Main topics of the week:
- Variables
- Type Checking
- Scope
- Procedural Abstraction

Variables
Although imperative languages put the focus on actions as reflected in the control flow, programs would not be very interesting without data. The data in a program is accessed through symbols known as variables. Particularly in an imperative language where assignment allows data values to be changed, the term variable is appropriate. Variables can be characterized by the properties: name, address, type, and value.

The name of a variable is simply how we refer to it in the program source. Each language has rules as to the permissible names that may be used as identifiers of variables, e.g., letters and digits, can’t begin with a digit, etc. The syntax of the language specifies these rules.

The address of the variable is its location in memory. Some languages such as C/C++ allow the programmer to know the address of a variable, and some languages do not provide any mechanism to get at a variable’s address – the mapping of variables to addresses is kept completely hidden from the programmer. Whether the address can be seen or not does not affect the fact that the variable is associated with some storage in memory and it is often useful to think about that storage even if we cannot gain direct access to it. When two variables use the same address, we say that the variable names are aliases. In this case, since the same location can be referred to by different names, actions that change the value could be through either of the names. This certainly complicates program analysis, but can be for certain programming paradigms. Although Java does not allow addresses to be seen, all non-primitive objects in Java are references and so aliases can occur there. In C/C++, where there is an address operator and pointer types, aliases are directly available.

The type of a variable characterizes the use and interpretation of the data stored. Most languages define some basic data types such as integer, character, floating point, and maybe Boolean. A type specifies a data size, interpretation of the bits and value ranges, and valid operations that can be performed with the type. Some languages may also define a primitive type of string. These primitive types are built in to the language. In C and Java, we have these basic types and keywords to declare them.

In addition to primitive types, languages may provide derived types. In C, there are derived types of pointers (for every variable type, we can have a variable which is a pointer to that type). C also permits arrays of any type. Unfortunately, the treatment of arrays in C is fairly simplistic. Arrays are essentially used to refer to contiguous chunks of memory, with the language aiding in viewing the memory as a sequence of objects. However, C provides little support beyond this. An array has no inherent size – in a sense the compiler forgets about the size as soon as the space is allocated. This means there can be no built in bounds checking or understanding of whether two array objects are the same size. It also means arrays cannot be easily passed by value, although there are some tricks to get around this. In addition to derived types, languages often provide user defined types. This is basically a mechanism for grouping data together. We’ll see in
object oriented programming that user defined data there gets all the focus. In imperative languages, the aggregate data types give structure to the data and allow the user to view the groups of data as a single **record** type.

Finally, a variable has a **value**. The value is just the contents of the memory associated with the variable, interpreted properly according to its type. The way that we usually think about a variable is in terms of its value, typically abstracting away the idea of the physical storage as a collection of bits and viewing the variable as an integer, or a floating point number, or as a record.

**Lvalue versus Rvalue.** We have seen that an important characteristic of imperative languages is **assignment** – the ability to change a value of a variable in memory. The assignment expression in C is a binary operator ‘=’, using infix notation, with the left hand side being the object to be changed and the right hand side being the new value. The semantics of an assignment construct thus make an important distinction between the operands. In particular, the left operand must be an “assignable” object – typically a variable name, but more generally a reference to a memory location that is to be changed. The right operand on the other hand is just a value, again commonly a variable, but more generally a value obtained as the evaluation of a complicated expression. This distinction is where we get the terms **lvalue** and **rvalue**. For example, the C compiler may produce an error message like “invalid lvalue in assignment” when it sees a statement like i+j = i; This is because syntactically, the object to be assigned to is i+j, but that expression does not refer to a changeable memory location. Certainly any variable (non-constant) could appear as an lvalue, but there are other less obvious lvalues. The contents of a pointer variable, an offset into an array, or even a function return value de-referenced to a member of a structure. That is, the left operand of an assignment could be any expression, as long as that expression evaluates to a memory location that can be assigned to. Although these rules are consistent, they can make static analysis of an imperative program very hard or impossible if we want to determine everything that can change by a given assignment. Also, a simple variable may appear as either a left or right operand in an assignment; in this case, it is viewed in one context as an lvalue and another context as an rvalue. Actually, when an lvalue is required, we use the location of the variable, and when an rvalue is required, we use the value of the variable. This could be confusing since the variable as it appears in the statement looks the same on either side – its context is what determines the interpretation. (We’ll see in ML that explicit notation is required to make this distinction.)

