# C Fundamentals and Console I/O

ITCS 2116: C Programming

College of Computing and Informatics

Department of Computer Science

# Outline

- C Coding Style
- Platform Independence?
- C Compilation Steps
- **`gcc`**
- C99 and C89
- Console I/O
- Streams
- Character I/O
- **`printf`**

# C Coding Style (Conventions)

- Universal agreement
  1. clarity and consistency are very important
  2. indentation, white space, and comments helpful
  3. consistent naming conventions helpful
- See the Style Guidelines for ITCS 2116 in *Canvas*
- Tools (intelligent editors, *indent*, etc.) will take care of much formatting for you.

# Does it Matter?

Consider the following entries from the International Obfuscated C Code (IOCC) Contest…

> **ob·fus·cate**: render obscure, unclear, or unintelligible: *the spelling changes will deform some familiar words and obfuscate their etymological origins*.

```
#include\
                                <stdio.h>
                    #include                <stdlib.h>
                    #include                <string.h>

                   #define w "Hk~HdA=Jk|Jk~LSyL[{M[wMcxNksNss:"
                  #define r"Ht@H|@=HdJHtJHdYHtY:HtFHtF=JDBIl"\
                 "DJTEJDFIlMIlM:HdMHdM=I|KIlMJTOJDOIlWITY:8Y"
                #define S"IT@I\\@=HdHHtGH|KILJJDIJDH:H|KID"\
               "K=HdQHtPH|TIDRJDRJDQ:JC?JK?=JDRJLRI|UItU:8T"
              #define _(i,j)L[i=2*T[j,O[i=O[j-R[j,T[i=2*\
             R[j-5*T[j+4*O[j-L[j,R[i=3*T[j-R[j-3*O[j+L[j,
            #define t"IS?I\\@=HdGHtGIDJILIJDIItHJTFJDF:8J"

    #define y                yy(4),yy(5),              yy(6),yy(7)
  #define yy(              i)R[i]=T[i],T[i ]           =O[i],O[i]=L [i]
#define Y _(0          ], 4] )_ (1 ], 5] )_ (2      ], 6] )_ (3 ], 7] )_=1
#define v(i)(      (( R[ i ] * _ + T [ i ]) * _ + O [ i ]) * _ + L [ i ]) *2
double b = 32  ,l ,k ,o ,B ,_ ; int Q , s , V , R [8 ], T[ 8] ,O [8 ], L[ 8] ;
#define q( Q,R ) R= *X ++ % 64 *8 ,R |= *X /8 &7 ,Q=*X++%8,Q=Q*64+*X++%64-256,
# define  p      "G\\QG\\P=GLPGTPGdMGdNGtOGlOG"   "dSGdRGDPGLPG\\LG\\LHtGHtH:"
#  define W         "Hs?H{?=HdGH|FI\\II\\GJlHJ"    "lFL\\DLTCMlAM\\@Ns}Nk|:8G"
# define   U           "EDGEDH=EtCElDH{~H|AJk}"       "Jk?LSzL[|M[wMcxNksNst:"
#  define u                  "Hs?H|@=HdFHtEI"            "\\HI\\FJLHJTD:8H"
char  *   x                   ,*X , ( * i )[              640],z[3]="4_",
*Z = "4,8O4.8O4G" r U "4M"u S"4R"u t"4S8CHdDH|E=HtAIDAIt@IlAJTCJDCIlKI\\K:8K"U
 "4TDdWDdW=D\\UD\\VF\\FFdHGtCGtEIDBIDDIlBIdDJT@JLC:8D"t"4UGDNG\\L=GDJGLKHL\
FHLGHtEHtE:"p"4ZFDTFLT=G|EGlHITBH|DIlDIdE:HtMH|M=JDBJLDKLAKDALDFKtFKdMK\
\\LJTOJ\\NJTMJTM:8M4aGtFGlG=G|HG|H:G\\IG\\J=G|IG|I:GdKGlL=G|JG|J:4b"W
S"4d"W t t"4g"r w"4iGlIGlK=G|JG|J:4kHl@Ht@=HdDHtCHdPH|P:HdDHdD=It\
BIlDJTEJDFIdNI\\N:8N"w"4lID@IL@=HlIH|FHlPH|NHt^H|^:H|MH|N=J\\D\
J\\GK\\OKTOKDXJtXItZI|YIlWI|V:8^4mHLGH\\G=HLVH\\V:4n" u t t
"4p"W"IT@I\\@=HdHHtGIDKILIJLGJLG:JK?JK?=JDGJLGI|MJDL:8M4\
rHt@H|@=HtDH|BJdLJTH:ITEI\\E=ILPILNNtCNlB:8N4t"W t"4u"
p"4zI[?Il@=HlHH|HIDLILIJDII|HKDAJ|A:JtCJtC=JdLJtJL\
THLdFNk|Nc|\
:8K"; main (
int C,char**        A) {for(x=A[1],i=calloc(strlen(x)+2,163840);
C-1;C<3?Q=_=       0,(z[1]=*x++)?((*x++==104?z[1]^=32:--x), X =
strstr(Z,z))      &&(X+=C++):(printf("P2 %d 320 4 ",V=b/2+32),
V*=2,s=Q=0,C     =4):C<4?Q-->0?i[(int)((l+=o)+b)][(int)(k+=B)
]=1:_?_-=.5/    256,o=(v(2)-(l=v(0)))/(Q=16),B=(v(3)-(k=v(1)
))/Q:*X>60?y    ,q(L[4],L[5])q(L[6],L[7])*X-61||(++X,y,y,y),
Y:*X>57?++X,  y,Y:*X >54?++X,b+=*X++%64*4:--C:printf("%d "
,i[Q][s]+i[Q ][s+1]+i[Q+1][s]+i[Q+1][s+1])&&(Q+=2)<V||(Q=
0,s+=2)<640
||(C=1));}
```

