Main topics of the week:

- Control Flow Constructs
- Sequencing and expression evaluation
- Procedural Abstraction

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 goto. 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 \( a+b\times c \) groups as \( a+(b\times c) \). 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 \( a+b\times c \), we tend to think of evaluation order (the \( b\times c \) is done first), but all that the language specification says is that \( b\times c \) 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.

**Procedures**

Procedures are the basic building blocks of imperative languages. Although code can be organized into blocks affected by control flow, the procedure allows a name and environment to be given to a block. Not only does this mean that the code in the block can be reused, but it can also be encapsulated so that the detailed implementation is hidden from a sequential reading of the code. That is, the procedure permits the code to be modularized, which is the real advantage of structured programming. The program can be viewed at a high level as mostly a sequence of procedure calls. Each of these procedures, in turn, can be viewed as a sequence of procedure calls, until we eventually decompose to a sequence of the lowest level of statements. Procedures also allow common implementations to be hidden in libraries that can be reused in a way that appears to extend the capabilities of the language.

Formally, **functions** refer to blocks of code that return a value so may be thought of as extending the operations of a language, and **procedures** as blocks of code that group statements together so effectively extend the language’s statements to more complex super statements. Some languages make this distinction and others do not. (Notably, C has only functions – procedures are just functions that do not return a value.) For the most part, we’ll use these terms interchangeably, essentially keeping the C model in mind. Some languages require a keyword to indicate a procedure, and others use syntax. C uses the syntax of parentheses to indicate function calls or function definitions.

**Procedure Characteristics.** Procedures are defined by their **name**, **formal parameters**, and **result type.** Of course, the actual code of the procedure is the procedure body. From a compiler’s point of view, the three characteristics are enough for static analysis. Knowing the parameters determines how to arrange a call to the procedure. Knowing the return type determines how the result will be handled. Procedures can of course have side effects in an imperative language, and that is often the point of using them. Pure functions may appear to be used in an identical way in a pure functional language, but in the pure functional languages there are no side effects and the function call is purely for the value calculated. Analysis of program source in an imperative language is more difficult since the side effects mean that the actions in the procedure could be different depending on when it is called, even if it is called with identical values. (A case in point is a C function using a static variable for the exact purpose of causing it to behave differently on the first call.)
Procedure Calling Abstractions. A procedure executes in an environment that is mostly determined by the parameters passed to the procedure (also called function arguments). There are three primary ways that parameters can be passed:

Call-by-value means that the procedure is given a copy of the values that appear on the function call. That is, the formal placeholder parameter in the function body takes on the value of the actual parameter in the function call. This provides a degree of separation between the caller and the function code in the sense that the function only works with a copy of the caller’s values, not directly with the variables in the calling code. This protection is the main advantage of call-by-value – it allows more implementation hiding and leads to fewer surprises. Assignments to the procedure’s formal parameters (if they are allowed) will not have any effect on the caller’s parameter variables. However, there is some cost associated with call-by-value, namely the cycles and memory required to make copies of values. For simple data types, this cost is usually fairly trivial (unless we are dealing with a simple function called so many times that this cost adds up). For larger user defined data types, the cost of the call-by-value copy could be prohibitive. (And when we talk about constructors in C++, the cost there must be considered.) This is the calling convention used in C, C++, and Java. However, since all objects in Java (non-primitive types) are references in Java, one effectively gets call-by-reference for objects passed to methods.

Call-by-reference means that the procedure has access to the actual parameter on the call. Although the name of the formal used in the procedure need not be the same name as used on the call, there is no separate copy created. In a compiled language, the compiler must arrange for appropriate linkage so that the procedure code has the desired access. This form of calling convention opens the door to the procedure having side effects upon the caller’s parameters, and that is often the design intent. When that is not the design, of course, the programmer can be in for surprises. Call-by-reference becomes a way for the procedure to provide return values. The caller may be invoking the procedure in order to have it fill in values in the caller’s variables. Some languages, like Pascal, use extra keywords to indicate call-by-reference instead of the default call-by-value. As noted, Java objects are always references, so for objects, Java effectively has call-by-reference. In C, pointers are often used to obtain call-by-reference. In this case, the value passed is a pointer, which, with appropriate de-referencing, allows the procedure to effectively implement call-by-reference.

