# ECE 2400 Computer Systems Programming
## Fall 2018
### Topic 1: Introduction to C

School of Electrical and Computer Engineering  
Cornell University

revision: 2018-09-03-15-59

<table>
<thead>
<tr>
<th>Chapter</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, Literals, Operators, Expressions</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2.1. Variables</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>2.2. Literals</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>2.3. Operators</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>2.4. Expressions</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2.5. Simple C Programs</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>Blocks, Scope, Name Binding</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>3.1. Blocks</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>3.2. Scope</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>3.3. Name Binding</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>Functions</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>4.1. Function Definition</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>4.2. Function Call</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>4.3. The printf Function</td>
<td>16</td>
</tr>
</tbody>
</table>
5 Conditional Statements

5.1. Boolean Operators .................................................. 17
5.2. if/else Conditional Statements .................................. 19
5.3. switch/case Conditional Statements ......................... 22

6 Iteration Statements

6.1. while Loops ......................................................... 25
6.2. for Loops ......................................................... 27

7 Syntactic Sugar

Copyright © 2018 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.
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>Syntax</th>
<th>Semantics</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence of statements</td>
<td>It is raining outside. Should I use an umbrella?</td>
<td>It is raining outside.</td>
<td>punctuation; “I” is a pronoun; “is” uses present tense</td>
<td>remember that it is raining outside when considering umbrella</td>
</tr>
<tr>
<td>Sentence</td>
<td>Sentence</td>
<td>Sentence grammar</td>
<td>Sentence meaning</td>
<td>Memory of prior statements</td>
</tr>
</tbody>
</table>
2. Variables, Literals, Operators, Expressions

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, Literals, Operators, Expressions

- A variable is a box (in the computer’s memory) which stores a value; variables have names and are used for “state”

- A literal is a value written exactly as it is meant to be interpreted; a literal is not a name, it is the value itself

- An operator is a symbol with special semantics to “operate” on variables and literals

- An expression is a combination of variables, literals, and operators which evaluates to a new value
2. Variables, Literals, 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

• Statements in C must end with a semicolon

2.2. Literals

• A **literal** is a value written exactly as it is meant to be interpreted

• A variable is a name that can represent different values

• A constant variable is a name that can represent the same value

• A literal is not a name but the value itself

• **Example integer literals**
  – 13 literally the number 13 in base 10
  – -13 literally the number -13 in base 10
  – 0x13 literally the number 13 in base 16 (i.e., 19 in base 10)
  – 0xdeadbeef literally a large number in base 16
2.3. 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, possibly using a literal

```
1 int my_variable;
2 my_variable = 42;
```

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

```
1 int my_variable = 42;
```

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

• An **expression** is a combination of variables, literals, 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!
2.5. Simple C Programs

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

**Translating our “English” program into a C program**

```
01 int x;
02 x = 3;
03 int y;
04 y = 2;
05 int z;
06 z = x + y;
```

An empty box in a state diagram means the variable contains an undefined value.

**Draw a state diagram corresponding to the execution of this program**

```
01 int x = 3;
02 int y = 2;
03 int z = x + y * 5;
04 y = x + y * x + y;
```
3. Blocks, Scope, Name Binding

- Blocks, scope, and name binding help provide syntax and semantics for managing more complex programs

3.1. Blocks

- A block is a compound statement
- Curly braces are used to open and close a block ({}); the trailing semicolon may be (should be) omitted
- Blocks are critical for defining functions, conditional statements, and iteration statements

```c
1  { int x = 2; int y = x; }
2  { int z = 3; z = z + 1; }
```
3.2. Scope

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

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

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

- C blocks create new local scopes

- We can declare new variables that are only in scope in the block

- We can declare new variables in the local scope with same name as a variable declared in the parent scope (syntax)

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

3.3. Name Binding

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

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

```java
01 int x = 1;
02 {
03   int x = 2;
04   int y = x;
05 }
06 int y = x;
```

```java
01 int x = 1;
02 {
03   x = 2;
04     {
05       int y = x;
06       int x = 3;
07         x = 4;
08     }
09   x = 5;
10 }
11 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 execute a function
- All code in C programs are inside functions!

4.1. Function Definition

```c
rtype function_name( ptype0 pname0, ptype1 pname1, ... )
{
    function_body;
}
```

- A **function name** is a unique identifier for the function
- The **function body** is the parameterized sequence of statements
- The **parameter list** is a list of parameter types and names
- The **return type** is the type of the value returned by the function

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

4.2. Function Call