What is the purpose of this program?

```
[I++]=i):0:0:0;D(
);}if(I<1)c(C("
bad\40la"yout"))
for(i=0;256>(R=i)
;n[i++]=R)for(A=8
;A>0;A--
)R=((R&1)==0)?(un
signed
int)R>>(01):((uns
igned/*kero
Q',KSS
*/)R>>1)^0xedb883
20;m=a[I-
1];a[I]=(m
<N)?(m=N+8):++m;f
or(i=00;i<I;e[i++
]=0){
```

# Ex.: Some GNOME Project Guidelines

- “Programmers should strive to write good code so that it is easy to understand and modify by others
- Important qualities of good code
  - clarity
  - consistency
  - extensibility
  - correctness”

# Example… (cont'd)

- "It is important to follow a good naming convention for the symbols in your programs
  - Function names should be of the form **`module_submodule_operation`**, for example, **`gnome_canvas_set_scroll_region`**
  - Symbols should have descriptive names: do not use **`cntusr()`**, use **`count_active_users()`** instead
  - Function names are lowercase, with underscores to separate words, like this: **`gnome_canvas_set_scroll_region()`**"

# Example… (cont'd)

- "Macros and enumerations are uppercase, with underscores to separate words, like this: **`GNOMEUIINFO_SUBTREE()`** for a macro
- Typedefs and structure names are mixed upper and lowercase, like this: **`GnomeCanvasItem`**, **`GnomeIconList`**"

# Example… (cont'd)

- "Very short and terse names should only be used for the local variables of functions; never call a global variable **x**; use a longer name that tells what it does"

# Example from Linux Guidelines

- “Tabs are 8 characters, and indentations too
- Put the opening brace last on the line, and put the closing brace first:

```
if (x is true) {
    we do y
}
```

- Functions have the opening brace at the beginning of the next line:

```
int function(int x)
{
    body of function
}”
```

# Our Guidelines! (These Matter!)

- Make sure to include file level comments in all programs
  - Author(s) name and UNC Charlotte email address(s)
  - Briefly describe the purpose of program or module within program
- Use function comments
  - Function's purpose
  - Inputs (global or parameters)
  - Outputs (return values and side effects)
  - Pre-conditions
  - Post-conditions (including side effects)

# Our Guidelines! (These Matter!)

- Global Variables
  - Describe purpose
- Magic Numbers
  - Use `#define` except for obvious numbers (-1, 0, 1, 2)
    - Unless those numbers have a specific named purpose or are an exit code!!!
    - We cover `#define` in more detail in a later module.

