

C / C++ and Unix Programming

Materials adapted from Dan Hood and Dianna Xu

C and Unix Programming

- Today's goals
 - History of C
 - Basic types
 - **printf**
 - Arithmetic operations, types and casting
 - Intro to linux

UNIX History

- The UNIX operating system was born in the late 1960s. It originally began as a one man project led by Ken Thompson of Bell Labs, and has since grown to become the most widely used operating system.
- In the time since UNIX was first developed, it has gone through many different generations and even mutations.
 - Some differ substantially from the original version, like Berkeley Software Distribution (BSD) or Linux.
 - Others, still contain major portions that are based on the original source code.
- An interesting and rather up-to-date timeline of these variations of UNIX can be found at

<http://www.levenez.com/unix/history.html>.

General Characteristics of UNIX as an Operating System (OS)

- **Multi-user & Multi-tasking** - most versions of UNIX are capable of allowing multiple users to log onto the system, and have each run multiple tasks. This is standard for most modern OSs.
- **Over 40 Years Old** - UNIX is over 40 years old and its popularity and use is still high. Over these years, many variations have spawned off and many have died off, but most modern UNIX systems can be traced back to the original versions. It has endured the test of time. For reference, Windows at best is half as old (Windows 1.0 was released in the mid 80s, but it was not stable or very complete until the 3.x family, which was released in the early 90s).
- **Large Number of Applications** – there are an enormous amount of applications available for UNIX operating systems. They range from commercial applications such as CAD, Maya, WordPerfect, to many free applications.
- **Free Applications and Even a Free Operating System** - of all of the applications available under UNIX, many of them are free. The compilers and interpreters that we use in most of the programming courses here can be downloaded free of charge. Most of the development that we do in programming courses is done under the Linux OS.
- **Less Resource Intensive** - in general, most UNIX installations tend to be much less demanding on system resources. In many cases, the old family computer that can barely run Windows is more than sufficient to run the latest version of Linux.
- **Internet Development** - Much of the backbone of the Internet is run by UNIX servers. Many of the more general web servers run UNIX with the Apache web server - another free application.

The C Language

- Currently one of the most commonly-used programming languages
- “High-level assembly”
- Small, terse but powerful
- Very portable: compiler exists for virtually every processor
- Produces efficient code
- It is at once loved and hated

History of C

- Developed during 1969-73 in the bell labs
- C is a by product of **Unix**
- C is mostly credited to
Dennis Ritchie
- Evolved from B, which evolved from BCPL



History of C

- Original machine (DEC PDP-11) was very small
 - 24k bytes of memory,
 - 12k used for operating systems
- When I say small, I mean memory size, not actual size.



Why is C Dangerous

- C's small, unambitious feature set is an advantage and disadvantage
- The price of C's flexibility
- C does not, in general, try to protect a programmer from his/her mistakes
- The International Obfuscated C Code Contest's (<http://www.ioccc.org/>) 1995 [winning entry](#)

Programming Process

- Source code must carry extension .c
- But may be named with any valid Unix file name
 - Example: **01-helloworld.c**



Example program filename convention in this course

Example

```
/* helloworld.c,  
   Displays a message */
```

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

```
int main() {
```

```
    printf("Hello, world!\n");
```

```
    return 0;
```

```
}
```

```
helloworld.c
```

Hello World in C

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

Preprocessor used to share information among source files

Similar to Java's import

```
int main() {  
    printf("Hello, world!\n");  
    return 0;  
}
```



