Mental Models of Computation

To verify how our programs work, we can compile them, run the resulting executables, and check that they behave as expected. Testing is a powerful, direct method of program verification, but it can be insufficient or infeasible for a variety of reasons:

- We are in the middle of development and our program is not yet in a state to run.
- There is a very large number of possible inputs to our program (perhaps an infinite amount) that we cannot possibly cover with a finite number of runs of our program.
- It is too complicated or costly to run our program, e.g., because it is attached to a physical device or is part of a much larger, complex system.

For these reasons, we would like alternative methods of verifying our programs that we can use in tandem with testing.

To this end, we introduce mental models of computation for C. A mental model of computation is simply a model of execution of our computer programs we can perform in our head or on paper. This allows us to quickly verify our programs—usually small pieces of it like an individual function—by hand. Furthermore, our mental models of computation closely aligns with the semantics of the C programming language, so understanding these models will, in turn, help us deeply understand C. Finally, our mental models of computation help us debug our code by giving us the ability to crisply describe what is going wrong in our programs.

Expressions and Statements

Before we introduce the two models of computation that we will use in this course, we first review the two major syntactic categories in C: expressions and statements.

Expressions

Expressions in a programming language generalize arithmetic expressions found in mathematics.

Definition 1. An expression is a language construct that evaluates to a value.

For example, the expression $3 + 5 \times 8$ evaluates to 43. We know this via operator precedence which tells us that multiplication must be carried out before addition. An expression is typically composed of sub-expressions, i.e., smaller expressions. In mathematics, we usually call these sub-expressions operands. 5 * 8, 3, 5, and 8 are all sub-expressions of the expression above. 43 is both an expression but also a value.
Definition 2. A value is an expression that takes no further steps of evaluation.

An example of a floating-point value is 3.5. The string "hello world" is also a value.

Finally, types categorize sets of values as well as expressions.

Definition 3. A type classifies a set of values. By extension a type also classifies expressions (by the type of value they produce).

The expression $3 + 5 \times 8$ as well as its resulting value 43 have type int. Note that it is not always the case that an expression must be entirely composed of the same type. For example, comparison expressions such as $x \times 2 < 3 + 5$ have int sub-expressions but the final computed value is a boolean.

Mathematics defines a series of operators and rules of precedence over them: $+, -, *, /$. Programming languages typically add many more operators that operate over different types, e.g., the mod operator $\%$, boolean “and” and “or” operators $\&\&$ and $||$, and boolean negation $!$. Boolean negation is an example of a unary operator, an operator that only takes a single argument. Function calls that return values, e.g., $\sqrt{2}$, are also operators where the arguments to the function are the operands.

Statements

In contrast to expressions which perform computation via repeated evaluation. Statements do not produce any values that can be used like an expression.

Definition 4. A statement is a program fragment that does not evaluate a value but instead produces one or more side-effects.

We cannot, for example, instead a variable declaration statement, e.g., int x = 5, into an addition expression because the statement does not produce a value. Instead, it has the effect of creating a new variable x and assigning it the value 5. Likewise, printf("hello world") also does not produce any values\(^1\) but has the effect of printing the text "hello world" to the console.

While expressions perform the actual computation in our program, effects are typically how we interact with the “outside world” with respect to our program. Examples of effects include:

- Variable declaration.
- Variable assignment.
- Printing to the console.
- Reading from or writing to a file.
- Receiving data from the network.

The various statements in our programming language either perform these effects directly or manage the execution of sequences of statements in our program.

\(^1\)That’s a bit of a lie. printf actually returns the number of characters printed to the console, but that value is very rarely used in practice, so we’ll pretend its return type is void.
The Substitutive Model

The first model we discuss concerns expressions. An expression evaluates via repeated substitution, and our model mirrors this process exactly. For example, here is how the expression $1 * 3 + 4 / 2 - 1$ evaluates step-by-step:

\[
\begin{align*}
1 * 3 + 4 / 2 - 1 & \quad \text{[ evaluated 1 * 3 ]} \\
\rightarrow & \quad 3 + 4 / 2 - 1 \quad \text{[ evaluated 4 / 2 ]} \\
\rightarrow & \quad 3 + 2 - 1 \quad \text{[ evaluated 3 + 2 ]} \\
\rightarrow & \quad 5 - 1 \quad \text{[ evaluated 5 - 1 ]}
\end{align*}
\]

The process of evaluating an expression proceeds as follows:

1. Find the next sub-expression to evaluate via operator precedence.
2. Evaluate that sub-expression to a value.
3. Substitute that sub-expression with its resulting value.
4. Repeat the process until the overall expression is a value.