# Our Guidelines! (These Matter!)

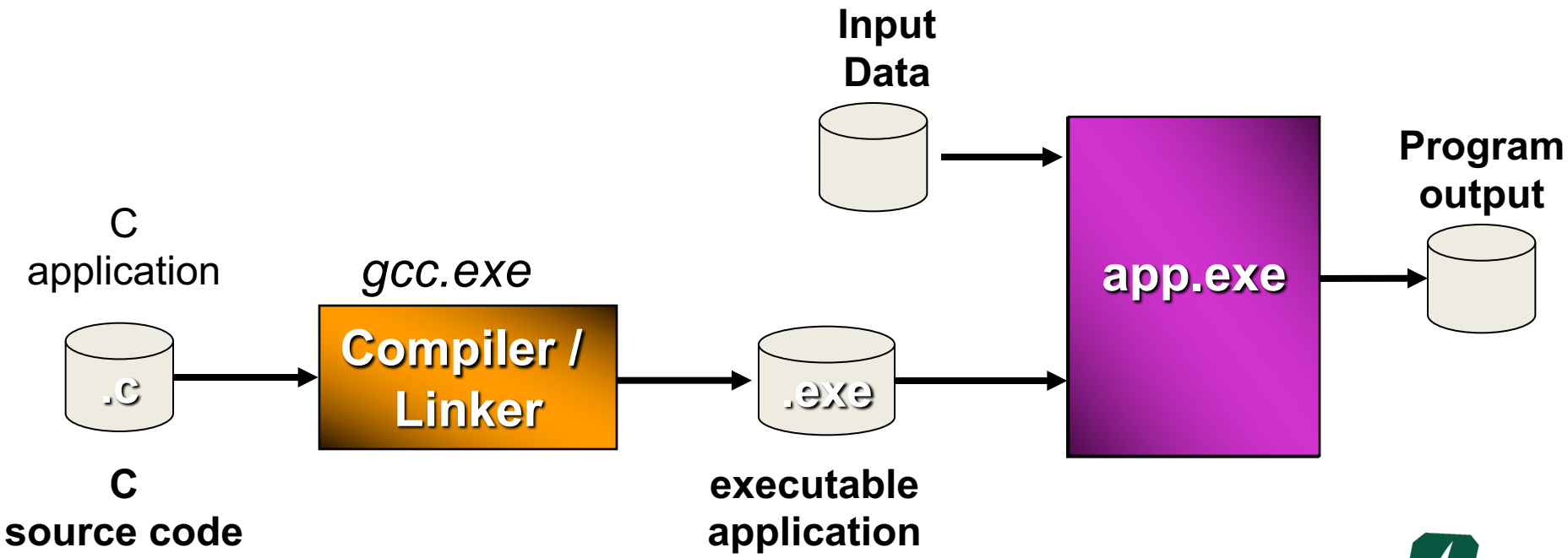
- Indentation
  - All indentation must be spaces (except for Makefiles)
  - The number of spaces for indentation must be consistent
    - 2 to 3 spaces
  - Indent:
    - Statements in a function
    - Statements in a control structure
    - Statements in a block { }

# Our Guidelines! (These Matter!)

- Curly Braces
  - Functions – opening curly brace on next line
  - Everything else – opening curly brace at end of control structure
- Statements
  - 1 statement per line

# Executing C Programs

1. High-Level Language (HLL) source code is **compiled** into the instruction set of the target computer
2. This code is loaded and **executed directly** by the host



# Platform Independence?

- Compiled
  - parts of the compiler (*front end*) are platform-independent
  - parts of the compiler (*back end*) are specific to the platform on which the program will be executed
- Interpreted
  - the Java compiler is platform-independent
  - the Java Virtual Machine (JVM) is platform-specific

