CSE 1320

Week of 02/06/2023

Instructor : Donna French

Variables in C

Rules of the Variable

- Must be declared
- Must be assigned a type
- Compiler reserves space in memory – amount depends on type

```
int x;
long y;
short z;
char a;
```

Variable Types in C

- Scalar types
  - enumerated
  - pointer
  - arithmetic – integer types and floating point types
- Aggregate types
- Function types
- Union types
- Void type
  - Type when a function does not return a value

The Integer Types

`int`

- scalar type
- usually equivalent to a word
- handled more efficiently than the other types in C

- Issues
  - the size of a word varies with different hardware
    - 16 bits on one computer and 32 bits on another
  - creates portability problems
  - largest value can vary

The Integer Types

`short int`	also referred to as `short`
`long int`	also referred to as `long`

- used to avoid issues with `int`
- behave like `int` with arithmetic operators
- major difference is the number of bytes used to store each value

The Integer Types

Conversion Specifications

`%ld`	a `long` in decimal
`%lo`	a `long` in octal
`%lx or %lX`	a `long` in hexadecimal
`%hd`	a `short` in decimal
`%ho`	a `short` in octal
`%hx or %hX`	a `short` in hexadecimal

The sizeof() Operator

`sizeof()`

gives the number of bytes associated with a specified type or variable

The argument to `sizeof()` can be a

- type name
- variable
- expression

sizeofDemo.c

```
printf("The sizeof(short)     is %d\n", sizeof(short));
printf("The sizeof(int)       is %d\n", sizeof(int));
printf("The sizeof(long)      is %d\n", sizeof(long));
printf("The sizeof(char)      is %d\n", sizeof(char));

printf("The sizeof(shortVar) is %d\n", sizeof(shortVar));
printf("The sizeof(intVar)   is %d\n", sizeof(intVar));
printf("The sizeof(longVar)  is %d\n\n", sizeof(longVar));

shortVar = intVar = longVar = MAX_INT;
printf("Assigning %d to shortVar, intVar, longVar\n\n", MAX_INT);

printf("The sizeof(shortVar) is %d\n", sizeof(shortVar));
printf("The sizeof(intVar)   is %d\n", sizeof(intVar));
printf("The sizeof(longVar)  is %d\n\n", sizeof(longVar));

printf("The sizeof(intVar+3/2+3*7-4)   is %d\n", sizeof(intVar+3/2+3*7-4));
```

```
short shortVar;
int   intVar;
long  longVar;
```

sizeofDemo.c

```
The sizeof(short)    is 2
The sizeof(int)      is 4
The sizeof(long)     is 8
The sizeof(char)     is 1

The sizeof(shortVar) is 2
The sizeof(intVar)   is 4
The sizeof(longVar)  is 8

Assigning 32767 to shortVar, intVar, longVar

The sizeof(shortVar) is 2
The sizeof(intVar)   is 4
The sizeof(longVar)  is 8

The sizeof(intVar+3/2+3*7-4)   is 4
```

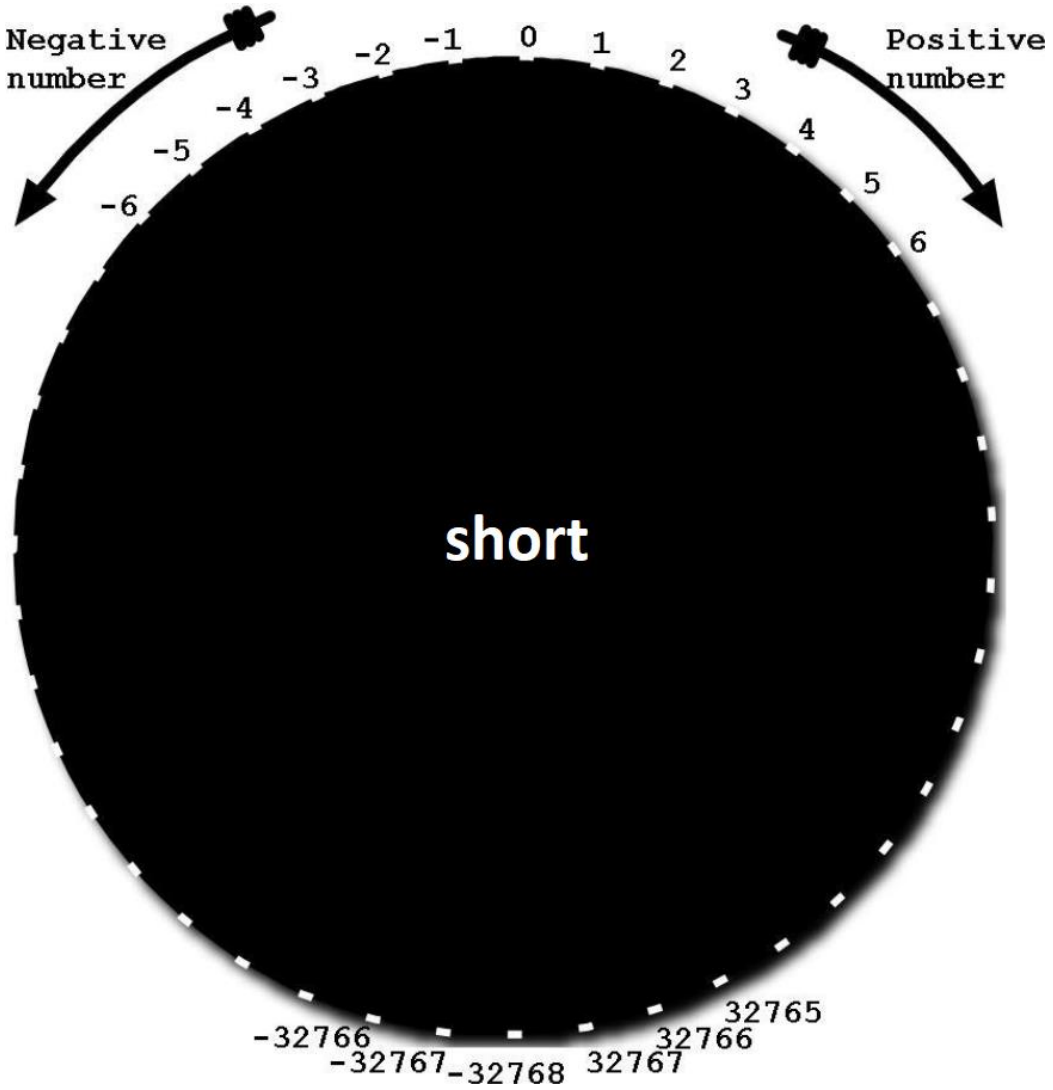
overflow2Demo.c

Overflow

When an arithmetic operation attempts to create a numeric value that is outside of the range that can represented with a given number of bits, we get