For arithmetic operations that we are familiar with, this process is straightforward. However, things become trickier when we add in more operators, e.g., how do we evaluate the expression $3 * 5 < 4 \; || \; 3 > 8 / 2 \; && \; 5 < 2$? It turns out that the arithmetic operations have lower precedence than the comparison operators which have lower precedence than the logical operators. Finally, logical-and ($\&\&$) has strictly higher precedence than logical-or ($||$), so we have:

\[
\begin{align*}
3 * 5 < 4 \; || \; 3 > 8 / 2 \; && \; 5 < 2 & \rightarrow & \quad \text{true} \; || \; \text{false} \\
\rightarrow & \quad \text{true} \; || \; \text{false} \\
\rightarrow & \quad \text{true}
\end{align*}
\]

Section 2.12 of K&R provides a complete table of operator precedence. Similar tables can also be found online.

Implicit Conversions and Casting

Recall that there are two broad categories of primitive types in C—integral and floating-point types. The difference within these categories is simply size. For example, a char takes up one byte of memory whereas an int usually takes up four. When expressions of different types mix, e.g., $3 + 3.5$, we must determine what is the output type to know what value the expression produces.

In general, C favors the type where no information will be lost. In the case of $3 + 3.5$, the two sensible results are $6$ of type int or $6.5$ of type double. In the first case, we lose precision—namely the fractional value $.5$—whereas in the second case, we obtain the exact answer. For this reason, C states the result of $3 + 3.5$ is a double rather than an int. Likewise, if I simply assign an int to a double variable, e.g., double $x = 3$, C automatically $3$ to the floating-point value $3.0$. These
are examples of implicit conversions where C automatically takes a value and converts it to the appropriate destination type.

In addition to implicit conversions, we can also force conversions to occur through the cast operator. The cast operator is a unary operator (i.e., it takes a single argument) which has the syntax:

\[(\texttt{<type>}) \ <\texttt{expr}>\]

where the \texttt{<type>} above is the target type of the cast and the \texttt{<expr>} is the expression whose result will be converted. For example, the expression \texttt{(double) 3} produces the value \texttt{3.0} of type \texttt{double}. Note that casts have higher precedence than the arithmetic operators, so that we need to parenthesize the expression appropriately if we wish to cast its overall value versus just a piece of it. For example:

\[(\texttt{int}) 3.5 + 8.2 \quad \texttt{// produces 11.2, the cast associates with 3.5}\]
\[(\texttt{int}) (3.5 + 8.2) \quad \texttt{// produces 11, the cast associates with the whole expression}\]

We can use our mental model of computation to explain how computations involving implicit conversions and casts operate. A common example of this is an averaging calculation, e.g., between \texttt{int} variables \(x\), \(y\), and \(z\):

\[
\frac{x + y + z}{3}
\]

If left alone, this will perform integer division in the final step, producing an \texttt{int}. This is unsuitable if we wanted to retain the fractional portion of the result. However, the following cast does not produce the desired result:

\[
\texttt{(double) ((x + y + z) / 3)}
\]

Because the cast occurs after the division has taken place. To remedy the situation, we need to introduce a \texttt{double} conversion before the division occurs. The following expressions all work identically:

\[
\frac{x + y + z}{3.0} \quad \frac{x + y + z}{\texttt{(double) 3}} \quad \texttt{(double) (x + y + z) / 3} \quad \frac{x + \texttt{(double) y + z}}{3}
\]

\section*{The Stack Model}

We use the substitutive model to describe how expressions evaluate, but we cannot use this model for statements. This is because side-effects are necessarily at odds with substitution. For example, consider the following sequence of statements:

```c
int x = 1;
printf("%d\n", x);
printf("%d\n", x+3);
```
For example, how do we handle a variable declaration statement using substitution? One thing we might try is substituting the value of the variable everywhere that variable occurs in the statements. For example, after executing the variable declaration, we would be left with the following statements to execute:

```c
printf("%d\n", 1);
printf("%d\n", 1+3);
```

This seems sensible. However, the approach begins to break down when we consider variable assignment in conjunction with declaration. For example, if we had the following statements instead:

```c
int x = 1;
printf("%d\n", x);
x = 5;
printf("%d\n", x+3);
```

we can’t simply substitute 1 for the occurrence of x in the second printf because of the intervening assignment to x. We can refine our approach to only substitute for x up until the point it is reassigned. However, in the presence of conditional statements, things get even more complicated:

```c
int x = 1;
// ...
if (y < 10) {
    x = 5;
}
printf("%d\n", x);
```

The assignment of 5 to x occurs only if we reach the conditional statement and y < 10. However, because there might be arbitrary code between the variable declaration and the conditional, we do not know at the point of the variable declaration whether we will enter the conditional and thus whether it is safe to substitute 1 for x in the printf below.