Hello World in C

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

Program mostly a collection of functions

“main” function special: the entry point

“int” qualifier indicates function returns an integer

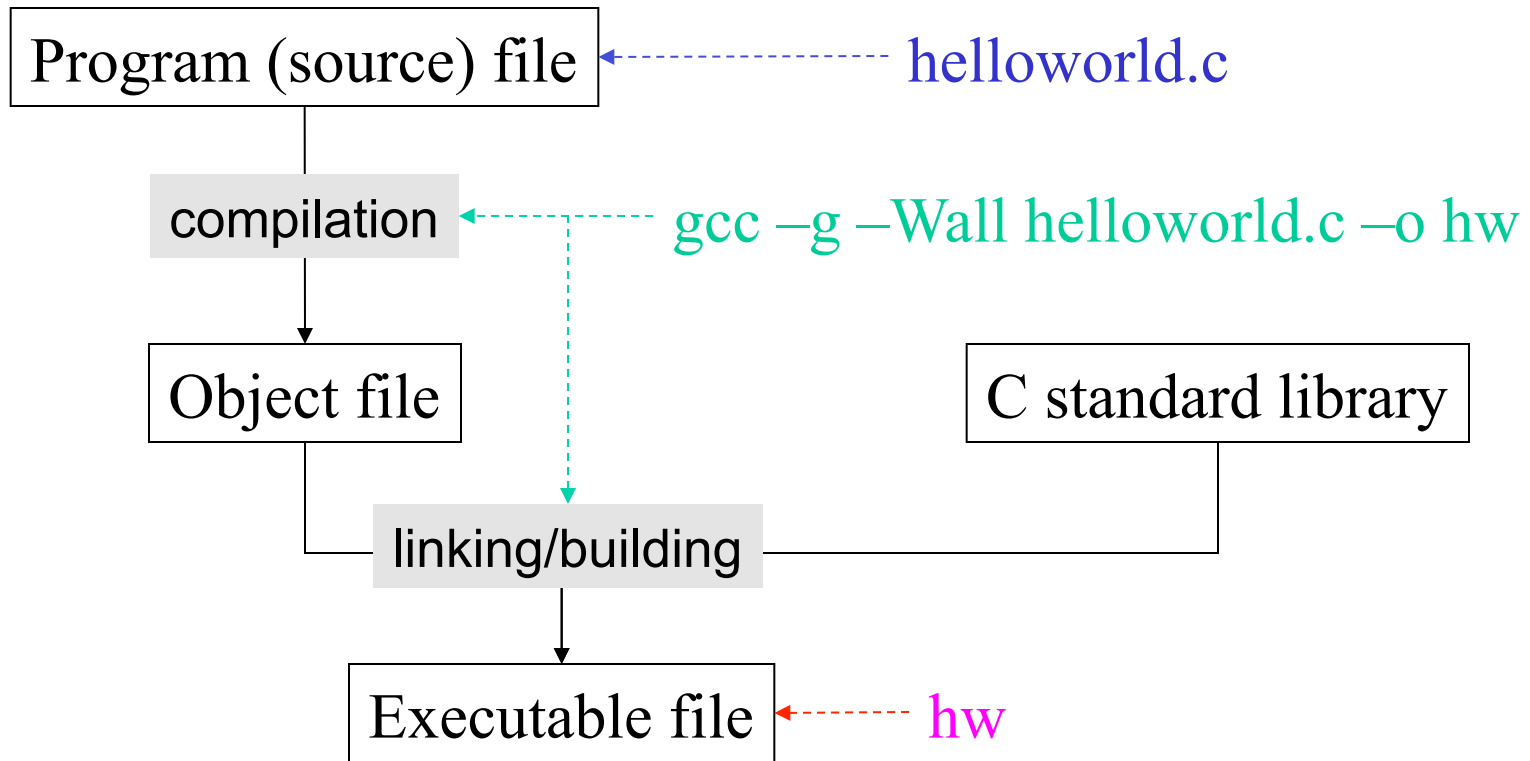
```
int main() {  
    printf("Hello, world!\n");  
    return 0;  
}
```

I/O performed by a library function

The Compiler

- gcc (Gnu C Compiler)
- gcc -g -Wall helloworld.c -o hw
- gcc flags
 - -g (produce debugging info for gdb)
 - -Wall (print warnings for all events)
 - -o filename (name output file with filename, default is a.out)

Programming Process Summary



All this is done under Unix

C Program Style

- Case sensitive
- Ignores blanks
- Comments
 1. Ignored between `/*` and `*/`
 2. Comments are integral to good programming!
- All local variables must be declared in the beginning of a function !!!

Data Types

- Integer
 - C keyword: **int, short, long**
 - Range: typically 32-bit (± 2 billion), 16-bit, 64-bit
- Floating-point number
 - C keyword: **float, double** In general, use double
 - Range: 32-bit ($\pm 10^{38}$), 64-bit
 - Examples: **0.67f, 123.45f, 1.2E-6f,**
0.67, 123.45, 1.2E-6

Variables and Basic Operations

- Declaration (identify variables and type)

```
int x;
```

```
int y, z;
```

- Assignment (value setting)

```
x = 1;
```

```
y = value-returning-expression;
```

- Reference (value retrieval)

```
y = x * 2;
```

Constants

- Integer
 - `const int year = 2002;`
- Floating point number
 - `const double pi = 3.14159265;`
- Constants are variables whose initial value can not be changed.
- Comparable to `static final`