**Type Checking.** As soon as we introduce types into a language we come to the issue of how much attention the compiler or interpreter should pay to types. Different languages have different degrees of type sensitivity: we refer to languages from “not typed”, to typed, to weakly typed, to strongly typed. What this means is how much the language enforces type agreement among operands in expressions. C is a typed language, but certainly not strongly typed. C often defaults a missing type to integer, and implements no type checking on function arguments (even worse, it doesn’t check function argument counts – a form of checking the type of the function). C++ and Java are much more strict about function arguments and less tolerant of implicit typing. However, all of these languages allow some mixing of types, particularly for numerical
Types where the mixing does not result in any loss of information. For example, in
expressions involving integers and floating point numbers, the integers are implicitly
converted to floating point for the operation: this is called **coercion**. Compilers typically
warn or prohibit assignment of float to int since this would result in loss of information.
These conversions are a great convenience to the programmer in most cases, but can
occasionally produce unexpected results or compiler/platform dependencies. (We’ll see
later in ML how annoyingly explicit we must be about types, but there are at least no
surprises.)

In a typed language, one of the most important jobs for the compiler/interpreter is the
type checking. The compiler/interpreter “thinks” of all of the objects in the program in
terms of their type. The language will specify what types are allowed to be combined in
expressions and whether any type coercion will take place. Type checking in a compiled
language is a static analysis that is carried out at compile time. It is based on information
about types in the program source (e.g., declarations). Compiled languages like C can
only implement strict type checking, so it may be that not all type errors are caught if the
program uses dynamic typing. Formally, there is no dynamic typing in C, but programs
may use type casts at run time. What is really going on with a type cast in this setting is
that the type is actually unknown or known to be incorrect from the compiler’s static
analysis point of view, and the programmer is getting the compiler to accept on faith the
programmer’s knowledge about the types and to possibly do some conversion of bits to
the indicated type. By the very nature of the type cast, errors cannot be caught by the
compiler.

When dealing with types, the issue arises as to what are **equivalent types**. Certainly
two variables declared together should be of the same type. What if the declarations are
separate, but use the same syntax and type names? In this case, we would say they are
equivalent. We could formally define **structural equivalence** as types being structurally
equivalent to themselves (the trivial case), or formed by applying the same type
constructor, or formed by a type definition of one to the other (e.g., a typedef in C). Thus,
we may have different type names, but if they are essentially synonyms, they are the
same type. Equivalence could be more restrictive, where the names of the types must be
identical. In C and C++, structural equivalence is used, except for user defined structures,
which must be name equivalent.

Some languages (like Pascal and C/C++) allow **pointer** types. This is a way of
allowing direct access to memory contents. Operators are provided to allow the language
to specify that the memory address of a variable be calculated and used in an expression.
Similarly, another operator allows the value stored at a location to be obtained. These
operations mean that a memory location’s contents can be manipulated in an expression
without ever having seeing the name of the variable that “owns” the memory. The power
of pointers is that the level of indirection allows more dynamic handling of memory
contents. Also, as we’ll see in procedure parameter passing, pointers allow a call by value
language to implement call by reference. Pointers can also allow efficient handling of
dynamically sized arrays of objects. The downside is that the compiler cannot statically
analyze what will happen at runtime.

C++ allows a **const** qualifier to be used in any variable declaration (and Java permits
the **final** qualifier). The use of such a qualifier extends the type information available to
the compiler. For a simple variable declaration that is marked constant, the compiler now knows that it is impermissible to modify the value of the variable. This could occur through direct assignment of course, but also less obviously in an array or more complicated expression. In C++, where pointers are possible, the compiler has to also be on the lookout for indirect assignments that could happen through the use of pointers. Although this might seem an impossible task for the compiler, it actually is possible for the compiler to keep track of when the value is modified as long as it includes the constant aspect as part of the type and carefully retains that constantness when it sees expressions using the address of the variable. For this reason, an array can be declared constant, and a pointer to an element of the array can be incremented or decremented and the constantness retained. However, if the pointer is incremented beyond the bounds of the array, then even though the constantness is retained, it is meaningless, as the pointer now refers to memory that may or may not be of the original array element’s type. The appropriate use of const in C++ allows the programmer to get the advantages of pointers in terms of performance and syntax efficiency, but avoid many of the common problems of unexpected side effects. This construct permits the expression of more design intent in the language itself.

Scope. The variables in a language are the identifiers we use to refer to stored data values. We have seen that blocks are used for control flow in an imperative language, but blocks also serve another very important purpose. They define a locale – an area of the code that is like a small environment within the larger program. Thus, variable declarations may be placed within a block, and the block determines the scope of the variable. The scope of the variable refers to where in the code the variable name may be used. It is a property of static analysis of the program. The basic scope rule in C/C++/Java is that a variable may be used any place in the code from the point where it is declared to the end of the block in which it is declared. C and C++ allow variables to have global scope (Java does not). This means that the variable can be referred to at any place in the code after its point of declaration. (Most languages’ scope rules have a sense of direction – forward toward the end of the file from the point of declaration.) A variable has local scope, also said to be a local variable, if it is declared within a block. We don’t want to confuse the scope of a variable with the storage for the variable. The scope simply refers to where the symbol can legally appear – it is part of the semantic analysis of the code since knowing if a variable is legal means finding its declaration in an appropriate place, and this is beyond the ability of a grammar to express. When scope is determined in this way by statically analyzing the source code without any notion of what the thread of control will look like for a particular execution, it is called static scope (or lexical scope). It is possible for a language to implement dynamic scope (Lisp does this), but we’ll see later that it can be unintuitive, and is not typically done. However, the dynamic binding that is done with virtual functions in C++ and interfaces in Java does capture some of the idea of dynamic scope in a predictable and useful way.

