# T01 Reading C Programs

School of Electrical and Computer Engineering  
Cornell University

**revision:** 2017-08-31-23-37

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Statements, Syntax, Semantics, State</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Variables, Operators, Expressions</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Variables</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Operators</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Expressions</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>Simple C Programs</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Name Binding</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>Functions</td>
<td>10</td>
</tr>
<tr>
<td>4.1</td>
<td>Function Definition</td>
<td>10</td>
</tr>
<tr>
<td>4.2</td>
<td>Function Call</td>
<td>11</td>
</tr>
<tr>
<td>4.3</td>
<td>The <code>printf</code> Function</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>Conditional Statements</td>
<td>14</td>
</tr>
<tr>
<td>5.1</td>
<td>Boolean Operators</td>
<td>14</td>
</tr>
<tr>
<td>5.2</td>
<td><code>if/else</code> Conditional Statements</td>
<td>16</td>
</tr>
</tbody>
</table>
5.3. switch/case Conditional Statements ........................................ 18

6 Iteration Statements ............................................................... 21
   6.1. while Loops ..................................................................... 21
   6.2. for Loops ....................................................................... 22

7 Syntactic Sugar .......................................................................... 24
Before you can learn to write, you must learn to read!
This is true for foreign languages and programming languages.

1. Statements, Syntax, Semantics, State

<table>
<thead>
<tr>
<th>Program</th>
<th>Statement</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence of statements</td>
<td>It is raining outside. Should I use an umbrella?</td>
<td>rain is water condensed from the atmosphere, outside means in the outdoors</td>
</tr>
<tr>
<td>Statement</td>
<td>Sentence</td>
<td>It is raining outside.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Sentence grammar</td>
<td>punctuation; “I” is a pronoun; “is” uses present tense</td>
</tr>
<tr>
<td>Semantics</td>
<td>Sentence meaning</td>
<td>remember that it is raining outside when considering umbrella</td>
</tr>
<tr>
<td>State</td>
<td>Memory of prior statements</td>
<td></td>
</tr>
</tbody>
</table>
An example “English” program

1. Create box named x.
2. Put value 3 into box named x.
3. Create box named y.
4. Put value 2 into box named y.
5. Create box named z.
6. Put x + y into box named z.

<table>
<thead>
<tr>
<th>stmt</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. Variables, Operators, Expressions

- A **variable** is a box (in the computer’s memory) which stores a value
- An **operator** is a symbol with special semantics to “operate” on variables and values
- An **expression** is a combination of variables, values, and operators which evaluates to a new value
2. Variables, Operators, Expressions

2.1. Variables

- A variable is a box (in the computer’s memory) which stores a value
- An identifier is used to name a variable
- A type specifies the kind of values that can be stored in a variable
- A variable declaration statement creates a new variable

2.2. Operators

- The assignment operator (=) “assigns” a new value to a variable
- An assignment statement combines the assignment operator with a left-hand side (LHS) and a right-hand side (RHS)
- The LHS specifies the variable to change
- The RHS specifies the new value

```plaintext
int my_variable;
my_variable = 42;
```

- A variable declaration statement and an assignment statement can be combined into a single initialization statement

```plaintext
int my_variable = 42;
```

- Other operators are provided for arithmetic functions such as addition (+), subtraction (-), multiplication (*), division (/), and modulus (%)
2.3. Expressions

- An **expression** is a combination of variables, values, and operators which evaluates to a new value

```
1  3 + 4
2  3 + 4 * 2 + 7
3  3 * 4 / 2 * 6
```

- **Operator precedence** is a set of rules describing in what order we should apply a sequence of operators in an expression

<table>
<thead>
<tr>
<th>Category</th>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplicative</td>
<td>* / %</td>
<td>left to right</td>
</tr>
<tr>
<td>Additive</td>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>Assignment</td>
<td>=</td>
<td>right to left</td>
</tr>
</tbody>
</table>

Be explicit – use parenthesis!
2.4. Simple C Programs

We can compose assignment and initialization statements which use variables, operators, and expressions to create a simple C program.

