ECE 2400 Computer Systems Programming Spring 2025

Topic 1: Introduction to C

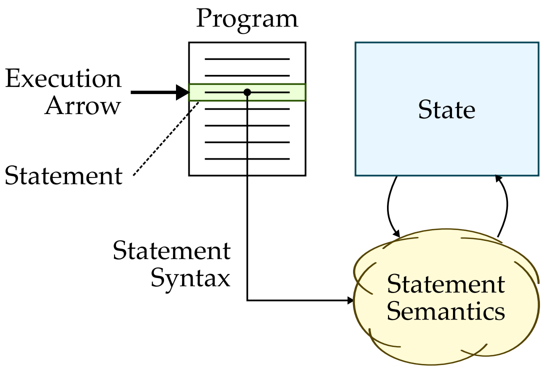
School of Electrical and Computer Engineering
Cornell University

revision: 2025-01-27-15-11

zyBooks The zyBooks logo is used to indicate additional readings and coding labs included in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

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



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

An example English "program"

```
Create box named x.
Put value 3 into box named x.
Create box named y.
Put value 2 into box named y.
Create box named z.
Put x + y into box named z.
```

2. Variables, Literals, Operators, Expressions

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

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

```c
int my_variable;
int MY_VARIABLE;
int myVariable;
int MyVariable;
int variable_0;
int _variable;
int __variable__;
int 0_variable;
int variable$1;
```

- ____________________________ 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 for a box 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

```
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 (%)
- 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

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

$$((3+4)\times 2)+7 \qquad (3+4)\times(2+7) \qquad 3+(4\times 2)+7$$
$$(7\times 2)+7 \qquad 7\times 9 \qquad 3+8+7$$
$$14+7 \qquad\qquad 11+7$$

$$((3\times 4)/2)\times 6 \qquad (3\times 4)/(2\times 6) \qquad 3\times(4/2)\times 6$$
$$(12/2)\times 6 \qquad 12/12 \qquad 3\times 2\times 6$$
$$6\times 6 \qquad\qquad 6\times 6$$

- 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!

2.5. 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;
x = 3;
int y;
y = 2;
int z;
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

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

3. Blocks and Scope

- Blocks and scope 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

```
{
  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

```
{
  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

```
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

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

The course zyBook includes more information on name binding which provides a precise set of rules for associating a specific variable name to a specific in-scope variable declaration.

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

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

- __________________ 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

```
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

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

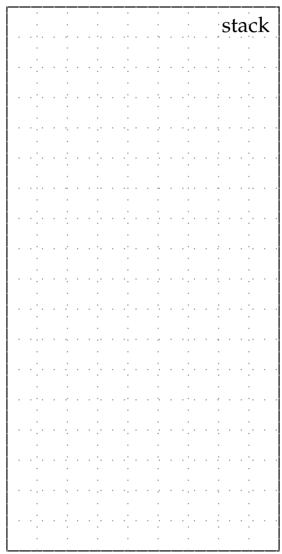
- To call a function we simply use its name and pass in one value for each parameter in the parameter list surrounded by parenthesis
- If parameters are expressions, then we must evaluate them *before* calling the function
- A function call is itself an expression which evaluates to the value returned 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;
}
```



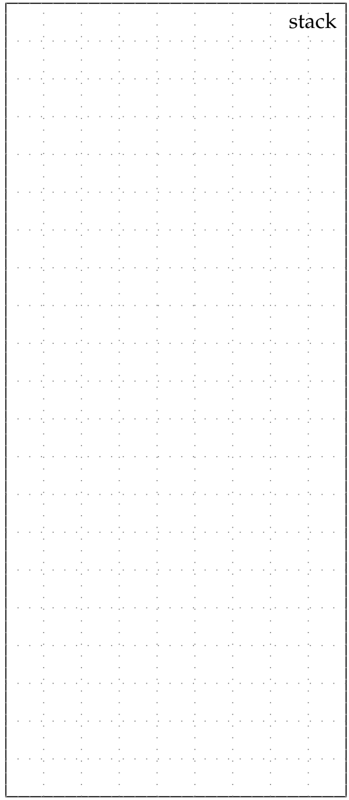
Use dot in an execution box for a function call. Always shift one column of execution boxes to the left when you move the execution arrow backwards. Use vertical line to for skipping statements.

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`.

```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;
}
```

zyBooks `printf` is used to send output to the console. The course zyBook also discusses `scanf` which is used to retrieve input from the console.

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 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

`expr1 == expr2`	tests if `expr1` is ________________ to `expr2`
`expr1 != expr2`	tests if `expr1` is ________________ to `expr2`
`expr1 < expr2`	tests if `expr1` is ________________ to `expr2`
`expr1 <= expr2`	tests if `expr1` is ________________ to `expr2`
`expr1 > expr2`	tests if `expr1` is ________________ to `expr2`
`expr1 >= expr2`	tests if `expr1` is ________________ to `expr2`
`!expr`	computes the logical ______ of `expr`
`expr1 && expr2`	computes the logical ______ of `expr1` and `expr2`
`expr1 || expr2`	computes the logical ______ of `expr1` and `expr2`

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

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

5.2. `if/else` Conditional Statements

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

- ________________ is an expression which returns a Boolean
- ________________ is executed if the conditional expression is true
- ________________ is executed if the conditional expression is false

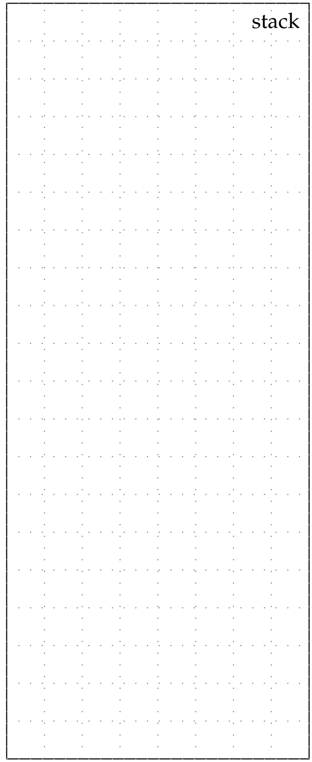
- Recall that blocks are compound statements

```
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

```c
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;
}
```



zyBooks The course zyBook includes more information on `switch/case` conditional statements that enable immediately jumping to a specific statement based on a selection expression.

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

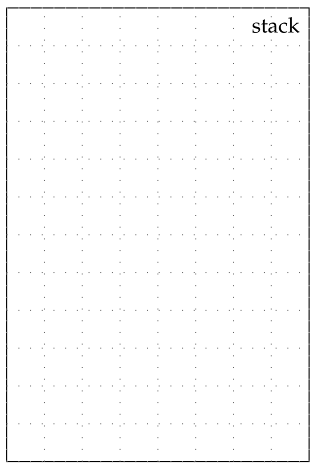
6.1. `for` Loops

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

- ____________________ is executed once before loop executes
- ____________________ is an expression which returns a Boolean
- ____________________ is a statement which is executed as long as the conditional expression is true
- ____________________ is executed at the end of each iteration

```
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()
{
  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.

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

```
int print_seq( int N ) {
```

zyBooks The course zyBook includes more information on `while` loops, an alternative form of iteration statement, and more generally, *syntatic sugar* which adds new syntax but not new semantics