

CS 137

Introduction to C

Fall 2025

Victoria Sakhnini

Table of Contents

Compiling and running our first program	2
Comments	4
Variables, Data Types, and Arithmetic Expressions	4
Variables in C	4
Back to printing (Output)	5
Basic Operations	6
Assignment Operators	7
Increment/Decrement Operators	8
Relational and Logical Operators	10
C Operator Precedence	11
Bit-Wise Operators	12
Two's complement form	15
Arithmetic Overflow and Underflow	16
Endianness	17
De Morgan's Theorem	18
Input – scanf	18
Additional Examples (Enrichment)	21
Extra Practice Problems	23

Compiling and running our first program

Let us begin with a program that displays "Programming is fun" in your window. The following is a C program to accomplish this task.

```
1. #include <stdio.h>
2. int main(void)
3. {
4.     printf("Programming is fun.\n");
5.     return 0;
6. }
```



- Lowercase and uppercase letters are distinct.
- Indentation is not mandatory, but it makes your code more readable.
- Use Tab characters to indent lines.
- `#include <stdio.h>` should be included at the beginning of just about every program you write. It tells the compiler information about the `printf()` output routine.
- `int main(void)` informs the system that the name of the program is `main()`, which returns an integer value, abbreviated "int".
- `main` is a special name that indicates precisely where the program is to begin execution.
- `void` means that the `main` function takes no arguments (I will discuss this later). The parentheses specify the start and the end of `main`.
- `printf` prints the followed string to screen.
- `\n` is the *newline* character.
- All program statements in C must be terminated by semicolon (;).
- `return 0;` means finish execution and return to the system (caller) a status value of 0. Zero indicates the program was completed successfully; anything non-zero is a failure. You can also omit the `return` statement; the C11 standard defaults to returning zero.

After you write the program in any editor you choose (Unix users often use `vi` or `vim`, more about it will be covered during the first lab). You need to save the file with extension `.c`

Let us assume that the program above is typed into a file called `prog1.c`. Next, we need to compile the program.

Using the GNU C compiler will be as simple as `gcc` command at the terminal followed by the file name, like this:

```
gcc prog1.c
```

If you make any mistakes, the compiler lists them. No errors were found in your program if nothing appeared except the command prompt. In this case, the compiler creates an executable version of your program and is called `a.out` by default. To run the program type `./a.out`

You can specify a different name for the executable file while using `-o` option:

```
gcc -o prog1 prog1.c or gcc prog1.c -o prog1
```

To run the program, simply type `./prog1`



```
@ubuntu1804-004% cat prog1.c
#include <stdio.h>
int main (void) {
    printf("Programming is fun.\n");
    //return 0;
}
@ubuntu1804-004% ls
prog1.c
@ubuntu1804-004% gcc prog1.c
@ubuntu1804-004% ls
a.out prog1.c
@ubuntu1804-004% ./a.out
Programming is fun.
@ubuntu1804-004% gcc prog1.c -o prog1
@ubuntu1804-004% ls
a.out prog1 prog1.c
@ubuntu1804-004% ./prog1
Programming is fun.
@ubuntu1804-004% gcc prog1.c -o myprog
@ubuntu1804-004% ls
a.out myprog prog1 prog1.c
@ubuntu1804-004% ./myprog
Programming is fun.
@ubuntu1804-004%
```


→ Compile prog1.c, and create an a.out executable file

→ Run the a.out file

→ Compile prog1.c, but rather than a.out, name the executable prog1

? What is the exact output (how many lines? What is printed in each line? Try to figure out the answer manually before running the code) of the following program?ⁱ

```
1. #include <stdio.h>
2. int main(void)
3. {
4.     printf("Programming is fun.");
5.     printf("And programming in C is even more fun.\n");
6.     printf("This\nis\neven\nmore\nfun!");
7.     return 0;
8. }
```

 : There are some options for an IDE if you like GUI-based learning. I've heard good things about Eclipse, VSCode, and Pelles C. You can use whatever editor you would like.

Back to returning success:

By convention, `return 0;` is a success and anything non-zero is a failure. There are alternatives to the above code:

```
1. #include <stdlib.h> //tells the compiler information about EXIT_SUCCESS and
2.                        // EXIT_FAILURE
3. int main(void)
4. {
5.     return EXIT_SUCCESS; // which is 0
6. }
7. // or return EXIT_FAILURE; // which is 1
```

Comments

A comment statement is used for documentation to enhance the program's readability. There are two ways to insert comments:

```
// writing comment on one line
/* writing comments on
multiple lines
This comment consists of three lines */
```

Variables, Data Types, and Arithmetic Expressions

Variables in C

A variable is a name given to a storage area that our program can manipulate. Each variable in C has a specific type that determines the valid range of values and the size and layout of the variable's memory.

Rules for variable names:

- It must begin with a letter or an underscore.
- After the first letter, it can be letters, numbers or underscores.
- Case sensitive.
- Cannot be keywords (e.g. `int`, `while` etc.)

There are five basic data types in C: `int`, `float`, `double`, `char`, and `_bool`

Type Specifiers: `long`, `long long`, `short`, `unsigned`, and `signed`; each could be based directly before the basic variable type, and they change the range of possible values that could be assigned to that variable.