# Steps in Compiling C Programs

Source Code

```
#define N 3
a=c+b*N;
```

***preprocessing***

Expanded Source Code

```
a=c+b*3;
```

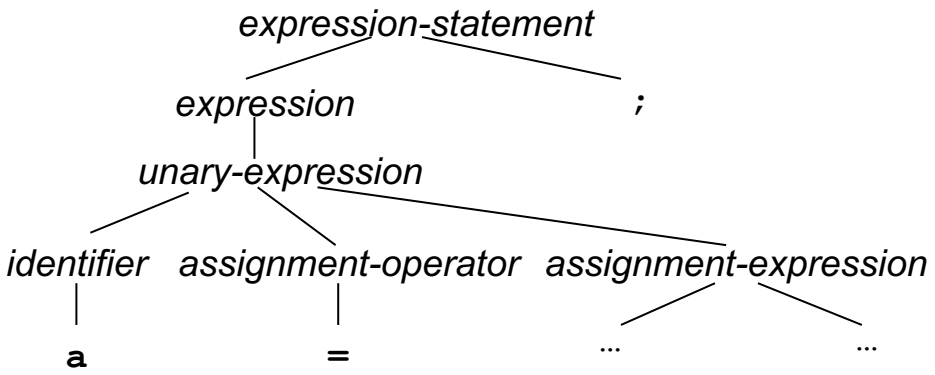
***lexical analysis***

Tokens

```
a   =   c   +   b   *   3   ;
```

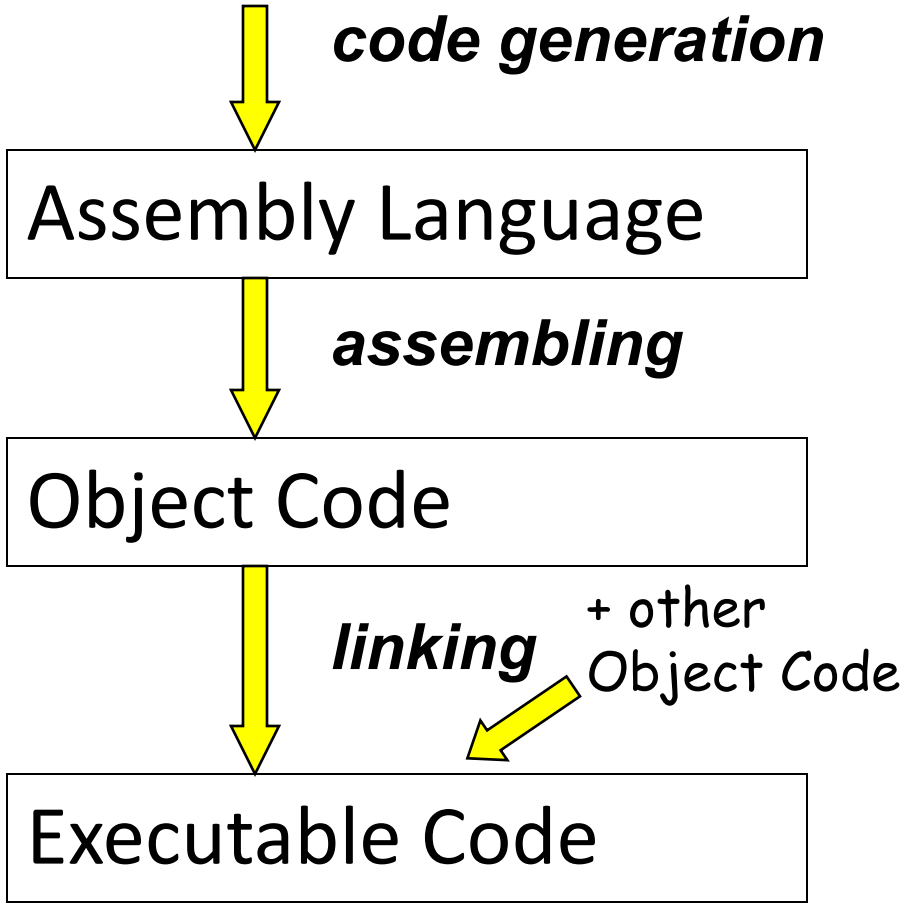
***parsing***

Parse Tree




***code generation***

# Steps… (cont'd)




```
mov ebx, b
imul ebx, ebx, 3
mov ecx, c
```

```
001110010111
```

```
00111001011101101010…
```

# Using the `gcc` Compiler

- **`gcc`** is a high-quality, open source compiler available for most platforms
- At the command prompt, type

```
gcc  -Wall -std=c99 pgm.c
```

where *pgm.c* is the C program source file

  - Creates an executable **`a.out`**
- **`-std=c99`** specifies that C99 standard features are allowed
- **`-Wall`** turns on all the important **warning messages**

# Compiler… (cont'd)

- GNOME (and me): "Make sure your code compiles with absolutely no warnings from the compiler. These help you catch stupid bugs."

