ECE 2400 Computer Systems Programming  
Fall 2020  

Topic 1: Introduction to C  

School of Electrical and Computer Engineering  
Cornell University  

revision: 2020-09-04-01-14  

1 Statements, Syntax, Semantics, State  

2 Variables, Literals, Operators, Expressions  
   2.1 Variables  
   2.2 Literals  
   2.3 Operators  
   2.4 Expressions  
   2.5 Simple C Programs  

3 Blocks, Scope, Name Binding  
   3.1 Blocks  
   3.2 Scope  

4 Functions  
   4.1 Function Definition  
   4.2 Function Call  
   4.3 The printf Function
5 Conditional Statements

5.1. Boolean Operators .................................................. 16
5.2. if/else Conditional Statements ............................... 18

6 Iteration Statements ............................................ 21

6.1. for Loops .............................................................. 21

7 Syntactic Sugar ..................................................... 23

Additional Topics Covered in zyBook

While the zyBook covers the topics in lecture, it also contains additional material not covered in lecture. Students are responsible for all material covered in lecture and in the zyBook. Material from both lecture and the zyBook will be assessed in the exams. Examples of additional material covered in the zyBook but not lecture include:

- basic input via scanf
- switch/case conditional statements
- while iteration statements
- bitwise operators
- variable name scope
- variable name scope in the context of functions
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>Sequence of statements</th>
<th>It is raining outside. Should I use an umbrella?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentence</td>
<td>It is raining outside.</td>
</tr>
<tr>
<td>Sentence grammar</td>
<td>punctuation; “I” is a pronoun; “is” uses present tense</td>
</tr>
<tr>
<td>Sentence meaning</td>
<td>rain is water condensed from the atmosphere, outside means in the outdoors</td>
</tr>
<tr>
<td>Memory of prior statements</td>
<td>remember that it is raining outside when considering umbrella</td>
</tr>
</tbody>
</table>
2. Variables, Literals, Operators, Expressions

- **variables** is a box (in the computer’s memory) which stores a value; variables have names and are used for “state”
- **literals** is a value written exactly as it is meant to be interpreted; a literal is not a name, it is the value itself
- **operators** is a symbol with special semantics to “operate” on variables and literals
- **expressions** is a combination of variables, literals, and operators which evaluates to a new value

An example English “program”

01 Create box named x.
02 Put value 3 into box named x.
03 Create box named y.
04 Put value 2 into box named y.
05 Create box named z.
06 Put x + y into box named z.
2.1. Variables

• __________ is a box (in the computer’s memory) which stores a value

• __________ is used to name a variable

• __________ specifies the kind of values that can be stored in a variable

• __________________________ 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 for a box that can hold different values

• A constant variable is a name that can hold a single 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

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

• \( = \) “assigns” a new value to a variable

• 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

```c
    int my_variable;
    my_variable = 42;
```

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

```c
    int my_variable = 42;
```

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

• Division is integer division
  – 6 / 2 is 3
  – 5 / 2 is 2 not 2.5

• Modulus is integer remainder
  – 6 % 2 is 0
  – 5 % 2 is 1

• We will explore overflow, underflow, etc in Topic 3
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

```
int x;
int y;
int z;
```

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

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

- Blocks, scope, and name binding provide syntax and semantics to help manage more complex programs

3.1. Blocks

- A block is a compound statement
- Curly braces are used to open and close a block ({})
- Blocks are critical for defining functions, conditional statements, and iteration statements

```c
{ 
    int x = 2;  
    int y = x;  
};

{ 
    int z = 3;  
    z = z + 1;  
};
```

- Since a block is itself a statement, it has a trailing semicolon
- In practice, the trailing semicolon may be (should be) omitted

```c
{ 
    int x = 2;  
    int y = x;  
}
```
3.2. Scope

- **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

```c
int w = 1;
{
    int x = 2;
    int y = 3;
}
int z = w;
```

Use an X on the right of a variable box to indicate that this variable has gone out of scope and thus has been deallocated

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

```c
int x = 1;
{
    int y = 2;
    {
        y = 3;
    }
    x = y;
}
int z = y;
```
4. Functions

• __________ names a parameterized sequence of statements
• __________ describes how a function behaves
• __________ is a new kind of expression to execute a function
• All code in C programs are inside functions!