Translating our “English” program into a C program

1. \texttt{int } \texttt{x} = 3;
2. \texttt{int } \texttt{y} = 2;
3. \texttt{int } \texttt{z} = \texttt{x} + \texttt{y};

Draw a state diagram of this program

1. \texttt{int } \texttt{x} = 3;
2. \texttt{int } \texttt{y} = 2;
3. \texttt{int } \texttt{z} = \texttt{x} + \texttt{y} \times 5;
4. \texttt{y} = \texttt{x} + \texttt{y} \times \texttt{x} + \texttt{y};
3. Name Binding

- So far we have only had one variable with a given name

```
1  int  x = 1;
2  int  x = 2;
3  int  y = x;
```

- **Scope** of a variable is the region of code where it is accessible

- C allows using blocks to create new local scopes

- Can declare new variables that are only in scope (locally) in the block

- Can declare new variables in the local scope with same name as a variable declared in the parent scope

- Curly braces are used to open and close a block (`{}`).

- Blocks are critical for defining functions, conditional statements, and iteration statements

```
1  int  x = 1;
2  {
3      int  x = 2;
4      int  y = x;
5  }
6  int  y = x;
```

- **Key Question:** When we use a variable name, what variable declaration is it referring to?

- **Name binding** is a set of rules to answer this question by associating a specific variable name to a specific in-scope variable declaration

- C uses **static (lexical) scoping** meaning the name binding happens statically at compile time
Steps for name binding

1. Draw circle in source code around use of a variable name
2. Determine which variables with that name are in scope
3. Draw line to variable declaration in the inner most enclosing block
4. Draw circle in source code around variable declaration

```java
int x = 1;
{
    int x = 2;
    int y = x;
}
int y = x;
```

```java
int x = 1;
{
    x = 2;
    {
        int y = x;
        int x = 3;
        x = 4;
    }
    x = 5;
}
int y = x;
```
4. Functions

• A function gives a name to a parameterized sequence of statements
• A function definition describes how a function behaves
• A function call is a new kind of expression
• To evaluate a function call, we execute the definition of the function, and the function’s result is the value of the function call
• A function creates a new block, and thus a new local scope
• All code in this course will be inside functions

4.1. Function Definition

```c
int avg( int x, int y )
{
    int sum = x + y;
    return sum / 2;
}
```

```c
int main()
{
    int a = 10;
    int b = 20;
    int c = ( a + b ) / 2;
    return c;
}
```

• Main is special: it is always the first function executed in a program
• Main returns its “value” to the “system”
• The return value is called the exit status for the program
• Returning 0 means success, 1 means failure
4.2. Function Call

- Function parameters and “local” variables declared within a function are effectively in a new block
- This block is the function’s frame or stack frame

Steps for calling a function

1. Draw called function’s frame w/ parameter boxes
2. Initialize parameters by evaluating expressions in function call
3. Record location of function call
4. Move execution arrow to first statement in called function
5. Evaluate statements inside the called function
6. At return statement, evaluate its argument, record value
7. Return execution arrow back to where the function was called
8. Erase the called function’s frame
9. Use function’s return value as value of function call

```c
int avg(int x, int y)
{
    int sum = x + y;
    return sum / 2;
}

int main()
{
    int a = 10;
    int b = 20;
    int c = avg(a, b);
    return c;
}
```
Draw a state diagram for this program

```c
int avg( int x, int y )
{
    int sum = x + y;
    return sum / 2;
}

int main()
{
    int y = 10;
    int x = 20;
    int a = avg( avg(y,x),
                 avg(30,40) );
    return a;
}
```
4.3. The `printf` Function

The `printf` function is provided by the C standard library and can be used to print values to the screen. Here is pseudocode for the `printf` function definition.

```c
printf( format_string, value0, value1, ... )
{
    substitute value0 into format_string
    substitute value1 into format_string
    ...
    display final format_string on the screen
}
```

Here is an example of calling `printf`.

```c
#include <stdio.h>

int avg( int x, int y )
{
    int sum = x + y;
    return sum / 2;
}

int main()
{
    int a = 10;
    int b = 20;
    int c = avg( a, b );
    printf( "average of %d and %d is %d\n", a, b, c );
    return 0;
}
```
5. Conditional Statements

- Conditional statements enable programs to make decisions based on the values of their variables
- Conditional statements enable non-linear forward control flow

5.1. Boolean Operators

- Boolean operators are used in expressions which evaluate to a “boolean” value (i.e., true or false)
- In C, a “boolean” value is just an integer, where we interpret a value of zero to mean false and any non-zero value to mean true