Output Functions

- Output characters

```
printf("Text message\n");
```

\n for new line

- Output an integer

```
int x = 100;
```

```
printf("Value = %d\n", x);
```

Output: Value = 100

Variations

- Output a floating-point number

```
double y = 1.23;  
printf("Value = %f\n", y);
```

- Output multiple numbers

```
int x = 100;  
double y = 1.23;  
printf("x = %d, y = %f\n", x, y);
```

15 digits below decimal
(excluding trailing 0's)

Output: `x = 100, y = 1.230000`

printf Summary

```
printf ( "  " ,  ) ;
```

- Text containing special symbols
 - **%d** for an integer
 - **%f** for a floating-point number
 - **\n** for a newline
- List of variables (or expressions)
 - In the order corresponding to the % sequence

Display Problem

- Problem
 - Precision of **double**: 15 digits
 - Precision of **%f**: 6 digits below decimal
 - Cannot show all the significant digits
- Solution
 - More flexible display format possible with **printf**

% Specification

- **%i** **int, char** (to show value)
- **%d** same as above (**d** for decimal)
- **%f** **double** (floating-point)
- **%e** **double** (exponential, e.g., **1.5e3**)

Formatting

- Precision `% .#f`
- Width `%#f, %#d`
 - Note: Entire width
- Zero-padding `%0#d`
- Left-justification `%-#d`
- Various combinations of the above

Replace #
with digit(s)

Formatting Example (1)

```
%f      with 1.23456789 >1.234568<
%.10f   with 1.23456789 >1.2345678900<
%.2f    with 1.23456789 >1.23<

%d      with 12345 >12345<
%10d    with 12345 >12345<
%2d     with 12345 >12345<

%f      with 1.23456789 >1.234568<
%8.2f   with 1.23456789 >1.23<
```

Formatting Example (2)

```
%d:%d          with 1 and 5 >1:5<
%02d:%02d      with 1 and 5 >01:05<

%10d   with 12345 >      12345<
%-10d  with 12345 >12345  <
```

```
11-formatting.c
```


Arithmetic Operators

- Unary: $+$, $-$ (signs)
- Binary: $+$, $-$, $*$ (multiplication),
 $/$ (division), $\%$ (modulus, int remainder)
- Parentheses: $($ and $)$ must always match.
 - Good: (x) , $(x - (y - 1)) \% 2$
 - Bad: $(x,) x ($

Types and Casting

- Choose types carefully
- An arithmetic operation requires that the two values are of the same type
- For an expression that involves two different types, the compiler will **cast** the smaller type to the larger type
- Example: $4 * 1.5 = 6.0$

Mixing Data Types

- **int** values only \Rightarrow **int**
 - **4** / **2** \Rightarrow **2**
 - **3** / **2** \Rightarrow **1**
 - **int x = 3, y = 2;**
x / **y** \Rightarrow **1** 
- Involving a **double** value \Rightarrow **double**
 - **3.0** / **2** \Rightarrow **1.5**

Assignment of Values

- `int x;`

- `x = 1;`

- `x = 1.5; /* x is 1 */`

warning

- `double y;`

- `y = 1; /* y is 1.0 */`

- `y = 1.5;`

- `y = 3 / 2; /* y is 1.0 */`

`int evaluation; warning`

Example

```
int i, j, k, l;  
double f;
```

mixingtypes.c

```
i = 3;  
j = 2;  
k = i / j;  
printf("k = %d\n", k);
```

```
f = 1.5;  
l = f;          /* warning */  
printf("l = %d\n", l); /* truncated */
```

sizeof and Type Conversions

- **sizeof (type)**

- The sizeof operator returns the number of bytes required to store the given type

Implicit conversions

- arithmetic
- assignment
- function parameters
- function return type
- **promotion if possible**

