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

ECE 120: Introduction to Computing

Basic I/O in C

Allowing Input from the Keyboard, Output to the Monitor

To control input and output (**I/O**), we use two functions from the standard **C** library.

Put this line at the top of your **C** program:

```
#include <stdio.h>
```

This directive tells the **C** compiler that your program **uses the standard C I/O functions**.

Write Output Using `printf`

To write text onto the display, use **`printf`**.

The “f” means “formatted.”

- When using the function,
- you must **specify the desired format** between quotation marks.

Example:

```
printf ("Here is an example.");
```

The function call above **writes the text between the quotes to the monitor**.

Use Backslash to Include Special ASCII Characters

Certain **ASCII** characters

- control text appearance, and
- are hard to put between quotes.

For example

- **ASCII**'s **linefeed** character (or lf, sometimes called newline)
- **starts a new line of text**.

To **include linefeed, write \n** between quotes.

The **backslash indicates a special ASCII character**. Use \\ for one backslash.

One Can Include Many Linefeeds

Example:

```
printf("This\ntext\\has\nlines!\n");
```

The call above prints the three lines below (at the left of the screen).

```
This
text\has
lines!
```

The next **printf** also starts on a new line (because of the linefeed at the end of the format).

Use Format Specifiers to Print Expressions

printf also prints expression values

For example, specifies what and how to print

```
printf ("Integers: %d %d %d\n",
        6 * 7, 17 + 200, 32 & 100);
```

Output: [followed by **ASCII** linefeed]

```
Integers: 42 217 32
```

The **expressions** to print

- appear **after the format specification**, and
- are **separated by commas**.

Many Format Specifiers are Supported

Format Specifier	Interpretation
`%c`	`int` or `char` as ASCII character
`%d`	`int` as decimal
`%e`	`double` as decimal scientific notation
`%f`	`double` as decimal
`%%`	one percent sign

These Tables Suffice for Our Class

Format Specifier	Interpretation
`%u`	**`unsigned int`** as decimal
`%x`	integer as lower-case hexadecimal
`%X`	integer as upper-case hexadecimal

See man pages on a lab machine for more.

Format Specifiers Print Only the Expression Values

If you want spacing, include it in the format.

Example:

```
printf("%d%d%d", 12, -34, 56);
```

prints

```
12-3456
```

Except for format specifiers and special **ASCII** characters like linefeed, **characters print exactly as they appear**.

Pitfall: Passing the Wrong Type of Expression

Be sure that your expressions (and ordering) match the format.

Example:

a `double`   an `int`

```
printf("%d %f", 10.0, 17);
```

may print (output is system dependent)

```
0 0.000000
```

A C compiler **may be able to warn you** about this kind of error.

Pitfall: Too Few/Many Expressions

If you pass more expressions than format specifiers, **the last expressions are ignored.**

If you pass fewer expressions than format specifiers, **`printf` prints … bits**!
(In other words, behavior is unspecified.)

Again, a **C** compiler **may be able to warn you** about this kind of error.

Read Input Using `scanf`

To read values from the keyboard, use `scanf`.

The “f” again means “formatted.”

`scanf` also takes
- a format in quotation marks, and
- a comma-separated list of variable addresses

memory address of variable A

Example:
```
int A;
scanf ("%d", &A);
```

reads a decimal integer, converts it to **2’s complement**, and stores the bits in **A**.

`scanf` Ignores White Space Typed by User

Example:
```
int A;
int B;
scanf ("%d%d", &A, &B);
```

The user can separate the two numbers with spaces, tabs, and/or linefeeds, such as …

```
 5 42          /* A is 5, B is 42 */

   5  /* two lines -> same result */
  42
```

The user must push <Enter> when done.

Other Characters in Format Must be Typed Exactly

If format includes characters

- other than format specifiers and white space
- user must **type them exactly** with no extra spaces. **Rarely useful**.

Example:
```
int A; int B;
scanf ("%d<>%d", &A, &B);
```

Type “`5<>42`” and `A==5`, `B==42`.

But type “`5  <>42`” and `A==5`, while `B` is unchanged (no initializer, so `B` contains bits).