overflow

Each type has its own range

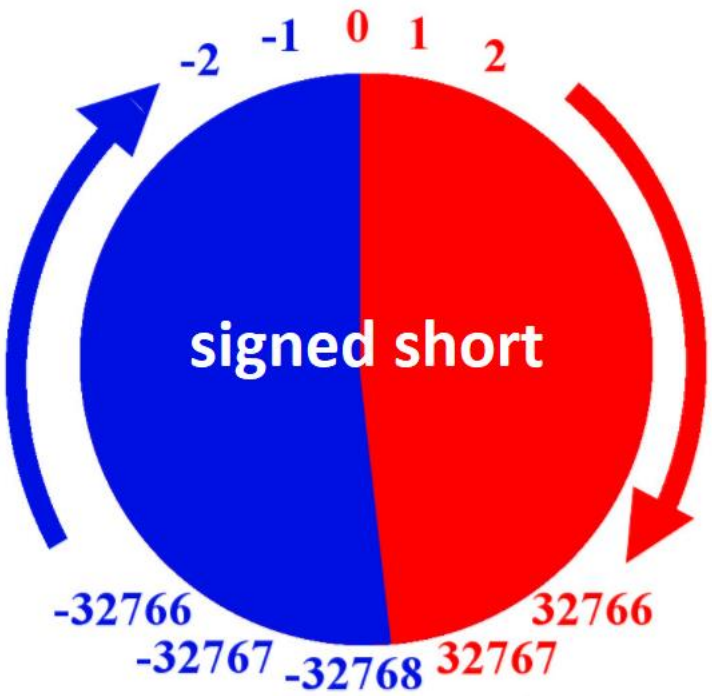


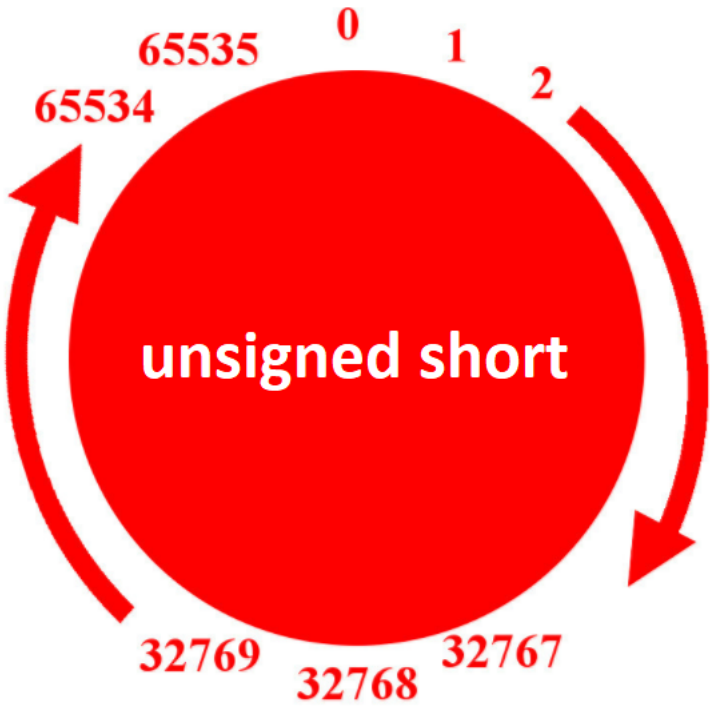
uoverflowDemo.c

Unsigned Types

```
short
int
long
```

```
unsigned short
unsigned int
unsigned long
```





```
The sizeof(short)      is 2
The sizeof(int)        is 4
The sizeof(long)       is 8

The sizeof(shortVar)  is 2
The sizeof(intVar)    is 4
The sizeof(longVar)   is 8

Assigning 32767 to
shortVar, intVar, longVar

The sizeof(shortVar)  is 2
The sizeof(intVar)    is 4
The sizeof(longVar)   is 8
```

```
The sizeof(unsigned short) is 2
The sizeof(unsigned int)   is 4
The sizeof(unsigned long)  is 8

The sizeof(ushortVar) is 2
The sizeof(uintVar)   is 4
The sizeof(ulongVar)  is 8

Assigning 65535 to
ushortVar, uintVar, ulongVar

The sizeof(ushortVar) is 2
The sizeof(uintVar)   is 4
The sizeof(ulongVar)  is 8
```

usizeofDemo.c

Unsigned Types

Conversion Specifications for `unsigned`

`%hu`	an `unsigned short` in decimal
`%u`	an `unsigned int` in decimal
`%lu`	an `unsigned long` in decimal

The `%x` (hexadecimal) and the `%o` (octal) conversion specifications indicate unsigned conversion.

ANSI C and Integer Types

limits.h

/usr/include/limits.h

Contains defines that set the sizes of integer types

```
/* Minimum and maximum values a 'signed int' can hold.  */
#  define INT_MIN        (-INT_MAX - 1)
#  define INT_MAX        2147483647

/* Maximum value an 'unsigned int' can hold.  (Minimum is 0.)  */
#  define UINT_MAX       4294967295U
```

`printf()` – field width specifier

```
printf(control_string, args, ...)
```

```
% [flag] [field width] [.precision] [size] conversion
```

field width

- optional
- a decimal integer constant specifying the minimal field width
- output will be right justified and blanks will be used to pad on the left
- will use more space than designated if more space is necessary to output expression

```c
int addend1;
int addend2;
int a;

printf("Enter first  addend ");
scanf("%d", &addend1);
printf("\nEnter second addend ");
scanf("%d", &addend2);

printf("\n\t%5d\n", addend1);
printf("\t\b+%5d\n\t", addend2);


for (a = 0; a < 5; a++)
{
   printf("=");
}

printf("\n\t%5d\n", addend1 + addend2);
```

```
Enter first  addend 12

Enter second addend 1234

            12      12
         + 1234    +1234
          =====     =====
           1246      1246


Enter first  addend 12345

Enter second addend 0

           12345      12345
         +    0     +0
          =====      =====
          12345      12345
```

Floating Point Types

- `float` – single precision
- `double` – double precision
- `long double` – extra precision

```
float floatVar = 3.14;
double doubleVar = 3.14159;
long double longdoubleVar = 3.1415926535897L;
```

`float.h` determines the limits of each type

For more details on floating point, check out this video
https://www.youtube.com/watch?v=PZRI1IfStY0

fsizeofDemo.c

```
float        floatVar;
double       doubleVar;
long double longdoubleVar;

The sizeof(float)           is 4
The sizeof(double)          is 8
The sizeof(long double)     is 16

The sizeof(floatVar)       is 4
The sizeof(doubleVar)      is 8
The sizeof(longdoubleVar) is 16

floatVar       = FLT_MAX;
doubleVar      = DBL_MAX;
longdoubleVar = LDBL_MAX;
```

Assigning
340282346638528859811704183484516925440.000000
to floatVar

Assigning
179769313486231570814527423731704356798070567525844996598917476803157260780028538760589558632766878171540458953514382464234321326889464182768467546703537516986049910576551282076245490090389328944075868508455133942304583236903222948165808559332123348274797826204144723168738177180919299881250404026184124858368.000000
to doubleVar

Assigning
11897314953572317650212638530309702051690633222946242004403237338917370055229707226164102903365288828535456978074955773144274431536702884341981255738537436786735932007069732632019159182829615243655295106467910866143117906321697788388961347865606003991487
53433211454911160088679845154866512852340149773037600009125479393966223151383622417838542743917838138717805889487540575168226347659235576974805113725649020884855222494791399377585026011773549180099796226026859508558883608159846900235645132346594476384939
85927645628457966177293040780660922910271504608538808795932778162298682754783076808004015069494230341172895777710033571401055977524212405734700738625166011082837911962300846927720096515350020847447079244384845491288672300061908512647211195136146752763351
95629275979572502780029807959041931396030214709970352764674455309220226796562809914982320833296412410385092391847347861219216972105434842870483534081130425730022164213489173471742348007148807510020643905172342476560047217680964861079949434157034763206435
58624207443504424380566136017608837478165389027809576975977286860071487028287955567141404632615832623602762896316173978484254486860699482708679680480787025118589308385465842230409088059962945945862019037660484467909260022254105307759010657606713472001258464069570302571389609837579989269545530523685607586831792231136395194688508807718721047052039575874800131431314442549439199401757531693393923668818561891299317291042529212368351599223220509980016771027840353601408292963981151228777681357060457893435355451696539561254048846447169786893211671087229088082778350518228857646062218739702851655083720992349483334435228984751232753726636066213902281264706234075352071724058665079518217303463782631353393707677490195019784169044182473806316282858685774143258116536404021840272491339332094921949842244273042701987304453662035026238695780468200360144729199712309553005720614186697485284685618651483271597448120312194675168637934309618961510733006555242148519520176285859509105183947250286387163249416761380499631979144187025430270675849519200883791515169401581740046711477877201459644461175204059453504764721807975761111720846273639279600339670470037613374509553184150073796412605047923251661354841291884211340823015473304754067072818763503617332908005951896325207071673904547771296822652062235914399193768044002923809031124379126147762559646942219813751469670794468703580043925076594516183798118593920495440361149153107822510726914869798092409467721427270124043771874092167566136349389004512323516681460893224006979931761017805338191849981933008410985993938760292601390911414526003720284872132411955424282101831204216104467404621635336900583664606591156298764745525068145003932941404131495400677602951005962253022823003631473824681059648442441324864573137437595096416168048024129351876204668135636877532814675538798871771836512893947195335061885003267607354388673368002074387849657014576090349857571243045102038730494854256702479339322809110526041538528994849203991091946129912491633289917998094380337879522093131466946149705939664152375949285890960489916121944989986384837022486672249148924678410206183364627416969576307632480235587975245253737035433882960862753427740016333434055083537048507374544819754722228975281083020898682633020285259923084168054539687911418297629988964576482765287504562854924265165217750799516259669229114977788962356670956627138482018191348321687995863652637620978285070099337294396784639879024914514222742527006363942327998483976739987154418554201562244154926653014515504685489258620276085761837129763358761215382565129633538141663949516556000264159186554850057052611431952919918807954522394649627635630178580896692226406235382898535867595990647008385687123810329591926494846250768992258419305480763620215089022149220528069842018350840586938493815498909445461977893029113576516774062322789831403347327660395223160342282471752818181884430488092132193355086987339586127607367086665237555567580317149010847732009642431878007000879734603290627894355374356444885190719161645514115576193939969076741515640282654366402676009508752394550734155613586793306603174472092444651353236664764973540085196704077110364053815007348689179836404957060618953500508984091382686953509006678332447257871219660441528492484004185093281190896363417573989716659600075948780061916409485433875852065711654107226099628815012314377944008749301944744330784388995701842710004808305012177123560622895076269042856800047718893158089358515593863176652948089031267747029662545110861548958395087796755464137944895960527975209874813839762578592105756284401759349329416214833956535018919681138909184379573470326940634289008780584694035245347939808067427323629788710086717580253156130235606487870925986528841635097252953709111431720488774740553905400942537542411931794417513706468964386151771884986701034153254238591108962471088538580868883777725864856414593426212108664758848926003176234596076950884914966244415660441955208681198977024 0.000000

to longdoubleVar

fsizeofDemo.c

```
The sizeof(floatVar)       is 4
The sizeof(doubleVar)      is 8
The sizeof(longdoubleVar) is 16
```

The contents of a variable do not change the `sizeof()` that variable.

