In this lab, we’ll consider the effect of a variable declaration in C, the significance of having to declare the type of each variable, and the construction of arithmetic expressions.

Exercises

Here are the pairs for today’s lab. Drivers are on the left.

- Sophie Gaschott and Tanner Tufto
- Wyatt Heritage and Erhaan Ahmad
- Colin Greenman and Zachary Susag
- Dennis Chan and Lilly Webster
- Mattori Birnbaum and Lex Martin
- Eli Salm and Josh Lavin
- Saung Thuya and Jae Hong Shin
- Cory McCartan and Jong Hoon Bae
- Yuyin Sun and Anna Blindermann
- Gemma Nash and Ella Nicolson
- Tyler Williams and Ying Zhang
- Sanjay Sudhir and Faizaan Ali

Variable Declarations

A variable declaration advises the compiler to reserve part of the computer’s memory for the single purpose of storing the variable’s value. If there is no initializer in the declaration — no equal sign, no expression to supply the value to be stored — the memory will be reserved anyway, and whatever pattern of zeroes and ones happens to be left over from previous uses of that storage will be interpreted as the value of the variable, even if it makes no sense in the context of the current program. This is a good argument for including initializers in all of your variable declarations unless you are certain — that is, unless you can prove — that your program will assign a new value to the variable before it tries to evaluate any other kind of expression using that variable.

A variable declaration also tells the compiler how much memory to reserve for the value of a variable. The zeroes and ones that are stored in a computer’s memory are called binary digits, or bits for short. You can think of each bit in a computer’s memory as a two-position switch, like a light switch, that remains in the same position, on or off, until someone comes along and toggles it. The switches in a computer’s memory retain their state indefinitely until they are toggled (or the power to the computer fails somehow).

Eight of these switches that are adjacent in memory and can be treated as a group make up a byte. The overall size of a memory is usually measured in bytes: A thousand bytes make a kilobyte, a thousand kilobytes make a megabyte, a thousand megabytes make a gigabyte, a thousand gigabytes make a terabyte, a thousand terabytes make a petabyte, and a thousand petabytes make an exabyte. Names for still larger quantities of memory have already been agreed upon, but we don’t need them quite yet to describe anything in the real world.

The type of a variable tells the compiler how many bytes of memory the program will need to store a value. On MathLAN workstations, it takes four bytes (thirty-two bits) to store a value of type int, and the same number of bytes to store a float. Only one byte is
needed for a value of type \texttt{char}, because C's \texttt{char} type does not include as many different characters as Scheme's does. (For instance, a Greek letter, such as \textalpha, is a character in Scheme but not in C.)

Each value of C's \texttt{char} type corresponds to a different setting of the eight one-bit switches that make up the byte. With eight switches, each of which can be set independently in either of two ways (on or off) the number of different switch patterns is $2^8$, which is 256. That's how many different characters C can express directly.

Since the 1999 revision of the C standard, C has also had a Boolean data type, called \texttt{bool}, containing the two values \texttt{true} and \texttt{false}. A \texttt{bool} really only needs a single bit, but a C compiler uses a full byte to hold a Boolean value because on most modern computers it is difficult for the processor to deal directly with single bits. Processors prefer to work with data at least eight bits at a time, and it slows them down when they have to isolate single bits.

Besides \texttt{int}, C supports the data types \texttt{short} and \texttt{long}, for occasions when the programmer chooses to work with whole numbers in ranges that are smaller or larger than the range that \texttt{int} provides. On the workstations we use, a \texttt{short} value occupies two bytes and must lie in the range from $-32768$ up to, but not including, $+32768$. A \texttt{long} value occupies eight bytes and must lie in the range from $-9223372036854775808$ up to, but not including, $+9223372036854775808$.

Finally, C supports the \texttt{double} type for floating-point numbers at higher precision than \texttt{float} provides. Each \texttt{double} value occupies eight bytes, and that is the amount of storage that the compiler reserves when you declare a variable of type \texttt{double}.

As you read through the variable declarations at the beginning of a function, to get to know the "cast of characters" for that function, it may be helpful to think of the storage locations they reserve as boxes, with fixed sizes, in which values are placed through initialization and assignment. Neither initialization nor assignment will work correctly if the expression on the right-hand side of the equals sign has a value that won't fit into the number of bytes of storage provided. Your program is unlikely to crash, but it will deliver incorrect answers, which is even worse.