# Some Useful `gcc` Options

| | |
|---|---|
| `-c` | Compile the source code but do not link (i.e., produce only the object file) |
| `-E` | Preprocess the source code only (i.e., expand macros, but do not compile the source code) |
| `-o` ***`file`*** | Put output in file named **file** |
| `--version` | Display version number of gcc |
| `-std=c99` | Support C99 language features |
| `-Wall` | Enable all warnings |
| `-g` | Produce information necessary to debug using **gdb** |

# `gcc` options… (cont'd)

| | |
|---|---|
| `-O, -O1` | Various optimization levels |
| `-D` ***`name`*** | Define name as a macro with value 1 (used for conditional compilation) |
| `-l`***`lib`*** | Search named **`library`** when linking |
| `-I`***`dir`*** | Add directory **`dir`** to the head of the list of directories to search for header files |
| `-L`***`dir`*** | Add directory **`dir`** to the list of directories to search for libraries containing object files (specified using the `-l` option) |

# A Word About C99

- The generations of C
  - K&R C
  - C89 (or C90)
  - **C99**
  - C11 to C17

} ISO standards

- **We will use C99 in this course**
  - For the most part, C99 adds to / clarifies earlier versions, does not invalidate earlier code.
  - The latest standard in wide use is C11. You may use its features in your code if you wish.

# *(Some) Differences C89↔C99*

1. Comments allowed to be C++ style (`//`)
2. `_Bool` macro is available
3. Additional library functions, and a few new header files
4. Variable length arrays
5. Variable declarations can appear anywhere in the code block
6. Variable declarations in `for` loops
7. Support for non-ASCII character sets (“wide” characters)

# *(Some) Differences… (cont'd)*

8. New `long long` integer data type
9. Functions must declare a return value
10. Macros may have variable number of arguments, denoted by ellipsis (…)
11. Functions may be inlined
12. Restricted pointers (prevent aliasing)

# *C99… (cont'd)*

- gcc 4.6.3 supports most of C99, but you may not be able to use…
  - wide characters
  - variable length arrays
  - complex numbers
  - extended integer types (`long long`)
- We will not need most of these features for this course.

# Console I/O

# What is I/O ?

- The **I** stands for **Input**, that is, the data entered by the user or read by the program from an external source.
  - External sources in C are usually referred to as *streams*. A text file and the console (terminal) are examples of streams.
- The **O** stands for **Output**, that is, the results produced by the program code.
  - Output in C is sent to a stream.
  - By default, C programs use the computer's console or terminal.
  - We will use the console and text files as the output stream.

# Console I/O in C

- I/O is provided by **standard library** functions
  - available on **all platforms**
- To use, your program must have

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

- …and it doesn't hurt to also have

```
#include <stdlib.h>
```

- *These are **preprocessor** statements; the .h files define function types, parameters, and constants from the standard library.*
  - *We will cover the preprocessor in more detail later in the course.*

# Streams

- A ***stream*** is a **file** or a **device** from which data is read, and/or to which data is written
- By **default**, every C program automatically has 3 open streams, called
  - the *standard input*
  - the *standard output*
  - the *standard error*

# Streams (cont'd)

- If you do not override them…
  - standard input means the keyboard, i.e., what the user types.
  - standard output & error means the terminal window.

# Stream (cont'd)

- Note: the **EOF** (end of file) character on your keyboard is either **`Ctrl-d`** (Unix, Linux, Mac OS X) or **`Ctrl-z`** (Windows)
- You can **redirect** the standard **input from a file**, e.g.,

```
pgm99 < infile.txt
```

- You can **redirect** the standard **output to a file**, e.g.,

```
pgm99 > outfile.txt
```

# **Reading** One Character from Standard Input

Definition (from **`stdio.h`**):

**`int getchar(void)`**

```
int c;

c = getchar();
if (c == EOF)  // check for end-of-file
    …
```

(see **`tiny_io.c`** in *Code samples and Demonstrations* in *Canvas*)

**`EOF`** is a macro that represents the **end of file character**.
It is defined in **`stdio.h`**

- Declaring **`c`** as type **`char`** and then comparing to **`EOF`** may not work