Floating Point Types

Using operators with floating point types.

arithmetic	+	-	*	/		
relational	==	!=	<	<=	>	>=
logical	!	&&	||			

Expression	Value	Type
`2.5 + 5.7`	`8.2`	`double`
`2.5 <= 3.62`	`1` (true)	`int`
`2.5 == 3.62`	`0` (false)	`int`
`2.5 / 3.62`	`0.6906`	`double`
`2.5 && 3.62`	`1` (true)	`int`
`!2.5`	`0` (false)	`int`
`!0`	`1` (true)	`int`

`printf()` – precision specification

```
printf(control_string, args, ...)
```

```
% [flag] [field width] [.precision] [size] conversion
```

.precision

- optional
- a period followed by a decimal integer specifying the number of digits to be printed in a conversion of a floating point value after the decimal point

Input and Output of Floating Point Values

Conversion Specifications for `scanf()`

```
%e %f %g      float
%le %lf %lg   double
%Le %Lf %Lg   long double
```

Conversion Specifications for `printf()`

```
%e  %f  %g  %E  %G    float, double
%Le %Lf %Lg %LE %LG   long double
```

For more about scientific notation

https://www.youtube.com/watch?v=Hmw0wJVud0k

perconDemo.c
psconversionDemo.c

```
Enter a float value for %e              12.3456
Value entered using %e   is             1.234560e+01
Value entered using %.2e is             1.23e+01

Enter a float value for %f              12.3456
Value entered using %f   is             12.345600
Value entered using %.3f is             12.346

Enter a double value for %le            12.3456
Value entered using %le is              1.234560e+01
Value entered using %.4le is            1.2346e+01

Enter a float value for %g              12.3456
Value entered using %g is               12.3456
Value entered using %.2g is             12

Enter a double value %lg                12.3456
Value entered using %lg is              12.3456
Value entered using %.3lg is            12.3

Enter a double long value for %Lg       12.3456
Value entered using %Lg is              12.3456
Value entered using %.4LG is            12.35
```

```c
float f1 = 1;
float f3 = 3;

double d1 = 1;
double d3 = 3;

long double ld1 = 1L;
long double ld3 = 3L;

printf("float version       of 1/3 %.65f\n\n",
        f1/f3);
printf("double version      of 1/3 %.65f\n\n",
        d1/d3);
printf("long double version of 1/3 %.65Lf\n\n",
        ld1/ld3);

printf("sum = %.65Lf\n\n",
        f1/f3 + d1/d3 + ld1/ld3);
```

```
float version       of 1/3
0.3333333432674407958984375
000000000000000000000000000
0000000000000


double version      of 1/3
0.3333333333333333148296162
562473909929394721984863281
2500000000000


long double version of 1/3
0.3333333333333333333423683
514373792036167287733405828
4759521484375


sum =
1.0000000099341073885863759
307390807862248038873076438
9038085937500
```

floatpreDemo.c

Decimal vs Hexadecimal vs Octal

Why do programmers always mix up Halloween and Christmas?

Because Oct 31 = Dec 25

Great video explaining decimal, hexadecimal, octal and binary

https://www.youtube.com/watch?v=5sS7w-CMHkU

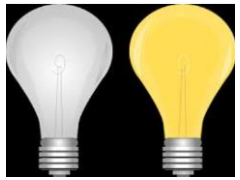
Variables in Computer Memory

Bit vs Byte vs Word

Binary Digit – Bit

0 1

holds 0 or 1 – TRUE or FALSE – ON or OFF



Byte

8 bits

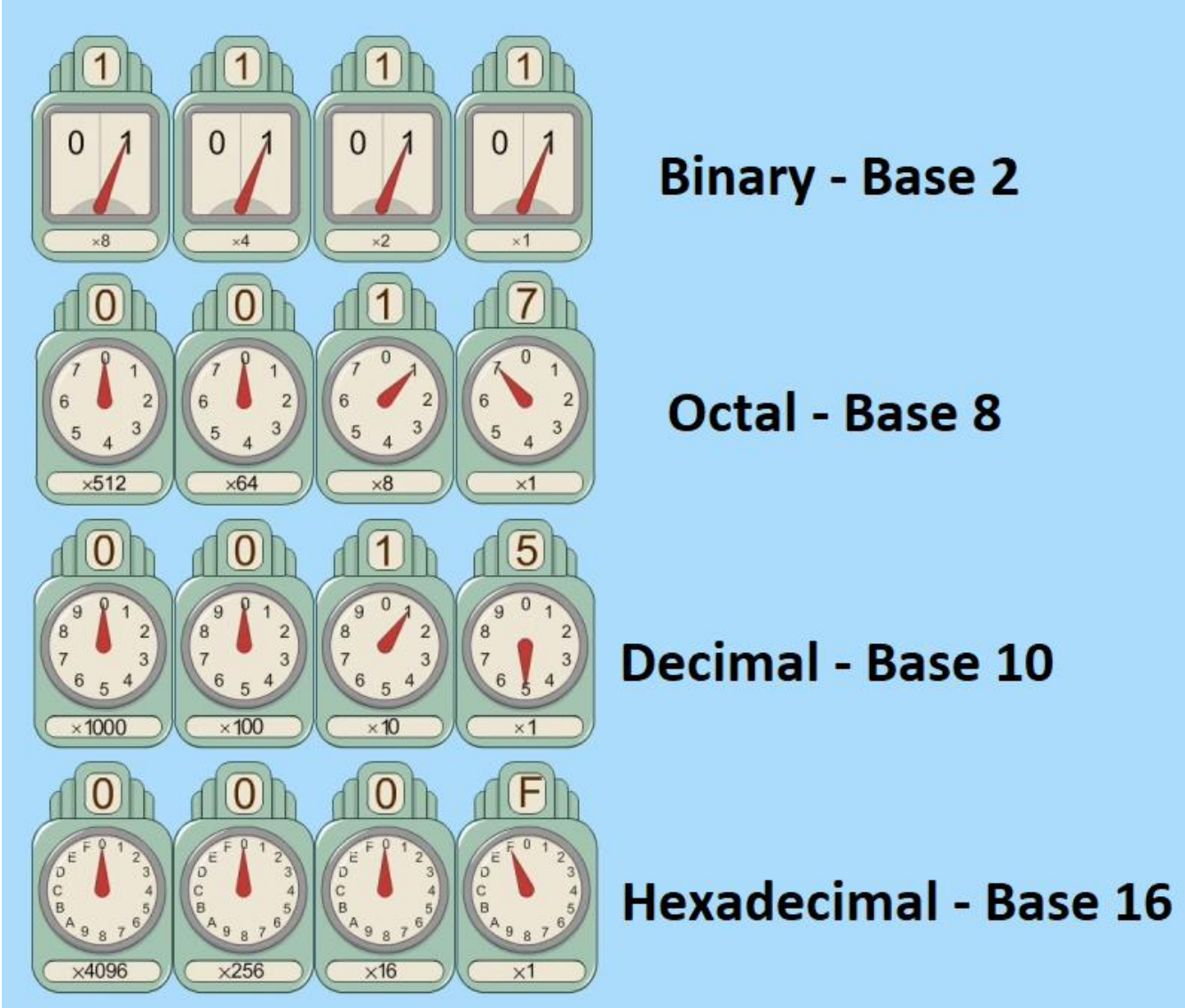
Word

0 1 0 1 0 1 0 0

one or more bytes

0 1 0 1 0 1 0 0 0 1 0 1 0 1 0 0

Integers and Different Integer Bases



Yet another programmer joke….

There are 10 types of people in the world:

Those who understand binary and those who don’t.

Integers and Different Integer Bases

Convert binary to decimal

Convert 11001011_2 to decimal

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	0	1	0	1	1
128	64			8		2	1

$128 + 64 + 8 + 2 + 1 = 203$

$11001011_2 = 203_{10}$

Integers and Different Integer Bases

Convert decimal to binary

Convert 203_{10} to binary

Divide in half and keep the remainder

$$1 \Leftarrow 3 \Leftarrow 6 \Leftarrow 12 \Leftarrow 25 \Leftarrow 50 \Leftarrow 101 \Leftarrow 203$$