Exercise 000: Suppose that a function contains declarations for three variables of type \texttt{int}, two of type \texttt{double}, and one of type \texttt{int}. How many bytes of storage does these variables occupy altogether?

Exercise 001: Look at the program that you wrote for the module 000 assignment. How many variable declarations does your \texttt{main} function contain? What is the total amount of storage that those variables occupy?

In C, binary operations on numerical values, such as addition, subtraction, and multiplication, all require that the two operands belong to the same type. If one of the operands in an arithmetic expression is of type \texttt{short}, it is "promoted" to type \texttt{int} (without changing its numerical value) in order to enable the operation to go forward. If one of the operands is of type \texttt{long}, the other operand is similarly extended to be of type \texttt{long} before the arithmetic is done.

Similarly with floating-point numbers: The actual operations are defined for \texttt{double} arguments, and any \texttt{float} values are promoted to \texttt{double} before the arithmetic is carried out.

Exercise 010: Compute the value of the expression $1000000 - 999997 * 100000$. Is this result in the \texttt{int} range? Can it stored into an \texttt{int} variable?

You can test this experimentally: Write a C program that carries out that computation, assigns the result to a variable of type \texttt{int}, and then prints out the value of that
variable.

Exercise 011: Does it help to change the type of the result variable to `long`? (Note that, in the call to `printf`, the placeholder for a long variable should be `%ld` rather than just `%d`.

The Scheme procedures `char->integer` and `integer->char`, which convert characters to and from their numeric equivalents, have no analogues in C. Instead, C allows programmers to use character constants and variables directly in arithmetic expressions. So, for instance, `'A' + 8` is a legitimate expression that adds 5 to the numeric equivalent of a capital A (which is 65). So the value of the expression is 73.

If you store the numerical result, 73, back into a variable of type `char`, it will become a character again — the character that is five positions down from `'A'` in the alphabetical ordering of the characters, namely `'I'`.

Exercise 100 Write a program that “subtracts” a capital letter `'J'` from a lower-case `'j'` and prints out an integer that tells how far apart the codes for these letters are. Is this distance the same for every capital letter and its lower-case equivalent?

(Hint: Typing `man ascii` in a terminal window brings up a table of all of the characters in the American Standard Code for Information Interchange and their numerical values. C takes its numerical codes for the ASCII characters from this standard.)

Arithmetic Operations

C supports the four basic operations on numbers: addition, symbolized by `+`; subtraction (`-`); multiplication (`*`); and division.

For floating-point numbers, division is symbolized by `/` and yields another floating-point number. For integer types, the story is more complicated. The operation symbolized by `/` is then the one that Scheme calls `quotient` or `div`: It discards any remainder from the result. Thus the value of, say, the expression `7 / 3` is not a rational number with a fractional part — instead, it’s simply 2, with the leftover remainder from the division being ignored.

There’s a separate operator, `%` that corresponds to Scheme’s `remainder` or `mod` procedure. It carries out a division, but keeps only the `remainder`, throwing away the `quotient`. So, for instance, the value of the expression `38 % 9` is 2, because 9 goes into 38 four times with 2 left over. This operator may not be applied to values of type `float` or `double`.

Exercise 101: Write a C function `evenly_divides` that takes two `int` arguments and determines whether the first of them evenly divides the second (that is, divides it with nothing left over — a remainder of 0). The function should return a Boolean value, `true` or `false`.

You may find it easier to write this function if you have done some of the reading and already know about `if`-statements. But actually no `if`-statement is needed here.

Exercise 110: Write and test a C function that calculates the harmonic mean \( \frac{2ab}{a+b} \) of any two positive real numbers `a` and `b`.

If a program needs to know how many bytes a value of a particular type occupies, it can compute that information during the execution of the program with an operator called `sizeof`. If you place the name of a type, enclosed in parentheses, after the keyword `sizeof`, you get an expression whose value is the number of bytes for that type. So, for instance, on MathLAN systems, the value of `sizeof(double)` is 8.

Exercise 111: Write a C program to confirm that a value of the `bool` data type occupies one byte of storage.