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expr1 == expr2</code></td>
<td>tests if <code>expr1</code> is equal to <code>expr2</code></td>
</tr>
<tr>
<td><code>expr1 != expr2</code></td>
<td>tests if <code>expr1</code> is not equal to <code>expr2</code></td>
</tr>
<tr>
<td><code>expr1 &lt; expr2</code></td>
<td>tests if <code>expr1</code> is less than <code>expr2</code></td>
</tr>
<tr>
<td><code>expr1 &lt;= expr2</code></td>
<td>tests if <code>expr1</code> is less than or equal to <code>expr2</code></td>
</tr>
<tr>
<td><code>expr1 &gt; expr2</code></td>
<td>tests if <code>expr1</code> is greater than <code>expr2</code></td>
</tr>
<tr>
<td><code>expr1 &gt;= expr2</code></td>
<td>tests if <code>expr1</code> is greater than or equal to <code>expr2</code></td>
</tr>
<tr>
<td><code>!expr</code></td>
<td>computes the logical NOT of <code>expr</code></td>
</tr>
<tr>
<td><code>expr1 &amp;&amp; expr2</code></td>
<td>computes the logical AND of <code>expr1</code> and <code>expr2</code></td>
</tr>
<tr>
<td>`expr1</td>
<td></td>
</tr>
</tbody>
</table>

Using these operators in an expression evaluates to either zero (false) or one (true)
### 5. Conditional Statements

#### 5.1. Boolean Operators

<table>
<thead>
<tr>
<th>Category</th>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary</td>
<td>!</td>
<td>right to left</td>
</tr>
<tr>
<td>Multiplicative</td>
<td>* / %</td>
<td>left to right</td>
</tr>
<tr>
<td>Additive</td>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>Relational</td>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>Equality</td>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>Logical AND</td>
<td>&amp;&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td>Logical OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assignment</td>
<td>=</td>
<td>right to left</td>
</tr>
</tbody>
</table>

Mixing boolean operators to create a complex expression

```cpp
#include <stdio.h>

int main()
{
    int x = 7 < 6 && 3 > 1 || !0;
    printf("%d\n",x);
}
```

Experiment with [http://cpp.sh](http://cpp.sh):
5.2. if/else Conditional Statements

```c
int min( int x, int y )
{
    int z;
    if ( x < y ) {
        z = x;
    }
    else {
        z = y;
    }
    return z;
}

int main()
{
    min( 5, 9 );
    min( 7, 3 );
    return 0;
}
```
5. Conditional Statements

5.2. if/else Conditional Statements

```c
int min( int x, int y )
{
    if ( x < y ) {
        return x;
    }
    return y;
}

int main()
{
    min( 5, 9 );
    return 0;
}

int min3( int x, int y, int z )
{
    if ( x < y ) {
        if ( x < z )
            return x;
    } else if ( y < z ) {
        return y;
    }
    return z;
}

int main()
{
    min3( 3, 7, 2 );
    return 0;
}
```
5.3. switch/case Conditional Statements

```c
int days_in_month( int month )
{
    int x;
    switch ( month )
    {
        case 1: x = 31; break;
        case 2: x = 28; break;
        case 3: x = 31; break;
        case 4: x = 30; break;
        case 5: x = 31; break;
        case 6: x = 30; break;
        case 7: x = 31; break;
        case 8: x = 31; break;
        case 9: x = 30; break;
        case 10: x = 31; break;
        case 11: x = 30; break;
        case 12: x = 31; break;
        default: x = -1;
    }
    return x;
}

int main()
{
    days_in_month( 7 );
    return 0;
}
```
int days_in_month( int month )
{
    int x;
    if ( month == 2 ) {
        x = 28;
    }
    else {
        switch ( month )
        {
        case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                x = 31;
                break;
        case 4:
            case 6:
            case 9:
            case 11:
                x = 30;
                break;
        default:
            x = -1;
        }
    }
    return x;
}
Indentifying Primes

Write a C function that takes one integer input \((x)\) that is between 0 and 9 (inclusive) and returns a boolean output. The function should return true if the input is prime (i.e., 2,3,5,7) and return false if the input is not prime. Use a \texttt{case/switch} conditional statement to explicitly check for primes.

