ECE 2400 Computer Systems Programming
Fall 2017

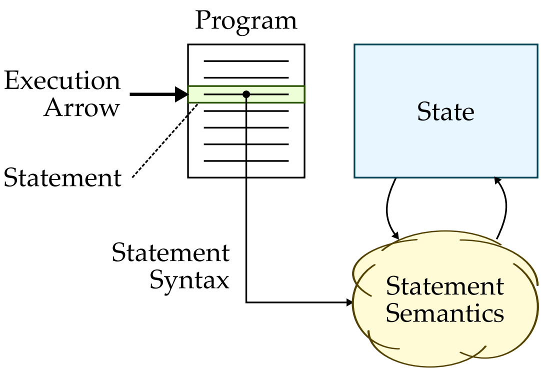
T01 Reading C Programs

School of Electrical and Computer Engineering
Cornell University

revision: 2017-09-06-20-23

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



Program	Sequence of statements	It is raining outside. Should I use an umbrella?
Statement	Sentence	It is raining outside.
Syntax	Sentence grammar	punctuation; "I" is a pronoun; "is" uses present tense
Semantics	Sentence meaning	rain is water condensed from the atmosphere, outside means in the outdoors
State	Memory of prior statements	remember that it is raining outside when considering umbrella

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.
```

stmt	x	y	z
1			
2			
3			
4			
5			
6			

2. Variables, Operators, Expressions

- A variable is a box (in the computer's memory) which stores a value; variables are used for "state"
- 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

2.1. 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

```
int my_variable;
int MY_VARIABLE;
int variable_0;
int 0_variable;
int variable$1;
```

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

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

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

```
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

```
3 + 4
3 + 4 * 2 + 7
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

Category	Operator	Associativity
Multiplicative	* / %	left to right
Additive	+ -	left to right
Assignment	=	right to left

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

```
int x;
x = 3;
int y;
y = 2;
int z;
z = x + y;
```

Use ? to indicate undefined value in stack frame diagram

Draw stack frame diagram corresponding to the execution of this program

```
int x = 3;
int y = 2;
int z = x + y * 5;
y = x + y * x + y;
```

3. Name Binding

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

```
int x = 1;
int x = 2;
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

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

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

```
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 execute a function
- All code in this course will be inside functions

4.1. Function Definition

```
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

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

```
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, greater than zero means failure

4.2. Function Call

```
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

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;
  return sum / 2;
}

int main()
{
  int a = 10;
  int b = 20;
  int c = avg( a, b );
  return 0;
}
```

Draw stack frame diagram corresponding to the execution of this program

```
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.

```
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;
}
```

- Execute this code via `http://cpp.sh`
- Examine the machine instructions via `https://godbolt.org`

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>
<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 < expr2</code></td><td>tests if <code>expr1</code> is less than to <code>expr2</code></td></tr>
<tr><td><code>expr1 <= expr2</code></td><td>tests if <code>expr1</code> is less than or equal to <code>expr2</code></td></tr>
<tr><td><code>expr1 > expr2</code></td><td>tests if <code>expr1</code> is greater than to <code>expr2</code></td></tr>
<tr><td><code>expr1 >= 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 && expr2</code></td><td>computes the logical AND of <code>expr1</code> and <code>expr2</code></td></tr>
<tr><td><code>expr1 || expr2</code></td><td>computes the logical OR of <code>expr1</code> and <code>expr2</code></td></tr>
</table>

Using these operators in an expression evaluates to either zero (false) or one (true)

Category	Operator	Associativity
Unary	`!`	right to left
Multiplicative	`* / %`	left to right
Additive	`+ -`	left to right
Relational	`< <= > >=`	left to right
Equality	`== !=`	left to right
Logical AND	`&&`	left to right
Logical OR	`||`	left to right
Assignment	`=`	right to left

Mixing boolean operators to create a complex expression

```
7 < 6 && 3 > 1 || !0
```

Experiment with `http://cpp.sh`:

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

5.2. `if/else` Conditional Statements

```
if ( conditional_expression ) {
  then_block;
}
else {
  else_block;
}
```

- A conditional expression is an expression which returns a boolean
- The then block is executed if the conditional expression is true
- The else block is executed if the conditional expression is false

```
if ( conditional_expression0 ) {
  then_block0;
}
else if ( conditional_expression1 ) {
  then_block1;
}
else {
  else_block;
}
```

- If the first cond expression is true, execute first then block
- If the first cond expression is false, evaluate second cond expression
- If second cond expression is true, execute second then block
- If second cond expression is false, execute else block

```
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;
}
```

```
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

```
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 result = days_in_month( 7 );

  // Indicate error to the system
  if ( result == -1 )
    return 1;

  // Indicate success to the system
  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 case/switch conditional statement to explicitly check for primes.

```
int is_prime( int x ) {
```

6. Iteration Statements

- Iteration statements enable programs to repeat 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

```
while ( conditional_expression ) {
  loop_body;
}
```

- A conditional expression is an expression which returns a boolean
- The loop body is executed as long as conditional expression is true
- An infinite loop is when conditional expression is never false

```
1  int gcd( int x, int y )
2  {
3    while ( y != 0 ) {
4      if ( x < y ) {
5        int temp = x;
6        x = y;
7        y = temp;
8      }
9      else {
10       x = x - y;
11     }
12   }
13   return x;
14 }
15
16 int main()
17 {
18   gcd(5,15);
19   return 0;
20 }
```

stmt	x	y
4		
4		
4		
4		
4		
4		

6.2. for Loops

```
for ( initialization_stmt; cond_expr; increment_stmt ) {
  loop_body;
}
```

- The initialization statement is executed once before loop executes
- The loop body is executed as long as conditional expression is true
- The increment statement is executed at the end of each iteration

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

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.

N	output						
0:	0						
1:	0	0					
2:	0	0	0				
3:	0	0	0	3			
4:	0	0	0	3	4		
5:	0	0	0	3	4	5	
6:	0	0	0	3	4	5	6

```
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

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

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

Assignment Operators

Sugar	Semantics
`x += y;`	`x = x + y;`
`x -= y;`	`x = x - y;`
`x *= y;`	`x = x * y;`
`x /= y;`	`x = x / y;`

Postfix/Prefix Operators

Sugar	Semantics
`x++;`	`x = x + 1;`
`++x;`	`x = x + 1;`
`x--;`	`x = x - 1;`
`--x;`	`x = x - 1;`

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

```
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;
}
```

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

Category	Operator	Associativity
Postfix	`a++ a--`	left to right
Unary	`! ++a --a`	right to left
Multiplicative	`* / %`	left to right
Additive	`+ -`	left to right
Relational	`< <= > >=`	left to right
Equality	`== !=`	left to right
Logical AND	`&&`	left to right
Logical OR	`||`	left to right
Assignment	`= += -= *= /= a?b:c`	right to left