1 1 0 0 1 0 1 1

$203_{10} = 11001011_2$

Integers and Different Integers Bases

Octal

Used when the number of bits in one word is a multiple of 3

Convert 1234_{10} to octal

1234/8	**154**/8	**19**/8	**2**/8
2	**2**	**3**	2

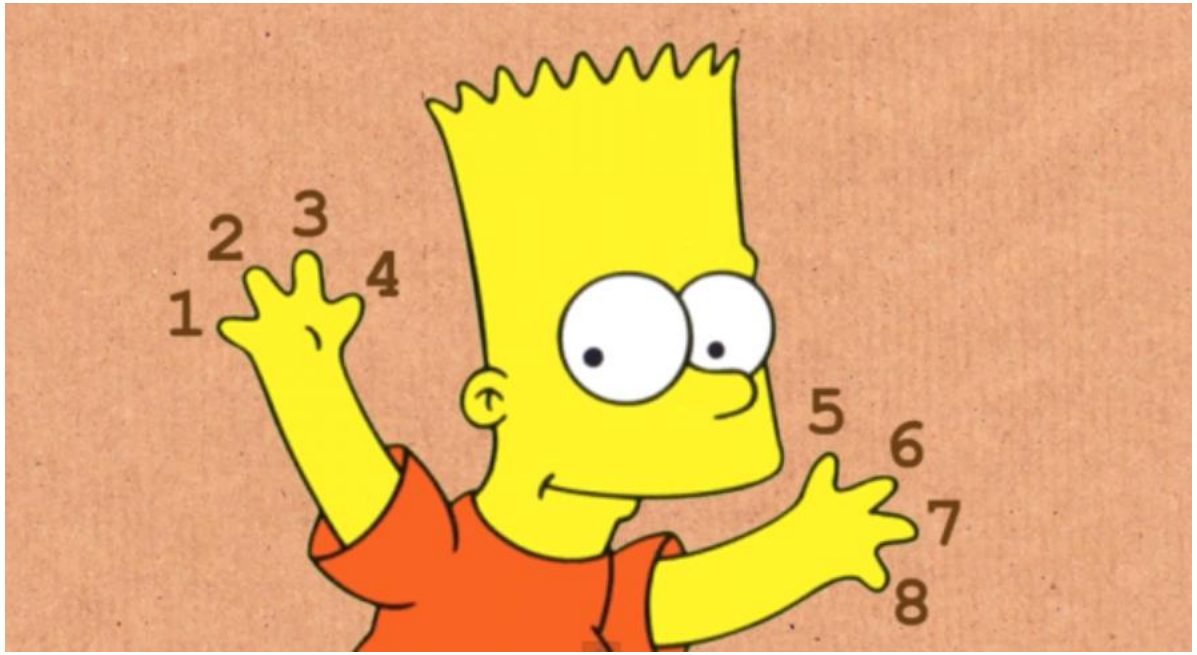
Divide by 8 and keep the remainder

$1234_{10} = 2322_8$

Integers and Different Integers Bases

If we count in decimal because we have 10 digits on our hands, then

the Simpsons must count in?



Integers and Different Integers Bases

Hexadecimal

Used when the number of bits in one word is a multiple of 4

0001111100111010

0001	1111	0011	1010
1	F	3	A

$0001111100111010_2 = 1F3A_{16}$

Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Integers and Different Integers Bases

Hexadecimal

Used when the number of bits in one word is a multiple of 4

Convert 1234_{10} to hexadecimal

1234/16	77/16	4/16
2	13	4

Divide by 16 and keep the remainder

$1234_{10} = 4D2_{16}$

Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

How To Do Number Conversions for OLQs

Since we will be typing all quizzes, we need to establish a method for writing our number conversions. Just writing the answer/result will be 0 points.

So if the question is

Convert 234_{10} to hexadecimal

You need to be able to write the work out like this…

234/16=14R10 14/16=0R14

so the answer is EA. We use the remainders for our answers.

10 = A
11 = B
12 = C
13 = D
14 = E
15 = F

How To Do Number Conversions for OLQs

Convert 232_{16} to base 10.

$(2*16^2)+(3*16^1)+(2*16^0) = 512+48+2 = 562_{10}$

Convert 342_8 to base 10.

$(3*8^2)+(4*8^1)+(2*8^0) = 192+32+2 = 226_{10}$

Convert 23_5 to decimal.

$(2*5^1)+(3*5^0) = 10+3 = 13_{10}$

How To Do Number Conversions for OLQs

Convert 232_{10} to base 16.

232/16 = 14R8 14/16=0R14
$E8_{16}$

Convert $E8_{16}$ to base 10.

$(E*16^1)+(8*16^0) = (14*16)+(8*1) = 224+8 = 232_{10}$

Convert 13_{10} to base 5.

13/5 = 2R3 2/5=0R2
23_5

How To Do Number Conversions for OLQs

#1 mistake made on number conversion questions is…

using the wrong technique

Converting **any** base to base 10 means using the exponents….

$$1212_3 = (1*3^3) + (2*3^2) + (1*3^1) + (2*3^0) =$$
$$= 27 + 18 + 3 + 2 = 50_{10}$$

Important to remember that x^0 is always 1 and x^1 is always x.

How To Do Number Conversions for OLQs

#1 mistake made on number conversion questions is…

using the wrong technique

Converting base 10 to **any** other base means dividing by the base

Convert 50_{10} to base 3.

50_{10} = 50/3=16R2 16/3=5R1 5/3=1R2 1/3=0R1

= 1212_3

Important to remember to keep dividing until the quotient (result of division) is 0

Types of Expressions

- every expression has an associated type
- operators and operands within the expression determine the expression's type
- in a binary operation, both operands are converted to the dominating type before being evaluated
- result will retain the dominate type
- most to least dominate
  - `long double`
  - `double`
  - `float`
  - `unsigned long`
  - `long`
  - `unsigned`
  - `int`

```
int a;
float b;
float c;

c = a + b;
```

`a` would be converted to `float`

This type of conversion is called automatic typecasting.

Forced Type Conversions

type cast

- the type of an expression can be temporarily changed with a type cast
- pair of parentheses enclosing a type specifier
- can be constructed with any of the basic types in C
- no restrictions on the use of type casts
- any type in C can be cast to any other type
  - data may be lost

C:\Users\Donna\Desktop\UTA\Programs\CSE1320\typecast3Demo.c - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

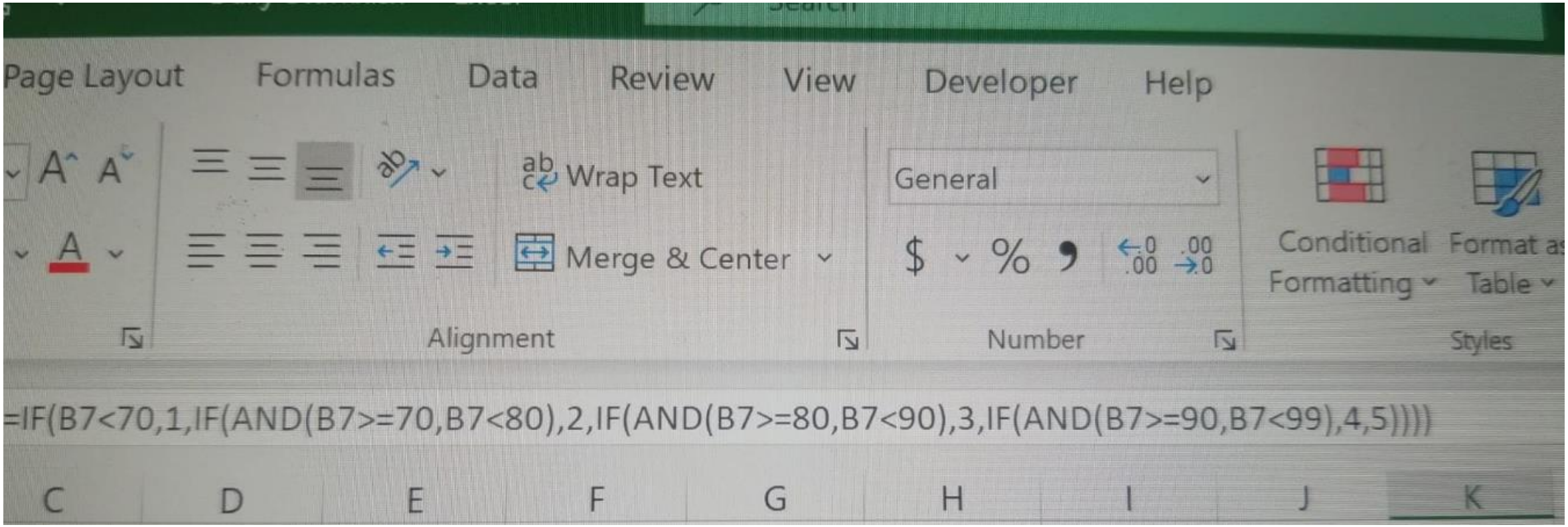
typecast2Demo.c | typecast3Demo.c

```c
/* typecast 3 demo */

#include <stdio.h>

int main(void)
{
    int int1 = 1;
    int int2 = 2;
    int int3 = 4;

    printf("Enter a value for int1 ");
    scanf("%d", &int1);

    printf("Enter a value for int2 ");
    scanf("%d", &int2);

    printf("Enter a value for int3 ");
    scanf("%d", &int3);

    printf("Sum = %d\n", int1+int2+int3);
    printf("Average = %d\n", (int1+int2+int3)/3);

    return 0;
}
```

