University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

Examples of C Programs with Loops

Time for Some Detailed Examples

Let's do some examples of program execution.

Before you can execute a program, you need to **learn how to compile**.

You will learn that **in the lab**.

You should also **take a look at the style guidelines** for the class (see the Wiki).

The examples obey most style rules, but space is tight in slides.

You may want to get out a sheet of paper…

Let’s See How This Loop Works

```
/* Print 20 Fibonacci numbers. */
int A = 1; int B = 1; int C; int D;
for (D = 0; 20 > D; D = D + 1) {
    printf ("%d\n", A);
    C = A + B;
    A = B;
    B = C;
}
```

NOTE: Example programs are available online.
Feel free to try them before/during/after class.

One Statement/Step at a Time…

comment	A	B	C	D	output
before loop	1	1	bits	bits	
init				0	
20 > D					
print A					1
C = A + B			2		
A = B	1				
B = C		2			
D = D + 1				1	

One Statement/Step at a Time…

comment	A	B	C	D	output
(previous slide)	1	2	2	1	
20 > D					
print A					1
C = A + B			3		
A = B	2				
B = C		3			
D = D + 1				2	

One Statement/Step at a Time…

comment	A	B	C	D	output
(previous slide)	2	3	3	2	
20 > D					
print A					2
C = A + B			5		
A = B	3				
B = C		5			
D = D + 1				3	

One Statement/Step at a Time…

comment	A	B	C	D	output
(previous slide)	3	5	5	3	
20 > D					
print A					3
C = A + B			8		
A = B	5				
B = C		8			
D = D + 1				4	

One Statement/Step at a Time…

comment	A	B	C	D	output
(previous slide)	5	8	8	4	
20 > D					
print A					5
C = A + B			13		
A = B	8				
B = C		13			
D = D + 1				5	

One Statement/Step at a Time…

comment	A	B	C	D	output
(previous slide)	8	13	13	5	
20 > D					
print A					8
C = A + B			21		
A = B	13				
B = C		21			
D = D + 1				6	

Each Loop Iteration Prints One Number

The output column on the last few slides **produces the first twenty numbers** in the Fibonacci sequence (on separate lines, without commas):

1, 1, 2, 3, 5, 8, 13, … , 6765

Steps for a Factorial Printing Program

Remember factorials?

$$N! = N \times (N - 1) \times \ldots \times 1$$

The next program…

- prints a welcome message,
- asks user to enter a number,
- uses **`scanf`** to get the number,
- checks that the user typed something valid,
- calculates the factorial of the user's number,
- and prints the factorial.

Recall that **`main`** is a Sequence of Statements

When we develop a program,

- we break down the problem into smaller steps,*
- and express each step with **C** statements.

The six steps on the previous slide

- Are written using **C** statements
- And appear in order in **`main`**.

* Part 4 of our class describes a systematic way to do so. Also see P&P Ch. 6.

Before Statements, We Declare Variables

We need two variables.

- In practice, a programmer may decide to declare more variables as they write statements.
- This program is already finished, so we know how many variables it needs…

```
int number;
/* number given by user         */
int factorial;
/* factorial of user's number */
```

How are Variable Names Chosen?

```
int number;
/* number given by user          */

int factorial;
/* factorial of user's number */
```

Variable names
- are **chosen to describe their meaning**,
- but we **use comments** to give further details.

These variable names are all lower-case.
Be consistent in how you use case with variable names in a program.

Use `printf` to Write to the Display

The **first two steps** use `printf`.

```
/* Print a welcome message,
   followed by a blank line. */
printf (">--- Welcome to the
factorial calculator! ---<\n\n");
```

```
/* A                                 's
   n
```

Warning: On two lines only on slides.
Do not break format (between quotes) over multiple lines!

```
printf ("What factorial shall I
calculate for you today? ");
```

Next Step: Wait for the User to Type a Number

After asking the user to enter a number,

- the program **waits for the user**
- **to type a decimal value** using **`scanf`**.

```
scanf ("%d", &number)
```

The format specifier **`%d`** tells **`scanf`** to **convert decimal ASCII to 2's complement.**

The expression **`&number`** tells scanf to **store the result into the variable `number`.**

Always Check the Return Value!

```
scanf ("%d", &number)
```

Remember that scanf also

- returns **1 if successful** (# of conversions)
- returns **-1 if the user typed something that isn't a decimal number** (such as "hahahaha" … those humans!)

A program can **use the return value** (the value of the `scanf` expression) **to determine what has happened**…

Next Step: Quit if the User Doesn't Behave

```
if (1 != scanf ("%d", &number)) {
   printf ("Only integers, please.\n");
   return 3; /* Program failed. */
}
```

The program **uses an if statement to check the result** of **`scanf`**.