\begin{verbatim}
int is_prime( int x ) {

}
\end{verbatim}
6. Iteration Statements

- **Iteration statements** enable programs to repeat code multiple times based on a conditional expression
- Iteration statements enable **backward flow control**

6.1. while Loops

```c
int gcd( int x, int y )
{
    while ( y != 0 ) {
        if ( x < y ) {
            int temp = x;
            x = y;
            y = temp;
        }
        else {
            x = x - y;
        }
    }
    return x;
}

int main()
{
    gcd(5,15);
    return 0;
}
```

<table>
<thead>
<tr>
<th>stmt</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
**do-while Loops**

- while loops execute the loop body zero or more times
- do/while loops execute the loop body one or more times

```c
1 int x = 0;
2 while ( x < n ) {
3     printf("%d ",x);
4     x = x + 1;
5 }
```

```c
1 int x = 0;
2 do {
3     printf("%d ",x);
4     x = x + 1;
5 } while ( x < n );
```

### 6.2. for Loops

```c
1 int mul( int x, int y )
2 {
3     int result = 0;
4     for ( int i=0; i<y; i=i+1 ) {
5         result = result + x;
6     }
7     return result;
8 }

10 int main()
11 {
12     mul(2,3);
13     return 0;
14 }
```
Output a sequence

Write a C function that takes one integer input (N) that is non-negative. The C function should output a sequence of integers according to the pattern on the left.

<table>
<thead>
<tr>
<th>N</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 0 0</td>
</tr>
<tr>
<td>3</td>
<td>0 0 0 3</td>
</tr>
<tr>
<td>4</td>
<td>0 0 0 3 4</td>
</tr>
<tr>
<td>5</td>
<td>0 0 0 3 4 5</td>
</tr>
<tr>
<td>6</td>
<td>0 0 0 3 4 5 6</td>
</tr>
</tbody>
</table>

```c
void print_seq( int N ) {
}
```
7. Syntactic Sugar

- **Syntactic sugar** adds new syntax but not new semantics
- Syntactic sugar makes it easier to write certain programming patterns
- Syntactic sugar does not introduce any fundamentally new behavior

**for loops are syntactic sugar**

```java
for ( int i = 0; i < y; i = i+1 ) {
    result = result + x;
}
```

```java
{
    int i = 0;
    while ( i < y ) {
        result = result + x;
        i = i + 1;
    }
}
```

**Assignment Operators**

<table>
<thead>
<tr>
<th>Sugar</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>x += y;</td>
<td>x = x + y;</td>
</tr>
<tr>
<td>x -= y;</td>
<td>x = x - y;</td>
</tr>
<tr>
<td>x *= y;</td>
<td>x = x * y;</td>
</tr>
<tr>
<td>x /= y;</td>
<td>x = x / y;</td>
</tr>
</tbody>
</table>

**Postfix/Prefix Operators**

<table>
<thead>
<tr>
<th>Sugar</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>x++;</td>
<td>x = x + 1;</td>
</tr>
<tr>
<td>++x;</td>
<td>x = x + 1;</td>
</tr>
<tr>
<td>x--;</td>
<td>x = x - 1;</td>
</tr>
<tr>
<td>--x;</td>
<td>x = x - 1;</td>
</tr>
</tbody>
</table>

Be careful, the **value** of ++x is x + 1, but the **value** of x++ is x.

```java
int i = 1;
int j = ++i; // i == 2; j == 2
int k = i++; // i == 3; k == 2
```
Ternary operator is syntactic sugar

```c
int min( int x, int y )
{
    if ( x < y ) {
        return x;
    }
    return y;
}

int min( int x, int y )
{
    return ( x < y ) ? x : y;
}
```

<table>
<thead>
<tr>
<th>Category</th>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Postfix</td>
<td>a++   a--</td>
<td>left to right</td>
</tr>
<tr>
<td>Unary</td>
<td>! +a -a</td>
<td>right to left</td>
</tr>
<tr>
<td>Multiplicative</td>
<td>* / %</td>
<td>left to right</td>
</tr>
<tr>
<td>Additive</td>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>Relational</td>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>Equality</td>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>Logical AND</td>
<td>&amp;&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td>Logical OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assignment</td>
<td>= += -= *= /= a?b:c</td>
<td>right to left</td>
</tr>
</tbody>
</table>