C source file | length : 407 lines : 25 | Ln : 2 Col : 1 Sel : 0 | 0 | Windows (CR LF) | UTF-8 | INS

emo.c

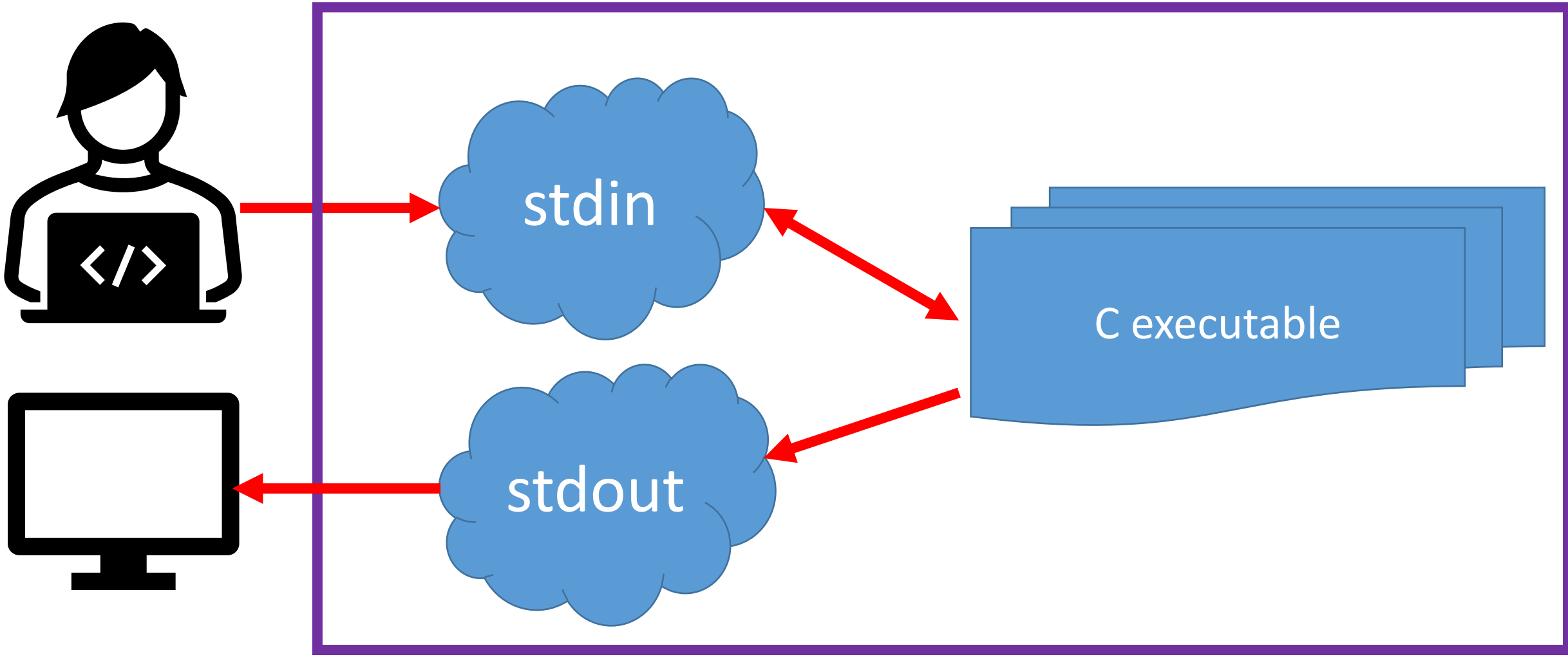
Using Logic Beyond Just Programming



As you type at the keyboard, the characters you type go into a buffer in memory called stdin.

When your executable program needs input – `scanf()` is asking for input – those characters are pulled from stdin.

When a `printf()` is executed, your program sends the data to be printed to a memory buffer called stdout.



This is not the complete story though…

What other key did you press when you answered the "Enter a number " prompt?

<ENTER>

You typed

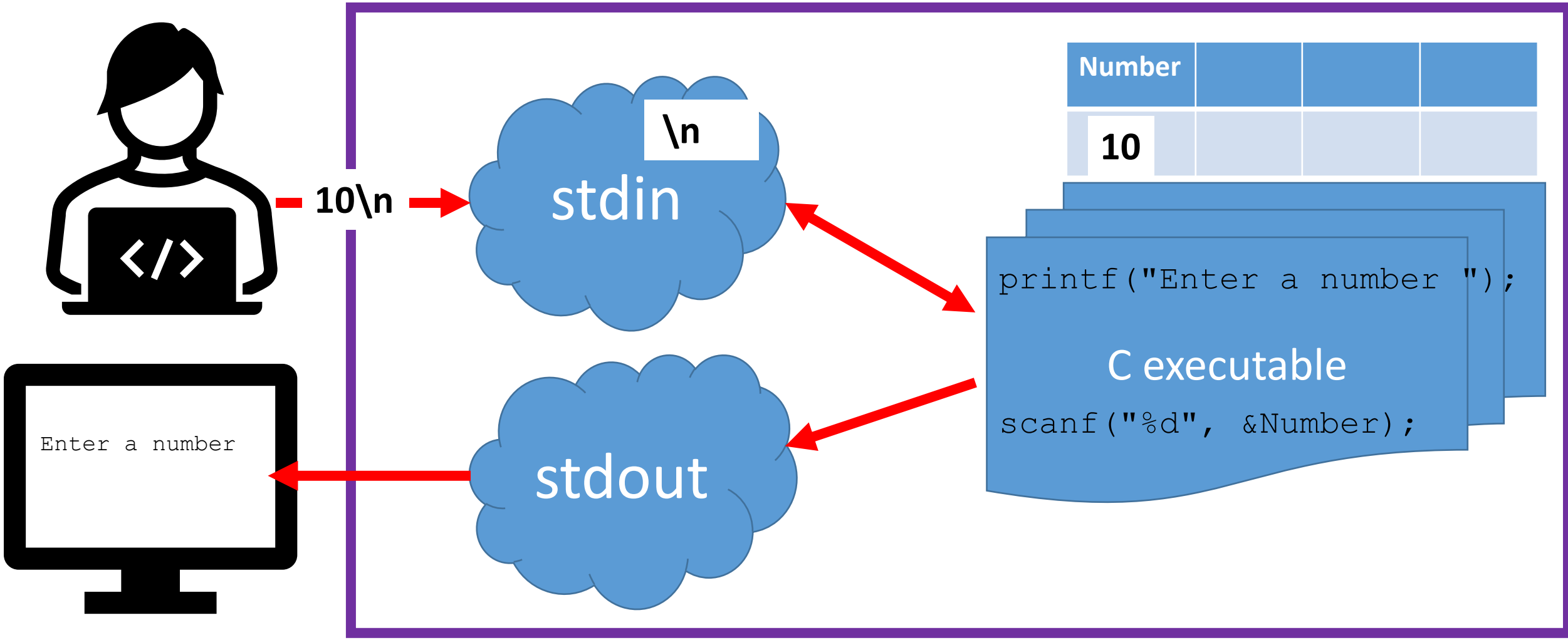
10<ENTER>

You have to press <ENTER> to complete your entry.

That <ENTER> is also sent to stdin as the `\n` character

What will happen if the next `scanf()` wants a character from stdin?

Will you the user be prompted to enter a character?



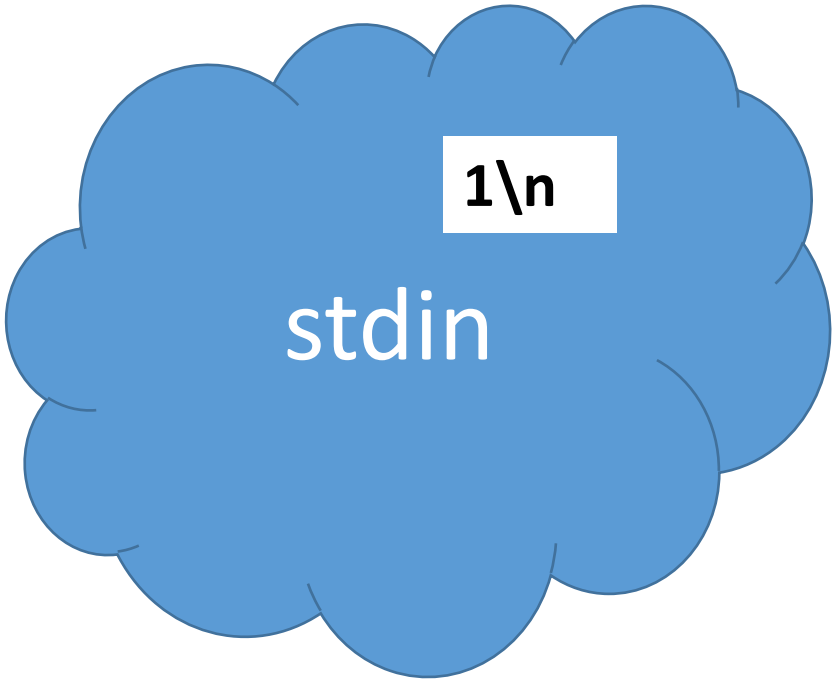
```c
#include <stdio.h>

int main(void)
{
   int Num1 = 0;
   int Num2 = 0;
   char Letter1;
   char Letter2;

   printf("Enter a value for Num1 ");
   scanf("%d", &Num1);

   printf("Enter a value for Num2 ");
   scanf("%d", &Num2);

   printf("You entered %d and %d\n", Num1, Num2);

   return 0;
}
```

```
gcc scanfnewlineDemo.c
./a.out
Enter a value for Num1 1
Enter a value for Num2 2
You entered 1 and 2
```