```c
int main()
{
    int a = 10;
    int b = 20;
    int c = (a + b) / 2;
    return 0;
}
```

- `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 zero means success in Linux
- Returning greater than zero means failure in Linux

4.2. Function Call

```c
function_name( pvalue0, pvalue1, ... )
```

- To call a function we simply use its name and pass in one value for each parameter in the parameter list surrounded by parenthesis
- If parameters are expressions, then we must evaluate them before calling the function
- A function call is itself an expression which evaluates to the value returned by the function
- Function parameters and "local" variables declared within a function are effectively in a new block which is called the function’s stack frame
- The value of each parameter is copied into these local variables (call-by-value semantics)
Steps for calling a function

0. Allocate variable for return value on caller’s stack frame
1. Draw called function’s stack 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, update variable in caller
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

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

int main()
{
    int a = 10;
    int b = 20;
    int c = avg( a, b );
    return 0;
}
```
Draw a state diagram corresponding to the execution of this program

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

int main()
{
    int y = 10;
    int x = 20;
    x = avg( avg(y,x), avg(30,40) );
    return 0;
}
```
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)
• Logical operators also have a place in the operator precedence table

<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>
5. Conditional Statements

5.2. if/else Conditional Statements

```plaintext
if ( conditional_expression )
  then_statement;
else
  else_statement;
```

- A **conditional expression** is an expression which returns a boolean
- The **then statement** is executed if the conditional expression is true
- The **else statement** is executed if the conditional expression is false
- Recall that blocks are compound statements

```plaintext
if ( conditional_expression0 )
  then_statement0;
else if ( conditional_expression1 )
  then_statement1;
else
  else_statement;
```

- If the first cond expression is true, execute first then statement
- If the first cond expression is false, evaluate second cond expression
- If second cond expression is true, execute second then statement
- If second cond expression is false, execute else statement
```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;
}
```
```c
int min3( int x, int y, int z )
{
    if ( x < y ) {
        if ( x < z )
            return x;
    } else if ( y < z ) {
        return y;
    } else return z;
}

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

```java
switch ( selection_expression ) {

case case_label0:
    case_statements0;
    break;

    case case_label1:
    case_statements1;
    break;

    case case_label2:
    case_statements3;
    break;

    default:
    default_statements;
}
```

- A selection expression is an expression which returns a value
- The value is matched against the case labels
- If there is a match, then corresponding case statements are executed
- A break statement means to jump to end of switch block
- If no case labels match then the default statements are executed
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()
{
    int num_days = days_in_month( 7 );
    // Indicate error to the system
    if ( num_days == -1 )
        return 1;
    // Indicate success to the system
    return 0;
}
Identifying 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 switch/case conditional statement to explicitly check for primes.

```c
int is_prime( int x ) {
    /* Your code here */
}
```
6. Iteration Statements

- Iteration statements enable programs to execute the same code multiple times based on a conditional expression
- Iteration statements enable backward flow control
- Two primary kinds of iteration statements: while and for loops

6.1. while Loops

```python
1 while ( conditional_expression )
2   loop_body;
```

- A conditional expression is an expression which returns a boolean
- The loop body is a statement which is executed as long as conditional expression is true
- An infinite loop is when conditional expression is never false
```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>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.2. for Loops

1. for ( initialization_stmt; cond_expr; increment_stmt )
2.  loop_body;

- The initialization statement is executed once before loop executes
- A conditional expression is an expression which returns a boolean
- The loop body is a statement which is executed as long as conditional expression is true
- The increment statement is executed at the end of each iteration

```c
int mul( int x, int y )
{
    int z = 0;
    for ( int i=0; i<y; i=i+1 ) {
        z = z + x;
    }
    return z;
}

int main()
{
    mul(2,3);
    return 0;
}
```
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 ) {
    // Your code here
}
```
7. Syntactic Sugar

- **Syntactic sugar** adds new syntax but not new semantics
- Syntactic sugar simplifies certain programming patterns
- Syntactic sugar does not introduce any fundamentally new behavior

**for loops are syntactic sugar**

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

{  
    int i = 0;
    while ( i < y ) {
        z = z + 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.

```c
int i = 1;
int j = ++i;
// i == 2; j == 2
```

```c
int i = 1;
int j = i++;
// i == 2; j == 1
```

**Ternary operator is syntactic sugar**

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

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

**Final operator precedence table**

<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>