4.1. Function Definition

```
1 rtype function_name( ptype0 pname0, ptype1 pname1, ... )
2 {
3     function_body;
4 }
```

• __________ is a unique identifier for the function
• __________ is the parameterized sequence of statements
• __________ is a list of parameter types and names
• __________ is the type of the value returned by the function

```
1 int avg( int x, int y )
2 {
3     int sum = x + y;
4     int ans = sum / 2;
5     return ans;
6 }
```

• **Function prototype** is just line 1

• Useful for informing the compiler that a function exists with a specific interface, but without specifying the implementation
```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

1. Evaluate parameters, allocate temp storage in caller’s stack frame?
2. Allocate storage on caller’s stack frame for the return value?
3. Allocate the callee’s stack frame with space allocated for parameters
4. Copy evaluated parameters from step 1 into callee’s stack frame
5. Record location of function call
6. Move execution arrow to first statement in callee
7. Evaluate statements inside the callee
8. At return statement, evaluate argument, update variable in caller
9. Return execution arrow back to where function was called in caller
10. Deallocate the callee’s stack frame

```
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;
}
```

Use dot in an execution box for a function call. Always shift one column of execution boxes to the left when you move the execution arrow backwards. Use vertical line to for skipping statements.
Draw a state diagram corresponding to the execution of this program

```
int add(int r, int s) {
    int t = r + s;
    return t;
}

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

int main() {
    int a = 10;
    int b = 20;
    int c = avg(a, b);
    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.

```
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.

```
#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 either 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>expr1 == expr2</td>
<td>tests if expr1 is equal to expr2</td>
</tr>
<tr>
<td>expr1 != expr2</td>
<td>tests if expr1 is not equal to expr2</td>
</tr>
<tr>
<td>expr1 &lt; expr2</td>
<td>tests if expr1 is less than expr2</td>
</tr>
<tr>
<td>expr1 &lt;= expr2</td>
<td>tests if expr1 is less than or equal to expr2</td>
</tr>
<tr>
<td>expr1 &gt; expr2</td>
<td>tests if expr1 is greater than expr2</td>
</tr>
<tr>
<td>expr1 &gt;= expr2</td>
<td>tests if expr1 is greater than or equal to expr2</td>
</tr>
<tr>
<td>!expr</td>
<td>computes the logical not of expr</td>
</tr>
<tr>
<td>expr1 &amp;&amp; expr2</td>
<td>computes the logical and of expr1 and expr2</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>
5.2. if/else Conditional Statements

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

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

int main()
{
    int a = min(5, 9);
    int b = min(7, 3);
    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()
{
    int a = min3( 3, 7, 2 );
    return 0;
}
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. for Loops

```c
for ( initialization_stmt; cond_expr; increment_stmt )
  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;
}
```

```c
int main()
{
  int a = 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 right. So for example, if N is 4, then the C function should print out 0 0 0 3 4. The C function should always return 0.

<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
int print_seq( int N ) {
```


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**

```c
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></td>
</tr>
<tr>
<td>x -= y</td>
<td></td>
</tr>
<tr>
<td>x *= y</td>
<td></td>
</tr>
<tr>
<td>x /= y</td>
<td></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`.

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

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

**Ternary operator is syntactic sugar**

```
1 int min( int x, int y )
2 {
3     if ( x < y )
4         return x;
5     return y;
6 }
```

```
1 int min( int x, int y )
2 {
3     return ( x < y ) ? x : y;
4 }
```

**Final operator precedence table**

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