Conversion Specifiers Similar to `printf`

Format Specifier	Interpretation
`%c`	store one **ASCII** character (as `char`)
`%d`	convert decimal integer to `int`
`%f`	convert decimal real number to `float`
`%lf`	convert decimal real number to `double`

Conversion Specifiers Similar to **`printf`**

Format Specifier	Interpretation
`%u`	convert decimal integer to **`unsigned int`**
`%x` or **`%X`**	convert hexadecimal integer to **`unsigned int`**

More Pitfalls for `scanf` than for `printf`

`scanf` has the same pitfalls as `printf`

- Be sure to **match format specifiers (and ordering) to variable types**.
- Be sure to **match number of specifiers to number of addresses** given.

And more!

- **Don't forget to write "&" before each variable.** (Behavior is again undefined, but can be quite difficult to find the bug.)

`printf` Returns the Number of Characters Printed

Function calls are expressions.

Both **`printf`** and **`scanf`** return **`int`**
(the calls evaluate to values of type **`int`**).

`printf` returns the number of characters printed to the display.

Writing a **`printf`** followed by a semicolon
- evaluates the expression (calls **`printf`**),
- then discards the return value.

The return value of `printf` is rarely used.

`scanf` Returns the Number of Conversions

`scanf` returns the number of conversions performed successfully, or -1 for no conversions.

The return value is **important for checking user input**.

For example,

```
if (2 != scanf ("%d%d", &A, &B)) {
    printf ("Bad input!\n");
    A = 42; B = 10; /* defaults */
}
```

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

ECE 120: Introduction to Computing

Statements in C

Remember: Statements Tell the Computer What to Do

In C, **a statement tells the computer to do something**.

There are **three types of statements**.

But statements can consist of other statements,

which can consist of other statements,

and so forth.

Many Statements are Quite Simple

Here are two of the three types…

```
;         /* a null statement */

/* A simple statement is often an
   expression and a semicolon. */
A = B;  /* simple statements */
printf ("Hello, ECE120!\n");
```

These two types **end with a semicolon** (;).

Compound Statements Consist of Other Statements

Third type: a **compound statement** consists of

- a **sequence of statements**
- **between braces**.

```
{    /* a compound statement */
     radius = 42;
     C = 2 * 3.1416 * radius;
     printf ("C = %f\n", C);
}
```

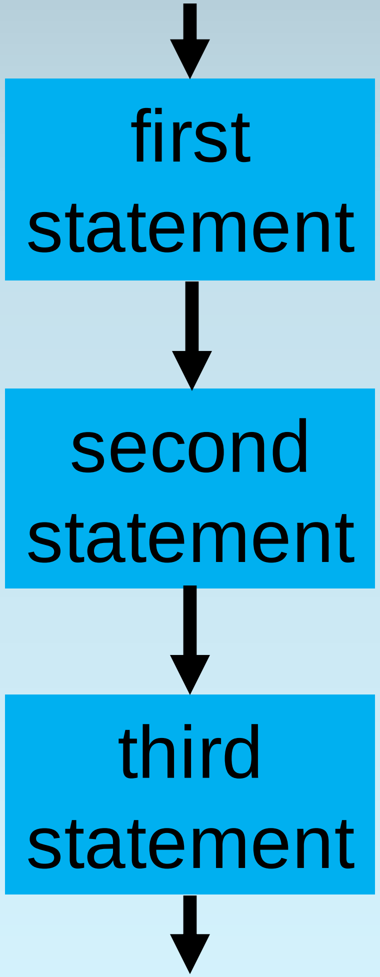
A compound statement may also contain variable declarations for use inside the statement.

A Program is a Sequence of Statements

The function body of **`main`** is a compound statement.

The function body of **`main`** thus **includes a sequence of statements**.