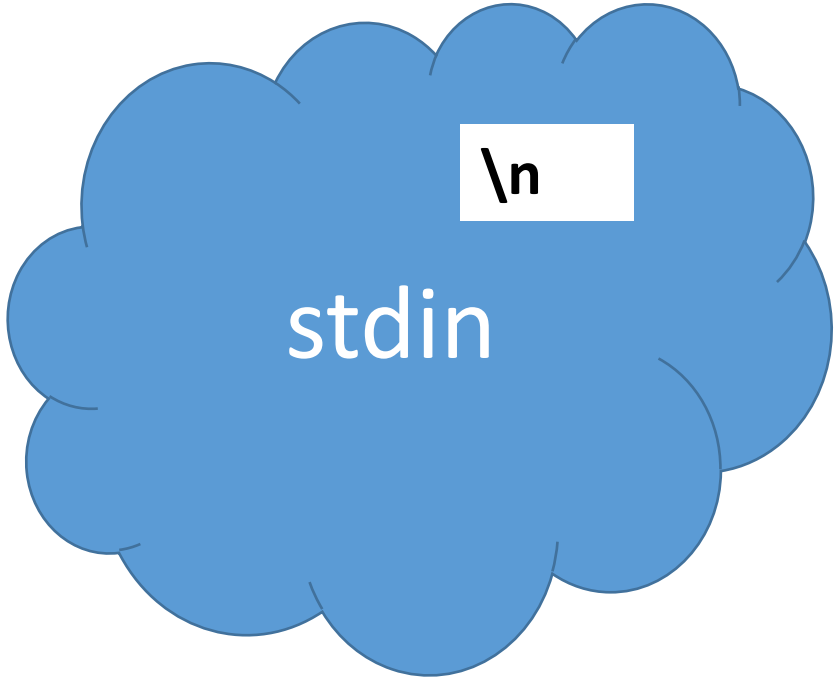
What about the <ENTER> keys in stdin?

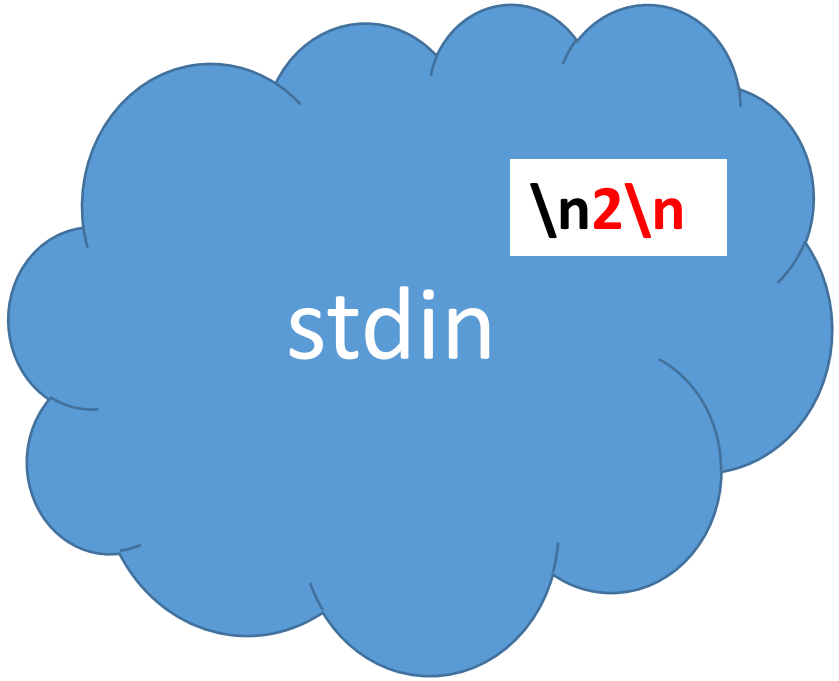
Where did they go?

When `scanf()` is used with `%d`, whitespace and special characters are skipped/ignored.

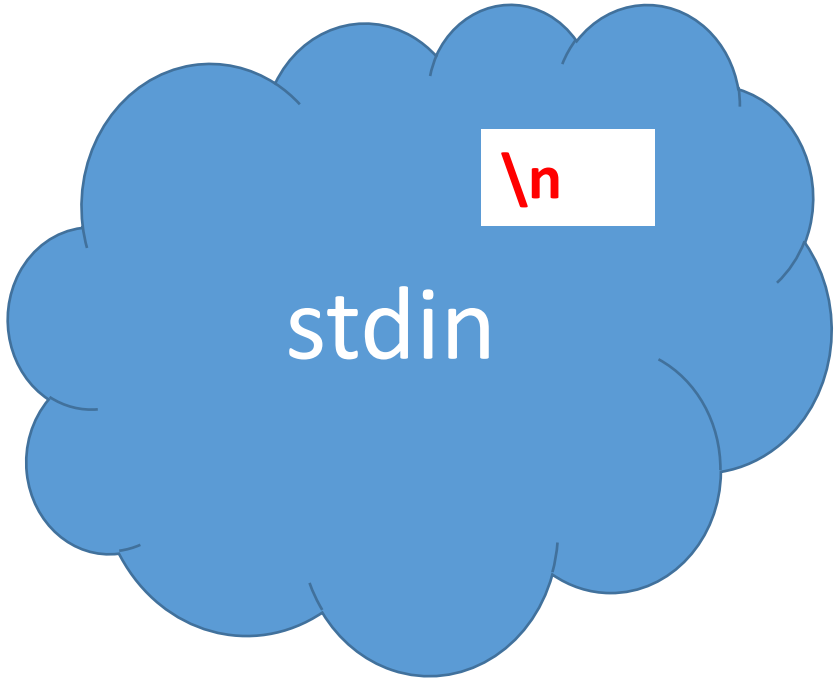


%d





%d



```c
#include <stdio.h>

int main(void)
{
   int Num1 = 0;
   int Num2 = 0;
   char Letter1;
   char Letter2;

   printf("Enter a value for Letter1 ");
   scanf("%c", &Letter1);

   printf("Enter a value for Letter2 ");
   scanf("%c", &Letter2);

   printf("You entered %d and %d\n",
          Letter1, Letter2);

   return 0;
}
```

```
gcc scanfnewlineDemo.c
./a.out
Enter a value for Letter1 A
Enter a value for Letter2 You entered 65 and 10
```

After A was entered, the user was not allowed to enter a value for Letter2 – the program printed and completed.

Why did it print 10 for Letter2?