Explicit conversions

- casting

```
int x;
```

```
x = (int) 4.0;
```

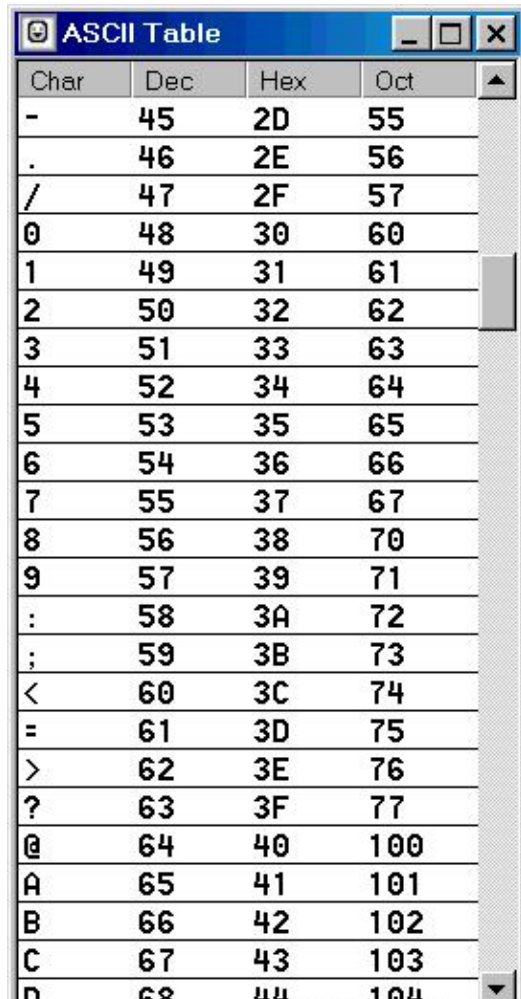

Use of **char** (character)

- Basic operations
 - Declaration: **char c;**
 - Assignment: **c = 'a';**
 - Reference: **c = c + 1;**
- Constants
 - Single-quoted character (only one)
 - Special characters: **'\n'**, **'\t'** (tab), **'\"'** (double quote), **'\''** (single quote), **'\'** (backslash)

Characters are Integers

- A **char** type represents an integer value from 0 to 255 (1 byte) or -128 to 127.
- A single quoted character is called a “character constant”.
- C characters use ASCII representation:
 - **'A' = 65 ... 'Z' = 'A' + 25 = 90**
 - **'a' = 97 ... 'z' = 'a' + 25 = 122**
 - **'0' != 0 (48), '9' - '0' = 9**
- **Never make assumptions of char values**
 - Always write **'A'** instead of 65

ASCII Table



Char	Dec	Hex	Oct
-	45	2D	55
.	46	2E	56
/	47	2F	57
0	48	30	60
1	49	31	61
2	50	32	62
3	51	33	63
4	52	34	64
5	53	35	65
6	54	36	66
7	55	37	67
8	56	38	70
9	57	39	71
:	58	3A	72
;	59	3B	73
<	60	3C	74
=	61	3D	75
>	62	3E	76
?	63	3F	77
@	64	40	100
A	65	41	101
B	66	42	102
C	67	43	103
D	68	44	104

American Standard Code
for Information Interchange
A standard way of
representing the alphabet,
numbers, and symbols
(in computers)

[wikipedia on ASCII](https://en.wikipedia.org/wiki/ASCII)

char Input/Output

- Input
 - **char getchar()** receives/returns a character
 - Built-in function
- Output
 - **printf** with **%c** specification

```
int main() {  
    char c;  
    c = getchar();  
    printf("Character >%c< has the value %d.\n", c, c);  
    return 0;  
}
```

chartypes.c

scanf Function

`scanf (" ",);`

- **Format string containing special symbols**
 - `%d` for **int**
 - `%f` for **float**
 - `%lf` for **double**
 - `%c` for **char**
 - `\n` for a newline
- **List of variables (or expressions)**
 - In the order corresponding to the `%` sequence

`scanf` Function

- The function `scanf` is the input analog of `printf`
- Each variable in the list **MUST** be prefixed with an **&**.
- Ignores white spaces unless format string contains **%c**

scanf Function

```
int main() {  
    int x;  
  
    printf("Enter a value:\n");  
    scanf("%d", &x);  
    printf("The value is %d.\n",  
x);  
    return 0;  
}
```

scanf with multiple variables

```
int main() {  
    int x;  
    char c;  
    printf("Enter an int and a char:");  
    scanf("%d %c", &x, &c);  
    printf("The values are %d, %c.\n",  
          x, c);  
    return 0;  
}
```

scanf.c

scanf Function

- Each variable in the list **MUST** be prefixed with an **&**.
- Read from standard input (the keyboard) and tries to match the input with the specified pattern, one by one.
- If successful, the variable is updated; otherwise, no change in the variable.
- The process stops as soon as **scanf** exhausts its format string, or matching fails.
- Returns the number of successful matches.

scanf Continued

- White space in the format string match any amount of white space, including none, in the input.
- Leftover input characters, if any, including one ‘\n’ remain in the input buffer, may be passed onto the next input function.
 - Use **getchar()** to consume extra characters
 - If the next input function is also **scanf**, it will ignore ‘\n’ (and any white spaces).