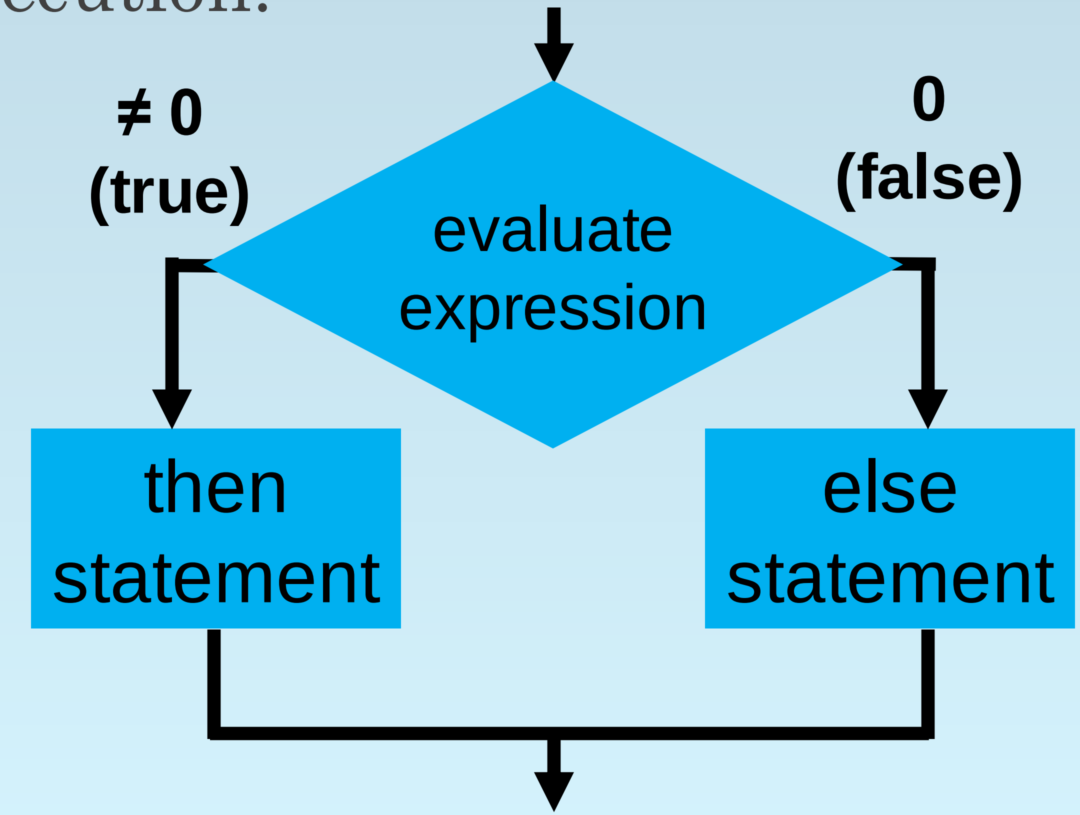
When program is started, **these statements execute in sequential order**.



Simple Statements Can Also Introduce Conditions

Simple statements in **C** can also introduce **conditional** execution.

Based on an expression, the computer executes one of two statements.



C's **`if`** Statement Enables Conditional Execution

Conditional execution uses the **`if` statement**:

```
if ( <expression> ) {
    /* <expression> != 0:
       execute "then" block */
} else {
    /* <expression> == 0:
       execute "else" block */
}
```

`<expression>` can be replaced with any expression, and **"else { … }" can be omitted**.

Examples of the `if` Statement

For example,

```
/* Calculate inverse of number. */
if (0 != number) {
    inverse = 1 / number;
} else {
    printf ("Error!\n");
}
```

