates that programs should not depend on undefined behavior, but it can be difficult to know when such dependence has crept in.

C and Java provide a whole class of **compound operators** to allow for more compact expressions, but also to avoid evaluation order problems. That is, for a common usage such as a=a+b, if the evaluation of a had side effects, not only would we be unsure whether the first or second occurrence was evaluated first (which could certainly affect the action taken), but we probably don’t want two distinct evaluations to start with since we don’t want the side effect to be doubled. Thus, something like a[i++] += b is well defined and predictable in its effect.

On the other hand, Java does specify evaluation order as part of the language specification. Evaluation is from left to right. This is spelled out in the Java specification for various instances where it might occur: Operands are evaluated left to right, and before the operator is evaluated; function arguments are evaluated left to right. As in C, the conditional operators **&&**, **||**, and **?:** have guaranteed evaluation order, with short circuiting. This is certainly an improvement from C (or C++) where evaluation order is mostly undefined, which can produce surprises from complicated expressions where the operands have intertwined side effects. However, even the Java documentation says “It is recommended that code not rely crucially on this specification.” That is, good programming design should not be dependent on relatively obscure evaluation order issues.
One reason for spending some time harping on the evaluation order in imperative languages is to gain an appreciation for the independence from evaluation order afforded by pure functional languages. It also draws attention to the complexity of static analysis – if you think of this issue from the compiler’s point of view, it would have to consider all possibilities for side effects and their interaction to determine if it could rearrange the order of evaluation without affecting the result. C takes the easy (and somewhat dated) way out, leaving the result undefined, so the compiler can do whatever is convenient, probably to the programmer’s surprise.

Control Flow. In the 1970s, structured programming became the preferred design goal of languages. The model for imperative languages was the machine execution itself, and assembly language programs are basically a maze of branches and conditional branches, or goto’s. The goto was painted as the evil weakness in programming to be avoided. Some languages prohibited the goto but in any case its use was highly discouraged. Java does not permit the goto (although it reserves the word goto, probably to make for more coherent error messages for C programmers used to using it). However, the structured approach does not do away with branching and jumping, it simply means that it is highly predictable by language constructs. The basic constructs are selection (if-then-else), iteration (over an enumeration, e.g., for; or over a logical condition, e.g., while) and branch tables (case statements). Most imperative languages have these language constructs and allow the code to be organized into blocks controlled by these constructs. They turn out to be sufficient to express most any imperative algorithm. Slightly more exotic branching is done with loop early termination and continuation (break and continue in C, Java). These are really disguised forms of goto’s, but are sufficiently well defined and predictable (and useful) to have found their place in the language. Likewise, the early return from a procedure is also a thinly disguised goto, but seems palatable enough. These constructs do not detract from the overall structure of a program and may in fact enhance it.

An important form of structuring a program as well as control flow is the procedure abstraction. While iteration loops allow reentrant code in the sense that it is iteratively re-executed, the procedure allows a reentrant block where the reentry can occur from many places. And because of the early return, more elaborate and useful control flow is possible. When we talk about data, we will focus more on the context of the procedure block, but for the procedure the block serves a significant purpose just for code structure and control flow. Procedure blocks also give a great advantage to the readability of code by encapsulating patterns of statements. The procedural abstraction is so important in imperative languages that this class of languages is sometimes referred to as the procedural languages (not to be confused with functional languages). Procedures also allow imperative languages to offer recursion, which can be an elegant control flow solution to some problems. Recursion is when a procedure contains a call to the same procedure. On the surface this would appear to be a case where we would never achieve termination. However, the paradigm here is one of reduction, where the logic of the procedure performs some computation involving a call to the same procedure in a simpler case. The base case (the simplest) does not involve the procedure call, so results in termination.
Scope and Blocks. As noted, one of the main concepts in an imperative language is the control flow, since the focus is on the execution of instructions and these control flow constructs may apply to blocks of statements rather than single statements. Languages use various syntax to indicate these blocks of control flow – curly braces in C/C++ and Java, BEGIN/END in Pascal, etc.. In the shell scripting language, blocks for each control flow construct have a delineator for the end of the block, e.g., fi, done, esac. Blocks are also used as the syntax of function definitions in C/C++ and Java – they essentially define named reentrant chunks of code, i.e., procedures.