`scanf` Notes

- Beware of combining `scanf` and `getchar()`.
- Use of multiple specifications can be both convenient and tricky.
 - Experiment!
- Remember to use the return value for error checking.

if-else Statement

```
int main() {  
    int choice;  
    scanf("%d", &choice); //user input  
  
    if (choice == 1) {  
        printf("The choice was 1.\n");  
    }  
    else {  
        printf("The choice wasn't 1.\n");  
    }  
    return 0;  
}
```

menu.c

Expressions

- Numeric constants and variables
E.g., **1**, **1.23**, **x**
- Value-returning functions
E.g., **getchar()**
- Expressions connected by an *operator*
E.g., **1 + 2**, **x * 2**, **getchar() - 1**
- All expressions have a type

Boolean Expressions

- C does not have type boolean
- False is represented by integer 0
- Any expression evaluates to non-zero is considered true
- True is typically represented by 1 however

Conditional Expressions

- Equality/Inequality

- **if** (**x** **==** 1)

- **if** (**x** **!=** 1) **≠**



== (equality)

= (assignment)

- Relation

- **if** (**x** **>** 0) **>**

- **if** (**x** **>=** 0) **≥**

- **if** (**x** **<** 0) **<**

- **if** (**x** **<=** 0) **≤**

The values are internally represented as integer.
true → 1 (not 0), false → 0

Assignment as Expression

- Assignment
 - Assignments are expressions
 - Evaluates to value being assigned

- Example

```
int x = 1, y = 2, z = 3;
```

```
x = (y = z);
```

3 ← **3 ← 3**
evaluates to 3

evaluates to 3 (true)



```
if (x = 3) {  
    ...  
}
```


Complex Condition

- And

```
if ( (x > 0) && (x <= 10) )
```

$0 < x \leq 10$

- Or

```
if ( (x > 10) || (x < -10) )
```

$|x| > 10$

- Negation

```
if ( ! (x > 0) )
```

$\text{not } (x > 0) \Leftrightarrow x \leq 0$

Beware that **&** and **|** are also C operators

Lazy Logical Operator Evaluation

- If the conditions are sufficient to evaluate the entire expression, the evaluation terminates at that point => **lazy**



- Examples

▫ `if ((x > 0) && (x <= 10))`

Terminates if `(x > 0)` fails

▫ `if ((x > 10) && (x < 20)) || (x < -10))`

Terminates if `(x > 10) && (x < 20)` succeeds

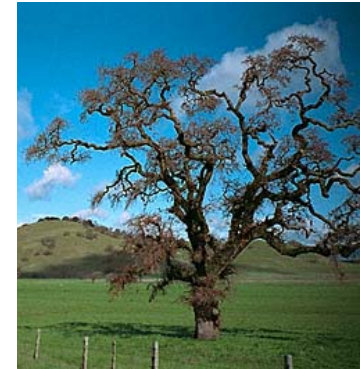
Use of Braces

```
if (choice == 1) {  
    printf("1\n");  
}  
else {  
    printf("Other\n");  
}
```