Examples of the `if` Statement

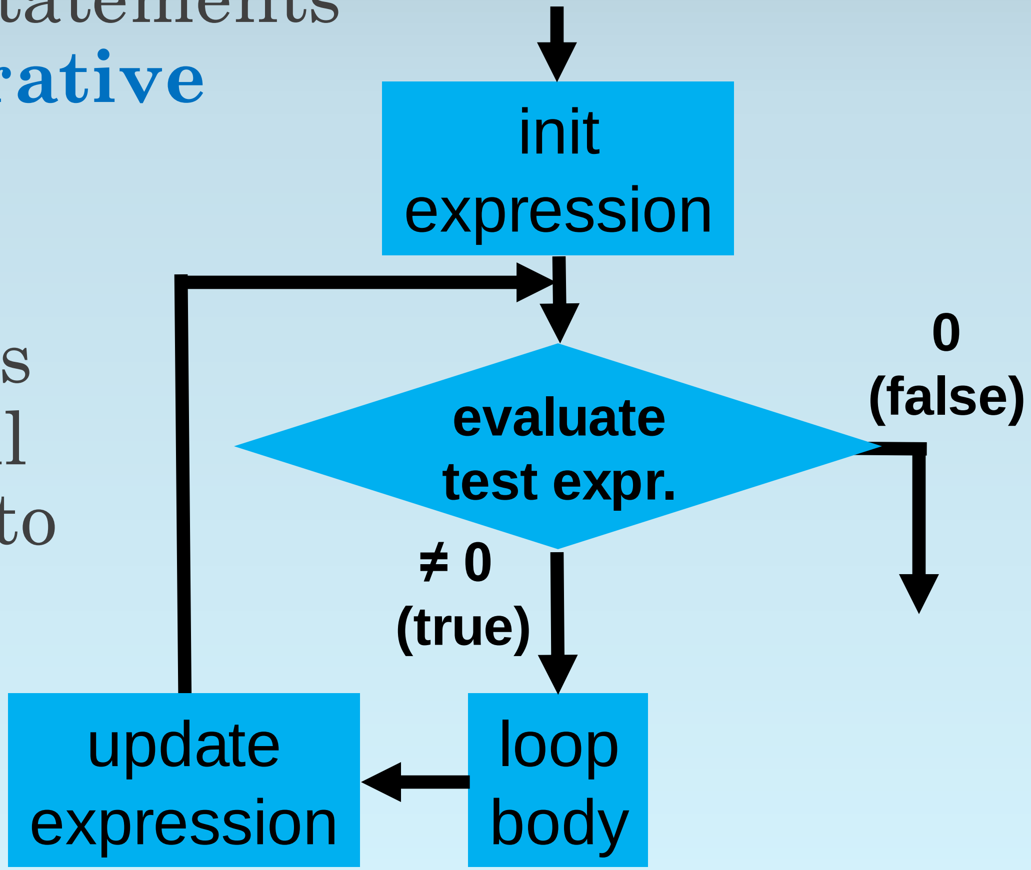
Or,

```
/* Limit size to 42. */
if (42 < size) {
    printf ("Size set to 42.\n");
    size = 42;
}
```

Simple Statements Can Also Be Iterations

Finally, simple statements can describe **iterative** execution.

This type of execution repeats a statement until a test evaluates to false (0).



C's **`for`** Loop Enables Iterative Execution

The following is called a **`for` loop**:

```
for (<init>; <test>; <update>) {
    /* loop body */
}
```

As shown on the previous slide, the computer:

1. Evaluates **`<init>`**.
2. Evaluates **`<test>`**, and stops if it is false (0).
3. Executes the **`loop body`**.
4. Evaluates **`<update>`** and returns to Step 2.

Iterations are Used for Repeated Behavior

```
/* Print multiples of 42 from
   1 to 1000. */
int N;
for (N = 1; 1000 >= N; N = N + 1) {
    if (0 == (N % 42)) {
        printf ("%d\n", N);
    }
}
```

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;
}
```

* *

Another Iterative Construct: the **`while`** Loop

C provides other loop constructs, but
only the `for` loop is needed for ECE120.

However, we may forget to remove
`while` loops from our example programs.

A **`while`** loop

- only specifies a **`<test>`** and a **`loop body`**,
- but is otherwise equivalent to a for loop.

```
while (<test>) {
    /* loop body */
}
```

* *

Easy to Map **while** Loop into **for** Loop

```
while (<test>) {
     /* loop body */
}
```

is completely equivalent to
(with empty **<init>** and **<update>**):

```
for ( ; <test>; ) {
     /* loop body */
}
```

* *

Execution of a `while` Loop

How does the computer execute a **`while`** loop?

```
while (<test>) {
    /* loop body */
}
```

We can simplify the rules for a **`for`** loop…

1. ~~Evaluates `<init>`~~. Skip this step.
2. Evaluates **`<test>`**, and stops if it is false (0).
3. Executes the **`loop body`**.
4. ~~Evaluates `<update>`~~ and returns to Step 2.

Skip this part.