If the user doesn't type a number, the program…
- **prints an error message**, then
- **terminates** and tells the OS that something went wrong (non-zero by convention).

Time for Some Real Work!

```
for (factorial = number; 1 < number;
      number = number - 1) {
     factorial = factorial *
                   (number - 1);
}
```

Note that **C allows you to add extra lines**

- in the middle of `for` loops
- and in expressions
- **to make the code more readable**.

Example: Factorial of 4

comment	`factorial`	`number`
before loop	bits	4
init	4	
1 < number		
loop body	12	
number = number - 1		3
1 < number		
loop body	24	
number = number - 1		2

Example: Factorial of 4

comment	`factorial`	`number`
(previous slide)	24	2
1 < number		
loop body	24	
number = number - 1		1
1 < number		
after loop	24	1

Second Example: Factorial of 7

comment	`factorial`	`number`
before loop	bits	7
init	7	
1 < number		
loop body	42	
number = number - 1		6
1 < number		
loop body	210	
number = number - 1		5

Second Example: Factorial of 7

comment	`factorial`	`number`
(previous slide)	210	5
1 < number		
loop body	840	
number = number - 1		4
1 < number		
loop body	2520	
number = number - 1		3

Second Example: Factorial of 7

comment	factorial	number
(previous slide)	2520	3
1 < number		
loop body	5040	
number = number - 1		2
1 < number		
loop body	5040	
number = number - 1		1

Second Example: Factorial of 7

comment	`factorial`	`number`
(previous slide)	5040	1
1 < number		
after loop	5040	1

Last Step: Print the Answer

```
printf ("\nThe factorial is %d.\n",
        factorial);
```

The format specifier `%d` tells `printf` to **convert 2's complement to decimal ASCII**.

The variable **`factorial` is the expression to be printed**.

Then the program **terminates (successfully)**: `return 0;`

University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

Learning to Read C Code

Another Useful Skill: Reading Code

You can learn a lot by reading code

- How to **express types of problems**.
- How to **properly use application programming interfaces** (APIs) for networking, mathematics, graphics, sound, animation, user interfaces, and so forth.
- How to **make code easy to read** (style).

It’s Often Necessary to Read Code to Understand It

We try to make you write plenty of comments.

When we give you code for class assignments, it will be well-commented (DISCLAIMER: THIS IS NOT A WARRANTY!)

In the real world…

- You will be lucky to find comments.
- Remember the Big Screw award?
- You will be really lucky to find comments in a language that you understand.

Let's Do an Exercise in Code Reading Together

Our next example has no topical comments and uses one-letter variable names.

Let's **figure out what it does**.

For more exercises of this type,

- **use the ECE120 C Analysis tool**.
- But note that the tool
  - has only 14 examples.
  - **Type an answer** before you press 'Check Answer.'

Structure is Similar to Previous Examples

Take a look at the program.

Basic structure is **similar to previous examples**:

- print a prompt,
- wait for input,
- check input for correctness,
- compute something, and
- print a result.

What Input is Expected?

Look at the following:

- the **`scanf`** format,
- the arguments (types must match),
- the error check and the error message.

As input, the program requires…

- two **2's complement** numbers (**`%d`**) (variables **`A`** and **`C`**)
- separated by an **ASCII** character (**`%c`**) (variable **`B`**)

Now Look at the Computation

`if-else` structure with **five cases**.

- The **last case is an error condition**.
- The other four are **ways of calculating variable `D`**.

Notice that variable **`D` is used for the final output**.

When Does the Computation Print an Error?

The last case is reached when…

- **B** is NOT a ‘+’, AND
- **B** is NOT a ‘–’, AND
- **B** is NOT a ‘/’, AND
- **B** is NOT a ‘*’.

In other words, the code generates an error

- **unless the user enters +, –, /, or ***
- as the character between two integers.

How is **D** Computed?

First case: when **B** is ‘+’, **D** is **A** + **C**.

Second case: when **B** is ‘–’, **D** is **A** – **C**.

Third case: when **B** is ‘/’, **D** is **A** / **C**.

Fourth case: when **B** is ‘*’, **D** is **A** * **C**.

So … the program is doing what?

computing the value of an expression with one arithmetic operator

University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

Learning to Test Your Programs

A Necessary Skill: Testing Code

How do you know that your program works?

There's only one correct answer: test it!*

Brooks' Rule of Thumb

- 1/3 planning and design
- 1/6 writing the program
- **1/2 testing**

Just because your program compiles
does not mean your program works!