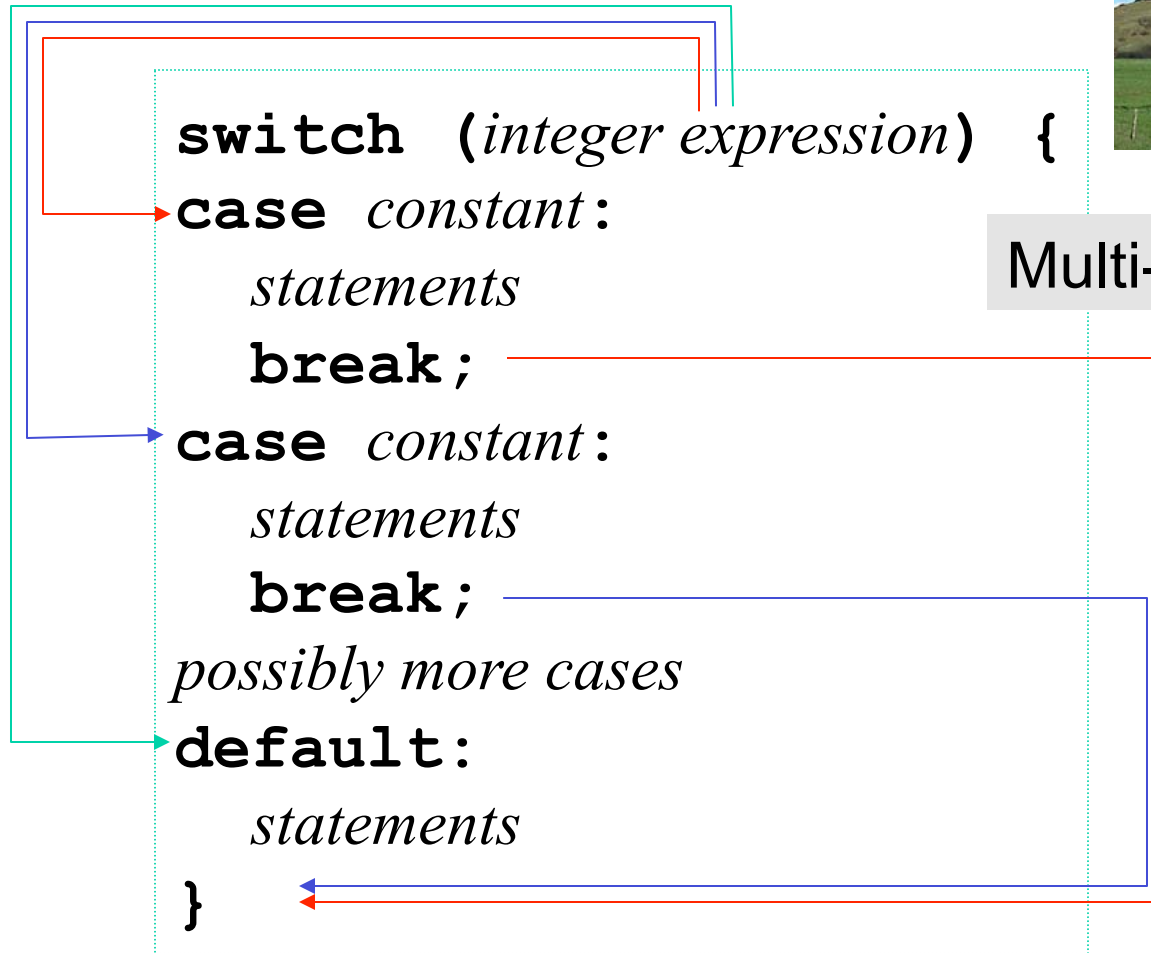
```
if (choice == 1)  
    printf("1\n");  
else  
    printf("Other\n");
```

When the operation is a single statement, '{' and '}' *can* be omitted.

switch Statement



Multi-branching



break Fall Through

- Omitting **break** in a **switch** statement will cause program control to fall through to the next case
- Can be a very convenient feature
- Also generates very subtle bugs
- **switch** statements only test equality with integers

Example

```
int x, y, result = 0; scanf("%d %d", &x, &y);
switch(x) {
    case 1: break;
    case 2:
    case 3: result = 100;
    case 4:
        switch(y) {
            case 5: result += 200; break;
            default: result = -200; break;
        }
    break;
    default: result = 400; break;
}
```

while Loops

```
while (true) {  
    /* some operation */  
}
```

while and Character Input

- **EOF** is a constant defined in `stdio.h`
 - Stands for End Of File

```
int main() {  
    int nc = 0, nl = 0; char c;  
    while ((c = getchar()) != EOF) {  
        nc++;  
        if (c == '\n') nl++;  
    }  
    printf("Number of chars is %d and number of  
        lines is %d\n", nc, nl);  
    return 0;  
}
```

charloop.c

Review: Assignment has value

- In C, assignment expression has a value, which is the value of the lefthand side after assignment.
- Parens in `(c = getchar()) != EOF` are necessary.
- `c = getchar() != EOF` is equivalent to
`c = (getchar() != EOF)`
- `c` gets assigned 0 or 1.

Summary

- C and Java's conditionals and loops are very similar
- C does not support booleans, uses 0 and 1 (not 0) instead
- Learn how to use **scanf** and **getchar**, especially with input loops
- Learn how C handles characters
- Programming style is important!