# **Writing** One Character to Standard Output

Definition (from **`stdio.h`**):

```
int putchar(int c)
```

```
char c;
int b;

…
b = putchar((int) c );
if (b == EOF)
    …
```

# Program **echochar.c**

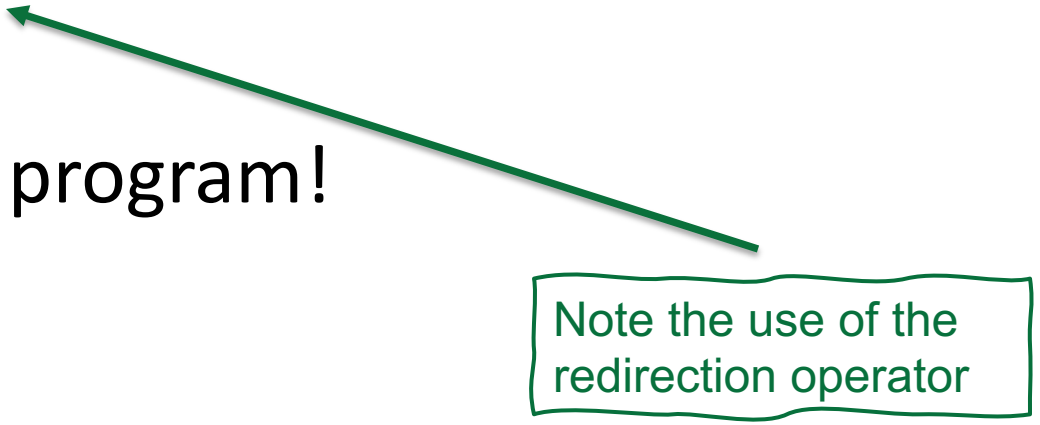
```
#include <stdio.h>

int main ( void )
{
    int c;
    c = getchar();
    while (c != '\n') {
        (void) putchar(c);
        c = getchar();
    }
    putchar('\n');

    return 0;
}
```

# Example: `echochar.c`

- Keyboard input vs. input from a file
  - use editor to type the input in a **file** called `in.txt`
  - then run `echochar` with input redirected from the file

```
% ./echochar < in.txt
```

- **No changes** to the program!



# The `printf()` function

- `putchar()` is too cumbersome to use for extensive, formatted output
- `printf()` is a much more convenient **library function** for formatted output, with built-in conversions of input parameters to printable form

# The `printf()` function

- Definition: `int printf(const char * format, …)`

Variable number of arguments

- `format` specifies how input arguments must be converted/formatted for output

# Parts of `format`

1. **%** (mandatory)
2. 0 or more **flags** (infrequently used)
3. **Minimum output field width** (pad with spaces) (useful for making things line up)
4. **.Precision** (minimum number of digits to right of decimal point)
   (optional, default is 6 digits)
5. **type of format conversion** (mandatory)

# Precision Matters

- `printf` the number 33.3:

| Format Specifier | Output |
|---|---|
| `%7.1f` | `  33.3` |
| `%14.10f` | `  33.2999992371` |
| `%.20f` | `33.29999923706054687500` |

# Some Types of Conversions

| Print as Type… | Specifier |
|---|---|
| `char` | `%c` |
| `unsigned int` | `%u` (in **decimal**)<br>`%o` (in **octal**)<br>`%x`, `%X` (in **hex**)<br>(`%lu`, `%lo`, `%lx` for long) |
| `signed int` | `%d`, `%i` (in **decimal**)<br>(`%ld`, `%li` for long) |
| `float` | `%f` |
| `float` | `%e`, `%E` (use scientific notation) |
| (string) | `%s` |

# Example

Program:

```
char c = 'a';
int i = 9999;
float f = 3.141592653589793 2;

printf("c = %c (%o in octal)\n", c, c);
printf("i = %6d (%x in hex)\n", i, i);
printf("f = %8.5f (%e in sci. notation)\n",
        f, f);
```

(see **format.c** in *Code samples and Demonstrations* in *Canvas*)

Output:

```
c = a (141 in octal)
i =   9999 (270f in hex)
f =  3.14159 (3.141593e+00 in sci. notation)
```