CRLF vs LF vs CR

CRLF

Windows

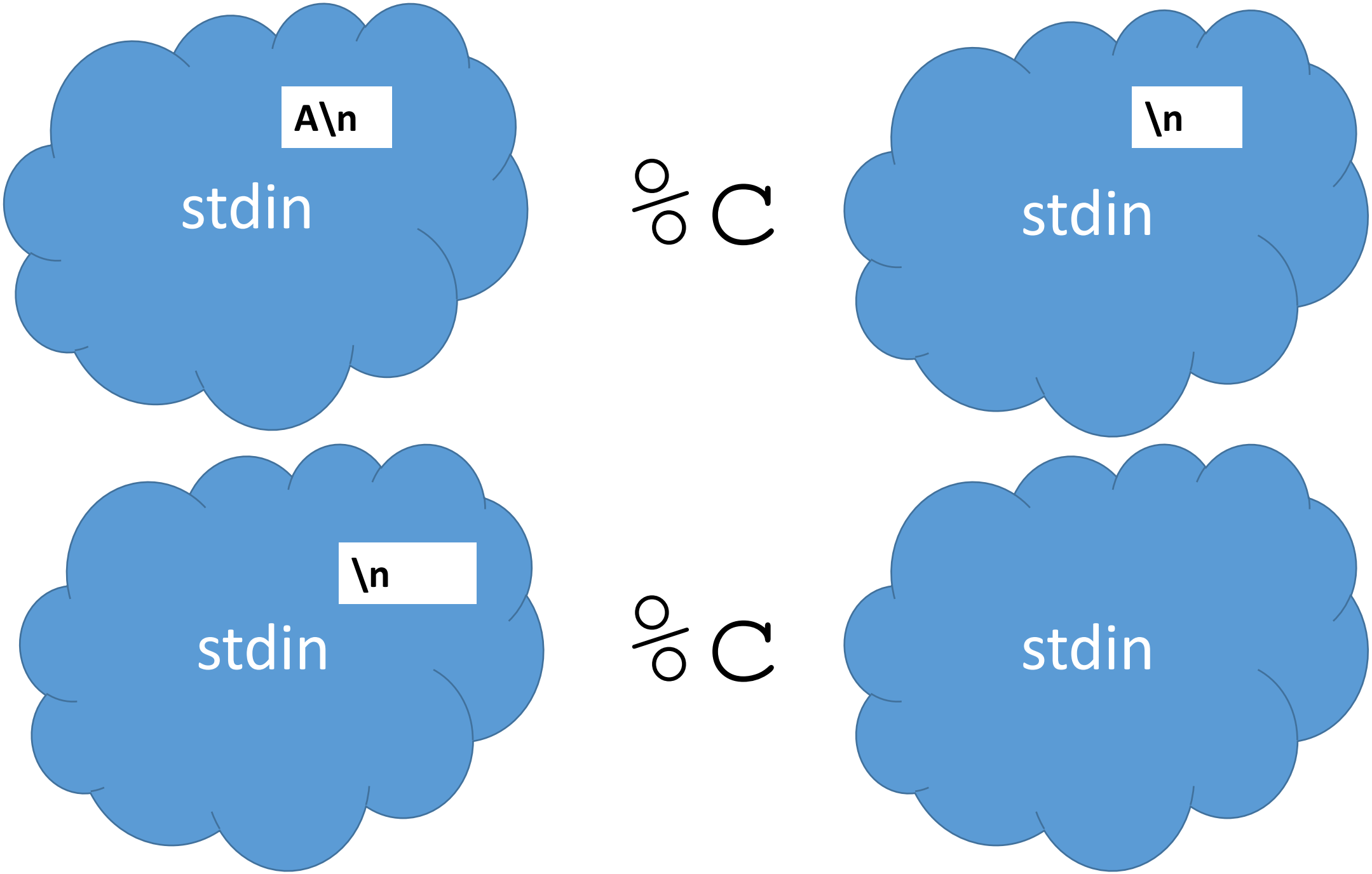
LF

Unix

CR

Mac

Ascii	Char	Ascii	Char	Ascii	Char	Ascii	Char
0	Null	32	Space	64	@	96	`
1	Start of heading	33	!	65	A	97	a
2	Start of text	34	"	66	B	98	b
3	End of text	35	#	67	C	99	c
4	End of transmit	36	$	68	D	100	d
5	Enquiry	37	%	69	E	101	e
6	Acknowledge	38	&	70	F	102	f
7	Audible bell	39	'	71	G	103	g
8	Backspace	40	(	72	H	104	h
9	Horizontal tab	41	)	73	I	105	i
10	Line feed	42	*	74	J	106	j
11	Vertical tab	43	+	75	K	107	k
12	Form feed	44	,	76	L	108	l
13	Carriage return	45	-	77	M	109	m
14	Shift in	46	.	78	N	110	n
15	Shift out	47	/	79	O	111	o
16	Data link escape	48	0	80	P	112	p
17	Device control 1	49	1	81	Q	113	q
18	Device control 2	50	2	82	R	114	r
19	Device control 3	51	3	83	S	115	s
20	Device control 4	52	4	84	T	116	t
21	Neg. acknowledge	53	5	85	U	117	u
22	Synchronous idle	54	6	86	V	118	v
23	End trans. block	55	7	87	W	119	w
24	Cancel	56	8	88	X	120	x
25	End of medium	57	9	89	Y	121	y
26	Substitution	58	:	90	Z	122	z
27	Escape	59	;	91	[	123	{
28	File separator	60	<	92	\	124	|
29	Group separator	61	=	93	]	125	}
30	Record separator	62	>	94	^	126	~
31	Unit separator	63	?	95	_	127	Forward del.


A\n
stdin
% C
\n
stdin
\n
stdin
% C
stdin

```c
#include <stdio.h>

int main(void)
{
   int Num1 = 0;
   int Num2 = 0;
   char Letter1;
   char Letter2;

   printf("Enter a value for Num1 ");
   scanf("%d", &Num1);

   printf("Enter a value for Letter2 ");
   scanf("%c", &Letter2);

   printf("You entered %d and %c\n", Num1, Letter2);

   return 0;
}
```

```
gcc scanfnewlineDemo.c
./a.out
Enter a value for Num1 1
Enter a value for Letter2 You entered 1 and
```

`scanf()` using `%d` left the `\n` in stdin when it read in `Num1`

The `scanf()` for `Letter2` looked in stdin and found `\n` and used it and did not prompt for a value.

So if we have a `scanf()` using a `%c` after a `scanf()`, we are going to have this issue of the <ENTER> being left in stdin and being used by the next `scanf()`

```
scanf("%d", &blahblah);

scanf("%c", &moreblahblah);
```

Will use \n leftover from previous input

The fix for this problem is very simple.

*C:\Users\frenc\Desktop\VM\CSE1310\scanfnewlineDemo.c - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

ShortcutDemo.c | DecisionTree.c | scanfnewlineDemo.c

```c
// Demo of scanf() and newline

#include <stdio.h>

int main(void)
{
    int Num1 = 0;
    int Num2 = 0;
    char Letter1;
    char Letter2;

    printf("Enter a value for Num1 ");
    scanf("%d", &Num1);

    printf("Enter a value for Letter2 ");
    scanf("%c", &Letter2);

    printf("You entered %d and %c\n", Num1, Letter2);

    return 0;
}
```

C source file | length : 343 | lines : 23 | Ln : 16 | Col : 12 | Pos : 253 | Windows (CR LF) | UTF-8 | INS

Type here to search

12:13 AM
2/4/2021

"%c" vs " %c" vs "%d"

Using

```
scanf("%d", …);
```

skips whitespace and special characters. `\n` is a special character; therefore, `%d` skips it.

Using

```
scanf("%c", …);
```

`%c` does not skip `\n` because `%c` processes whitespace and special characters

Using

```
scanf(" %c", …);
```

Putting a blank in front of the `%c` tells `scanf()` to skip whitespace and special characters

C: > Users > Donna > VSCODE > CSE1320 > StudentCode > Test.c > main(void)

```c
#include <stdio.h>

int main(void)
{
    int Op1, Op2;
    char operator;

    printf("Enter an expression ");
    scanf("%d%c%d", &Op1, &operator, &Op2);
    printf("The entered expression is %d %c %d\n", Op1, operator, Op2);

    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL wsl

```
frenchdm@DonnaPC:/mnt/c/Users/Donna/VSCODE/CSE1320/StudentCode$
```

scanf()

```
printf("Enter an expression \n");
scanf("%d", &Op1);
scanf(" %c", &operator);
scanf("%d", &Op2);
printf("The entered expression is %d %c %d\n", Op1, operator, Op2);

printf("Enter an expression \n");
scanf("%d %c%d", &Op1, &operator, &Op2);
printf("The entered expression is %d %c %d\n", Op1, operator, Op2);
```

Keep the space even when combining conversion specifiers

Bit Operations on the Integer Types

Bit operations

~	bitwise negation
>>	shift right
<<	shift left
&	bitwise and
^	bitwise xor
|	bitwise or

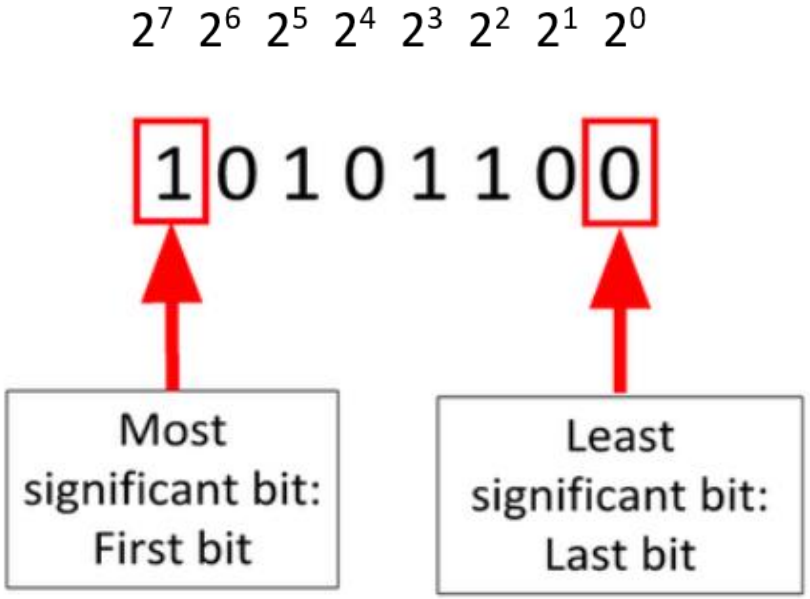
How To Read Bits

Least Significant Bit (LSB)

the bit in a binary number that is of the lowest numerical value

Most Significant Bit (MSB)

the bit in a binary number that is of the highest numerical value



Bit Operations on the Integer Types

How are they used? Why are we learning this?

Gaming software

- performance
  - deciphering online game protocols
- image masking – when one image needs to be placed over another
- 3D games to determine distances

IP addresses

- specify what is permitted and what is denied

Image compression/decompression

Bit Operations on the Integer Types

bitwise negation

~expression

where expression has an integer type

replaces all the 0 bits by 1 and all of the 1 bits by 0

a short will be represented by 16 bits/2 bytes

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**

Bitwise Negation

~0000000000000001

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

1	0	0	1
0	1	1	0
1	0	0	1

0	1	1	0
1	0	0	1
0	1	1	0

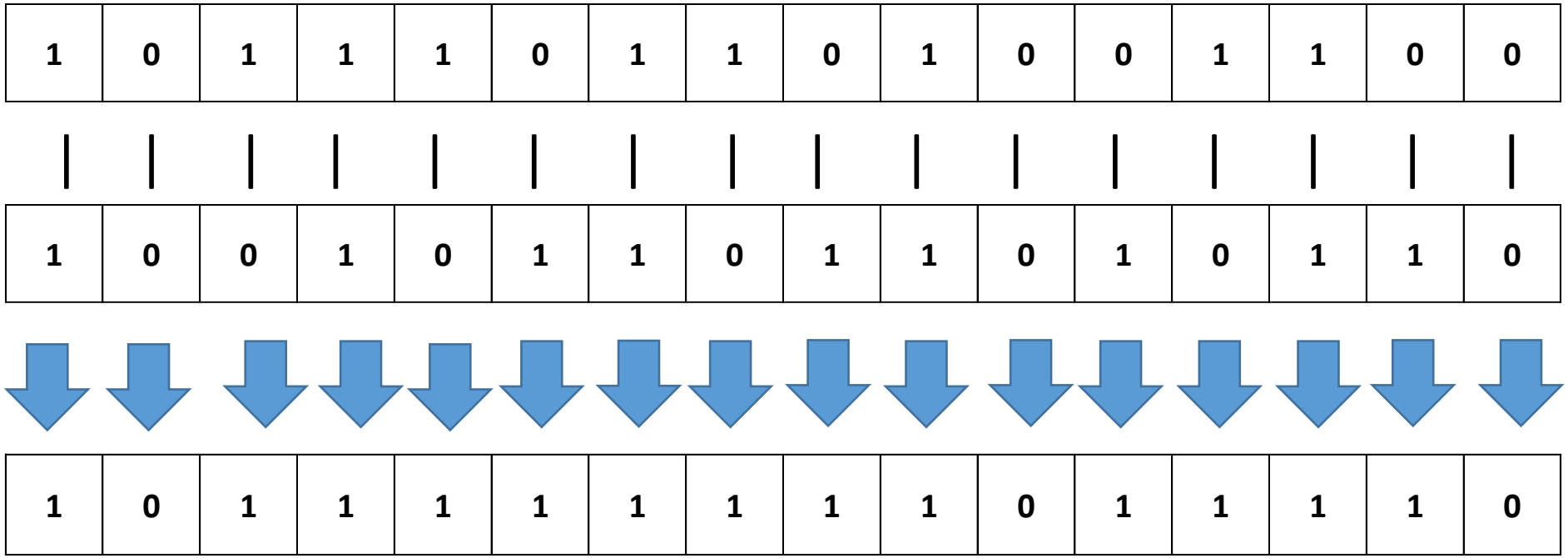
Bitwise AND

1011101101001100 & 1001011011010110

1	0	1	1	1	0	1	1	0	1	0	0	1	1	0	0
&	&	&	&	&	&	&	&	&	&	&	&	&	&	&	&
1	0	0	1	0	1	1	0	1	1	0	1	0	1	1	0
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
1	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0

Bitwise OR

1011101101001100 | 1001011011010110



Bitwise XOR

1011101101001100 ^ 1001011011010110

1	0	1	1	1	0	1	1	0	1	0	0	1	1	0	0

^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^

1	0	0	1	0	1	1	0	1	1	0	1	0	1	1	0

0	0	1	0	1	1	0	1	1	0	0	1	1	0	1	0

Precedence of Bitwise Operators

bitwise negation

bitwise and

bitwise xor

bitwise or

Associate from left to right

Using Bit Masks

Bit masks can be used

to detect whether or not a certain bit is on or off

to turn a bit on or off

Individual bits can be used as flags (0 has a certain interpretation and 1 has a different interpretation).

Masks can be used to evaluate and manipulate each bit.

Question – is the 4th bit on or off in 345?