Call-by-value-result is kind of in between call-by-value and call-by-reference. That is, the procedure has the value of the parameter, just like in call-by-value, but at the end of the procedure, the current value is copied back over the caller’s variable. So it is like call-by-reference in that the procedure can change the caller’s variable, but this doesn’t happen until the end of the procedure so it is like call-by-value up to that point.

Call-by-name means that the procedure actually gets the name of the formal parameter used in the procedure call. In particular, the argument is unevaluated. It is similar to call-by-reference except that the argument is not evaluated and the whole argument is somehow passed, along with code and an environment so that it can be evaluated in the procedure. The compiler/interpreter must also arrange that any naming conflicts are resolved by renaming variables in the procedure that might conflict. In a language with side effects, control flow is difficult to follow, with possibly disastrous
results. However, macro expansion in the C preprocessor is actually an example of simple call-by-name without the renaming.

Call-by-need is essentially call-by-name with lazy evaluation. That is, in the absence of side effects, the evaluation done where the value is needed in the procedure would return the same result. Call-by-need thus remembers the value so it can be reused, avoiding the overhead of re-computation for repeated uses of the variable.

Procedure activation frames. When a procedure is called, the compiler/interpreter must arrange for everything necessary to handle the call. In particular, control flow must be handled so that the program execution continues at the right place when the procedure completes; the parameters must be passed to the procedure code according to the language’s calling conventions, and the return value of the procedure must be made available to the caller. The mechanism for dealing with all of this information is called a procedure activation frame – it is a control block for a specific call to a procedure. Activation frames are created and discarded dynamically according to the dynamic execution of the program. An activation frame (also just called the stack frame) for C/C++/Java might look something like this:

The control link keeps track of where control goes when the function returns, and the access link keeps track of the chain of stack frames that reflect nesting of function calls. Note that the stack has space for the function arguments – we’ll look at examples of the calling conventions later. The stack frame also has space for the local variables, also called automatic variables in C since the space for them is automatically allocated by the creation of a stack frame. This is just an abstraction of what is on the stack frame – for a particular language and a particular compiler on a particular hardware platform, the details will reflect the use of hardware registers and other aspects of the platform.
Now let’s consider the stack frames for a specific code example under different calling conventions. Suppose we have the code:

```c
int bar(int a, int b) {
    int c = a * b;
    return c;
}

int foo(int a, int b) {
    int c = a-b;
    int d = a+b;
    return bar(c,d) / b;
}

void main() {
    int a = 7, b = 13;
    int c = foo(b, a);
}
```

At the point where the function `bar` is about to return, the stack frames would look like:

<table>
<thead>
<tr>
<th>function</th>
<th>local var</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>c</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>return</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>access link</td>
<td>foo</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>function</th>
<th>local var</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>d</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>return</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>access link</td>
<td>main</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>function</th>
<th>local var</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>c</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>parameters</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>return</td>
<td>void</td>
</tr>
<tr>
<td></td>
<td>access link</td>
<td>–</td>
</tr>
</tbody>
</table>

Notice the local variables in each function’s stack frame. The return values are indicated as the stack is unwound and the functions return. This stack picture shows what happens under a call-by-value parameter passing scheme.

Suppose that we had the following code:

```c
void foo(int a, int b) {
    int tempa = a;
    int tempb = b;
}
```
a = tempa-tempb;
b = tempa+tempb;
}

void main() {
    int a = 7, b = 13;
    int c;
    foo(b, a);
    c = a*b;
}

And parameter passing was still call-by-value. The picture would look like:

<table>
<thead>
<tr>
<th></th>
<th>before foo</th>
<th>during foo</th>
<th>after foo</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>local var tempb</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>local var tempa</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parameter b</td>
<td>7</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>parameter a</td>
<td>13</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>return value</td>
<td>none</td>
<td></td>
<td></td>
</tr>
<tr>
<td>access link</td>
<td>main</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>local var c</td>
<td>91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>local var b</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>local var a</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>parameters</td>
<td>none</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return value</td>
<td>void</td>
<td></td>
<td></td>
</tr>
<tr>
<td>access link</td>
<td></td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

Notice that under call-by-value, the stack frame for foo has its own parameter values that are copies of the values from main.

Consider the same code, but this time with parameter passing as call-by-reference. Then the picture would look like:
Note that there is only one storage area for the variables `a` and `b`, that is, `foo` does not have separate storage, but refers to the variables `a` and `b` in `main` (and note that `b` in `foo` refers to `a` in `main` and `a` in `foo` refers to `b` in `main`).