Rather than having to work out all these special cases, we instead use a different model of computation to model the execution of statements. This model, the stack model of computation, keeps track of:

1. The current statement being executed.
2. The stack of active stack frames which encompass the functions that are currently being executed along with the values of their local variables.
3. The output of the program

An Example

We illustrate the stack model of computation with an example. Consider the following C program:
void printNum(void) {
    int y = 5;
    printf("%d\n", y);
}

int main(void) {
    int x = 0;
    printNum();
    return 0;
}

Execution of this program starts in main. Thus, we record the state of our computation as follows:

Program Counter: 7

The Stack
======
--- main
     (empty)

The program counter is the current line we will execute. Because our program starts in main, we start execution of the program with the first statement of main at line seven. The stack records the active stack frames of the program. An stack frame records the name of the function call that is currently active along with all the local variables declared so far in that function along with their values. Initially, we do not have any local variables declared in main, so the stack frame corresponding to main is empty.

On line seven, we execute a variable declaration which adds the local variable to main’s stack frame. We thus update our state as follows:

Program Counter: 8

The Stack
======
--- main
     x [0]

On line eight, we execute a function call which causes execution of the program to jump to printNum. To record this, we add a new stack frame for printNum above main. The program counter is also changed to be the first line of printNum

Program Counter: 2

The Stack
======
--- printNum
     (empty)
--- main
  x [0]

We say that printNum is the currently active stack frame because it is the top-most frame on the stack. Execution of main, in contrast, is suspended pending the execution of the function above it on the stack, printNum. The stack obtains its name from this arrangement of stack frames where new stack frames are placed on top of the stack and we remove frames from the stack starting from the top. This is commonly referred to as first-in-last-out (FILO) behavior.

On line two, we execute another local variable declaration which allocates y within the currently active stack frame—printNum:

Program Counter: 3

The Stack
==========
--- printNum
  y [5]
--- main
  x [0]

On line three, we execute printf, passing in the value of y. The value passed in is the currently-recorded value of y within printNum’s stack frame, 5.

Program Counter: 4

The Stack Output
========== ======
--- printNum
  y [5]  5
--- main
  x [0]

Line four ends the printNum function, so we return from printNum. To do this, we pop the currently active stack frame which is printNum’s stack frame. Popping a stack frame means that the frame is removed from the stack, including its local variables. Execution of the program continues on the line after the call of printNum in main.

Program Counter: 9

The Stack Output
========== ======
--- main
  x [0]  5

The last line of main exits the function with a return statement, so execution of the program is complete!
Scoping and Variable Lookup

We say that the **scope** of a name, e.g., a variable or function, is the region of code in which it is valid to reference that name. For a variable declaration, we may reference a variable from its **point of declaration** to the close-curly braces that enclose the declaration. For example, the following code snippet outlines where the local variable `x` declared in `bar` is visible:

```c
void foo(void) {
    /* x is not visible here */
}

void bar(void) {
    /* x is not visible here */
    int x = 0;
    /* x is visible here, to the } below */
}
```

The scope of a variable in C is determined purely by the text of the source code. This is called **lexical scope**. The compiler determines whether all uses of names in our programs are legal and will reject any program (at compile time) that contains a reference to a name that is out of scope.

Assuming that a variable is visible, we determine the value of the variable by lookup in the stack during program execution. Because our stack frames allow us to discern which function calls are active, it is unambiguous which entry we look up. For example, consider the following code:

```c
void foo(void) {
    int x = 5;
    printf("%d\n", y);
}

int main(void) {
    int x = 0;
    foo();
    return 0;
}
```

Here, there are two declarations of `x`, one in `main` and the other in `foo`. Which one do we lookup on line three? Our mental model of computation at line three tells us!

**Program Counter:** 3

--- foo
x [5]
--- main
x [0]
foo is the active stack frame, so we look up $x$’s value in foo instead of main. Thus the printf produces 5 as expected.

In effect, local variable names, by definition, are scoped to their containing functions. This allows us to reuse local variable names in our functions without the need for coordinating names between all the functions that we have declared.

**Exercise (Stack Example)** Consider the following program:

```c
void foo(void) {
    int x = 5;
    /* Point A */
    foo();
    printf("%d\n", x);
}

int main(void) {
    foo();
    return 0;
}
```

Give the state of the computation (the program counter as well as the contents of the stack) at Point A outlined above. You may set the program counter to be one line after the comment.

Explain the behavior of this program using your model. What happens when this program executes? (Write your answers in a text file called stack.txt.)