```
number   345      0000000101011001
&        &        &
mask     16       0000000000010000
         ------   ----------------
         16       0000000000010000
```

Question – is the 2nd bit on or off in 123?

```
number   123      0000000001111011
&        &        &
mask     4        0000000000000100
         ------   ----------------
         0        0000000000000000
```

Question – is the 4th bit on or off in 200?

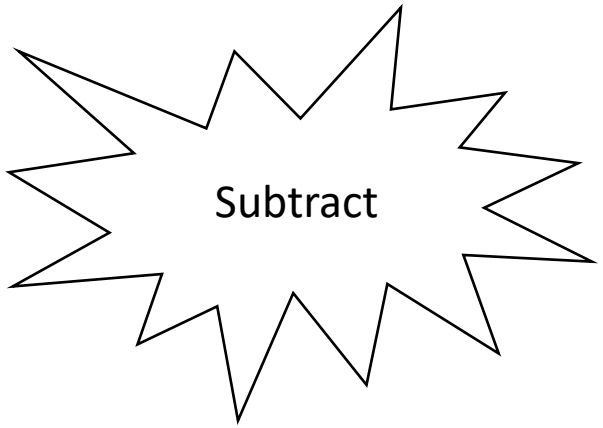
```
number   200      0000000011001000
&        &        &
mask     16       0000000000010000
         ------   ----------------
         0        0000000000000000
```

Question – is the 6th bit on or off in 124?

```
number   124      0000000001111100
&        &        &
mask     64       0000000001000000
         ------   ----------------
         64       0000000001000000
```

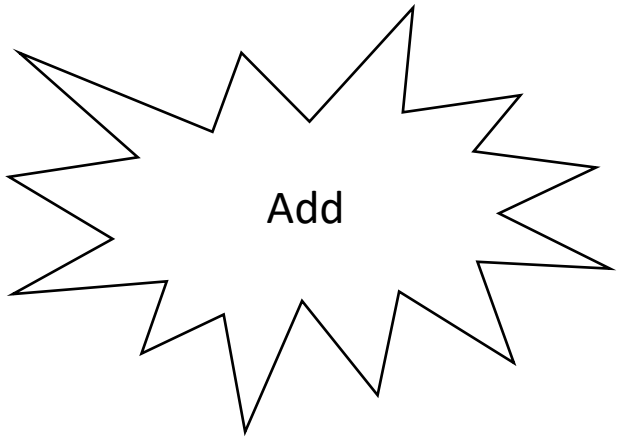
Question – what happens when the 4th bit is turned off in 345?

```
number   345      0000000101011001
^        ^        ^
mask     16       0000000000010000
         ------   ----------------
         329      0000000101001001
```



Question – what happens when the 4th bit is turned on in 200?

```
number   200      0000000011001000
^        ^        ^
mask     16       0000000000010000
         ------   ----------------
         216      0000000011011000
```



Question – what happens if we use a non power of 2 bit mask?

```
number   345      0000000101011001
^        ^        ^
mask     14       0000000000001110
         ------   ----------------
         343      0000000101010111
```

Did it add? Did it subtract? How did we get from 345 to 343?

It added and subtracted!

Turning a bit on adds and turning a bit off subtracts the power of 2 for that bit.

```
number   345       0000000101011001
^        ^         ^
mask     14        0000000000001110
         ------    ----------------
         343       0000000101010111
```

- 1st 1 in 14 XORs with 1 from 345 and flips to 0 which turns off 2^3 so 345 – 8 = 337
- 2nd 1 in 14 XORs with 0 from 345 and flips to 1 which adds 2^2 so 337+4 = 341
- 3rd 1 in 14 XORs with 0 from 345 and flips to 1 which adds 2^1 so 341 + 2 = 343
- 4th 0 in 14 XORs with 1 from 345 and stays 1 so no change to 343

Using Bit Masks

A bit mask can be used to determine if a number is odd or even.

What is the last bit in the binary representation of an even number?

$2_{10} = 10_2$

$4_{10} = 100_2$

$2468_{10} = 100110100100_2$

Using Bit Masks

A bit mask can be used to determine if a number is odd or even.

What is the last bit in the binary representation of an odd number?

$3_{10} = 11_2$

$7_{10} = 111_2$

$2469_{10} = 100110100101_2$

Using Bit Masks

A bit mask can be used to determine if a number is odd or even.

The binary representation of even numbers has an LSB/rightmost bit of 0

The binary representation of odd numbers has an LSB/rightmost bit of 1

So how to use a bit mask to determine if the 2^0 bit is on/off?

Use a bit mask of 2^0 which is the value 1.

Question – is our value odd or even?

```
number   2
&        &
mask     1
         -
         0
```

Compiler Error

```
[frenchdm@omega ~]$ gcc bitmaskDemo.c
bitmaskDemo.c: In function ‘main’:
bitmaskDemo.c:7: error: ‘BitMask’ undeclared (first use in this function)
bitmaskDemo.c:7: error: (Each undeclared identifier is reported only once
bitmaskDemo.c:7: error: for each function it appears in.)
```

```c
5  int main(void)
6  {
7      int Number = 0;  BitMask = 0;
8
```

```c
5  int main(void)
6  {
7      int Number = 0, BitMask = 0;
8
```

```c
// bitmask demo

#include <stdio.h>

int main(void)
{
    int Number = 0, BitMask = 0;

    printf("Enter number ");
    scanf("%d", &Number);

    printf("Enter bit mask ");
    scanf("%d", &BitMask);

    if (Number & BitMask)
        printf("The number is odd\n");
    else
        printf("The number is even\n");

    return 0;
}
```

```
[frenchdm@omega ~]$ a.out
Enter number 2
Enter bit mask 1
The number is even
[frenchdm@omega ~]$ a.out
Enter number 3
Enter bit mask 1
The number is odd
[frenchdm@omega ~]$ a.out
Enter number 4
Enter bit mask 1
The number is even
[frenchdm@omega ~]$ a.out
Enter number 7
Enter bit mask 1
The number is odd
[frenchdm@omega ~]$ a.out
Enter number 2468
Enter bit mask 1
The number is even
[frenchdm@omega ~]$ a.out
Enter number 2469
Enter bit mask 1
The number is odd
```