*Becoming a good tester will take years.
Don't worry if it seems tough.

Our Next Program Calculates the Roots of a Quadratic

Remember the equation?

$$\mathbf{F(x) = Ax^2 + Bx + C}$$

has roots (**F(x) = 0**) at

$$\mathbf{x} = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

where $\sqrt{N}$ is the square root of **N**.

Every Statement Must be Executed

How can we test our program?

Let's start with something simple.

Let's say that we have **a statement that is never executed by tests**.

Does the statement work correctly?

How can we know? We have no tests!

So, no, it **does not work correctly**.

At a minimum, we **must execute every statement** (called **full code coverage**).

What Happens When We Run the Program?

Imagine that we compile and run the program.

Take a look at the code.

The **first statement is a `printf`**.

The **`printf`** always executes, so

- we can **check whether the printf works**
- by simply **looking at the output**.

Choose a Line of Text as Our First Test

The program then **waits for input with `scanf`**.

What input should we give?

Let's **just choose something** concrete.

Say **"0 0 0"** (and then **<Enter>** to start).

What are the values of variables **`a`**, **`b`**, and **`c`**?

0, 0, and 0

What does **`scanf`** return? **3**

What happens next? **Skip the "then" block!**

Continue Analyzing Until the End

With input **"0 0 0"** our program next

- prints the equation to be solved, and
- calculates the discriminant **`D`**.

What is the value of `D`? 0

(Remember that **`a`**, **`b`**, and **`c`** are all 0.)

So **which of the three `if-else` blocks is executed** (first, second, or third)? **second**

And what is `x1`? 0 / 0 → NaN

Was that a Bug?

I think so.

The equation is not quadratic when `a` is 0.

The person who wrote the code
perhaps didn't think of that case.

And neither did I when I edited the code
to present to you.

Bugs can be subtle, and testing can be hard!

We won't fix the bug.

Remember: We Want Full Code Coverage

Let's try again with input **"1 0 0"**.

The same parts of the code execute.

And x1 is? 0

So the **single root is at 0**, and
the **program ends successfully**.

Our equation was $\mathbf{F(x) = x^2\ (+0x + 0)}$, so
plugging in $\mathbf{x = 0}$ does produce $\mathbf{F(x) = 0}$.

But our test does not execute all code!

Adjust the Inputs to Change the `if-else` Results

What statements did not execute?

- “then” block of `scanf` check
- first case of `if-else` solution computation
- third case of `if-else` solution computation

Let’s adjust our inputs
to execute the other solution cases.

“1 0 0” gave the second case because
`D` was not positive and **`D` was 0**.

Use "1 0 1" to Test the Third `if-else` Case

To get `D` negative, change `c` to 1
(then `D` is -4 == **0 * 0 – 4 * 1 * 1**).

For the next test,
- we type **"1 0 1"**,
- and the program tells us
- **there are no real roots**.

Our equation was **F(x) = x^2 (+ 0x) + 1**, so in fact no value of **x** can produce **F(x) = 0**.

Use “1 1 0” to Test the First `if-else` Case

For the first if-else case, we need `D` positive.

To get `D` positive, change `b` to 1 and c to 0
(then `D` is 1 == **1 * 1 – 4 * 1 * 0**).

For the next test,
- we type **“1 1 0”**,
- and the program gives **roots at 0 and -1**.

Our equation was **F(x) = x^2 + x (+ 0)**, so
F(x) = 0 at both **x = 0** and at **x = -1**.

We Need to Execute the "then" Block of `scanf`

So far, we have four tests:
"0 0 0" (known bug), **"1 0 0"**, **"1 0 1"**, **"1 1 0"**

But we still need a test to execute the "then" block of the `scanf` check!

Anything that stops `scanf` from finding three numbers will do. Let's type **"hello"**.

So **five tests** (and **verifying the output by hand!**) gives full code coverage for this program.

Good Testing Must Consider Both Purpose and Structure

Full code coverage is **just a starting point**.

In fact, you should notice that
- one of our tests (**“0 0 0”**)
- exposes a bug
- in a statement that was already covered
- by another test (**“1 0 0”**).

In general, good testing requires that one **think carefully about the purpose** of the code **as well as the structure** of the code.

* *

So Easy that a Computer Can Do It

Full code coverage is easy to explain.

Finding tests to cover more statements means solving some equations.

Computers are good at that (well … pretty good).

The automatic programming feedback tool uses this approach to try to find bugs in your code:

- generate tests to cover everything (if possible),
- then compare your program's results with a "gold" program (written by a professor or TA).