One issue that comes up with the notion of local variables is what the language should specify if a variable is declared a second time in the same scope. In C/C++/Java, this is considered to be an error. However, if a variable is re-declared in a nested scope, that is acceptable in these languages. But the next question is about the semantics of such a re-declaration. First, it must be of the same type, and second, the compiler arranges for
a new instance of a variable, and all subsequent references will refer to the variable instance in the closest block. In effect, this will hide the earlier declarations.

The following C++ code fragment shows these properties of variables.

```cpp
int i;
main() {
    int i;
    . . .
    if (i > 0) {
        int i;
        for (int i = 0; i < 10; ++i) {
            int j = i;
            . . .
        }
        int j = i;
        . . .
    } int j = i;
    . . .
}
}
int j = i;
while (j-- > 0) {
    int i;
    . . .
} . . .
j = i;
```

Scope often seems related to, but should not be confused with **storage**. The storage determines the **lifetime** of the variable, i.e., when during the program execution the storage exists and is valid. Remember that the scope only refers to where it is legal in the code to mention the variable’s name. There are three main types of storage that may be used for a variable:

**Static storage** is space created for the variable prior to the start of execution. In C/C++ and Java, the keyword **static** is used to modify a declaration to indicate this type of storage. Static variables are around for the entire execution life of a program. For class member data, static means there is only one instance of the data, regardless of how many instances of the class object exist (even zero). In a sense, static class data is global to the class objects. The way to think of static variables is that the compiler arranges for the execution image to have some space for these variables prior to beginning execution.

**Stack storage**, or **automatic storage** is used for local variables. C/C++ actually have a keyword **auto** to indicate this storage, but since it is the default, you almost never see it
used. Storage for such variables is automatically allocated as part of the stack. It is usually space in an activation frame that is placed on the stack for a block (procedure or just control block) and this is placed on the stack at execution time at the point in time where the block or procedure is entered (becomes active). At exit from the block or procedure, the activation frame is taken off the stack. So the lifetime of automatic variables is similar to their scope, except that there may be holes in the scope where the variable is hidden by a nested scope, but the variable’s storage is still around.

**Heap storage** or dynamic storage is storage explicitly requested during the execution of a program according to instructions coded by the programmer. The storage is in an area of memory used by the heap manager, which allocates, frees, and coalesces chunks of memory. This storage is obtained in C code by a call to malloc, which is a standard C library routine. In C++ code, the C++ operator new is used to allocate space for objects of a given type (and automatically calls the object’s constructor, too). In Java, the operator new is used to create an object as well. Objects on the heap have a lifetime that is automatically determined in Java to be as long as they are needed, i.e., as long as they are being referenced. When an object can no longer be referenced, the Java VM will make it available for garbage collection, which is periodically run to recover space that is no longer needed. In C/C++, objects on heap must be explicitly freed. C++ makes this easier with destructors that get called when an object goes out of scope, but it has no garbage collection for everything. Rather, the paradigm in C++ is that you can implement your own garbage collection (as well as heap allocation) that is tailored to the needs of the class. This allows for customization in the event that the one size fits all allocation/de-allocation strategy of Java is not suitable for every application’s needs.

**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);
}
```

<table>
<thead>
<tr>
<th>local variables (automatic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>function arguments</td>
</tr>
<tr>
<td>return value</td>
</tr>
<tr>
<td>control link</td>
</tr>
<tr>
<td>access link</td>
</tr>
</tbody>
</table>

Stack
At the point where the function \texttt{bar} is about to return, the stack frames would look like:

\begin{verbatim}
<table>
<thead>
<tr>
<th>Function</th>
<th>Local Var</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>\texttt{c}</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>\texttt{b}</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>\texttt{a}</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>120</td>
</tr>
<tr>
<td>foo</td>
<td>\texttt{d}</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>\texttt{c}</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>\texttt{b}</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>\texttt{a}</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>17</td>
</tr>
<tr>
<td>main</td>
<td>\texttt{c}</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>\texttt{b}</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>\texttt{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</td>
<td>-</td>
</tr>
</tbody>
</table>
\end{verbatim}

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:
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).