In this module, we will introduce the following:

Type	Storage Size	Value Range	Numeric
char	1 byte	$[-128, 127]$	$[-2^7, 2^7 - 1]$
unsigned char	1 byte	$[0, 255]$	$[0, 2^8 - 1]$
int	2 bytes	$[-32768, 32767]$	$[-2^{15}, 2^{15} - 1]$
int	4 bytes	$[-2147483648, 2147483647]$	$[-2^{31}, 2^{31} - 1]$
unsigned int	4 bytes	$[0, 4294967295]$	$[0, 2^{32} - 1]$
short int	2 bytes	$[-32768, 32767]$	$[-2^{15}, 2^{15} - 1]$
unsigned short int	2 bytes	$[0, 65535]$	$[0, 2^{16} - 1]$
long int	4 bytes	$[-2147483648, 2147483647]$	$[-2^{31}, 2^{31} - 1]$
long int	8 bytes	$[-9.22 \cdot 10^{18}, 9.22 \cdot 10^{18} - 1]$	$[-2^{63}, 2^{63} - 1]$
long long int	8 bytes	$[-9.22 \cdot 10^{18}, 9.22 \cdot 10^{18} - 1]$	$[-2^{63}, 2^{63} - 1]$
unsigned long long int	8 bytes	$[0, 1.84 \cdot 10^{19} - 1]$	$[0, 2^{64} - 1]$



- `int` value consists of a sequence of one or more digits. A minus sign preceding the sequence indicates that the value is negative. Examples: 1345, 0, -45.
- `float` value contains decimal places. You can omit digits before or after the decimal point, but obviously, you can't omit both. Examples: 3., 124.89, -.008, 23., -45.7654
- `char` variable can be used to store a single character. Examples: 'a', ';', '8', '\n'.
- In C language, the integer takes 2 bytes for a 32-bit compiler and 4 bytes for a 64-bit compiler.

Back to printing (Output)

Previously, we learned that `printf` takes a string to be printed, for example, `printf("hi");`

However, there is another format that allows us to print variables.

```
printf(string, argument(s));
```

`string` is what will be displayed, and the `argument(s)` are the values displayed along with the `string`. Where?

Exactly where the placeholders are placed in the `string`.

`%d` is the placeholder for a signed integer, `%c` for a character, `%u` for an unsigned integer, etc. We will introduce the rest as needed.

Let us have a look at the following program and the output:

```
1. #include <stdio.h>
2. int main(void)
3. {
4.     int a = 3;
5.     printf("1 + 2 = %d and %d + %d = 9 \n", a, a, 2*a);
6.     return 0;
7. }
```

Output:

1 + 2 = 3 and 3 + 6 = 9

Explanation:

The first %d was replaced with the value of a (3), the second %d was replaced with the value of a (3), and the third %d was replaced with the value of 2*a (6).

? What happens if the number of placeholders is less than the number of arguments or vice versa?ⁱⁱ

Basic Operations


+/-/* to add/subtract/multiply two values


/ to divide two values (if both values are integers, then the result is an integer as well (all decimal portions of the result is lost)

% to give the remainder of dividing the first value by the second value

() can be used as well.

All the above follow BEDMAS¹ rules and then left-associative rules.

 There is no exponentiation operator in C. You need to use repeated multiplication or a special library.

 What is the value of $a / b * b + a \% b$? Assuming that a and b are integers.ⁱⁱⁱ

¹ B-brackets, E-exponents, DM-multiply or divide (left to right), AS-add subtract (left to right)

Assignment Operators

- * Basically, done last in order of operations.
- * Right associative: evaluate everything to the right first, then assign the value.
- * Syntax: `variable = expr; , for example a = 3;`
- * There are also assignment and operator operations. Syntax: `variable operation= expr;`
for example `a+=2;` means (roughly²) `a = a + 2;` `a *= b + c` means `a = a * (b + c);`
`a /= b + c;` means `a = a / (b + c);`
- * Order is important! `a +=2;` and `a=+ 2;` are different! Why? Hint^{iv}? 🤔

Make sure you assign a value to a variable before accessing the variable to use its value

For example, you should not write `x++` or `x += 5` without first assigning a value to `x`.

Please don't assume that it has zero value when you define `x` (such as `int x;`).

² roughly because of weird situations if `a` is an object with some other side effects but we will not cover those cases in CS137.

Increment/Decrement Operators

They are unary operators that add or subtract one. There are two types: prefix, applied before the variable is used, and suffix, applied after the variable is used.

`++a;` would increment `a` by one and then possibly use `a`.

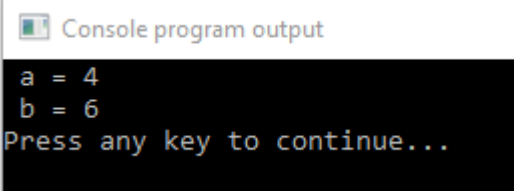
`--a;` would decrement `a` by one and then possibly use `a`.

`a++;` would use `a` first (if applicable), then increment `a` by one.

`a--;` would use `a` first (if applicable), then decrement `a` by one.

Example: (This is not really how we will be using those operators, but this example shows how we can complicate things in C, thus, make sure you keep your code understandable and simple)

```
1. #include <stdio.h>
2. int main(void)
3. {
4.     int a = 3, b = 1;
5.     b += a++ + 2;
6.     printf(" a = %d\n b = %d\n", a, b);
7.     return 0;
8. }
```



Console program output

```
a = 4
b = 6
Press any key to continue...
```

In the beginning, `a` is 3 and `b` is 1, then `b += a++ + 2;` which means `b = b + a++ + 2;` We use `a` first (3), add 2 and add `b(1)` for a total of 6 to be assigned to `b`, `a` is incremented by 1 and its value becomes 4.

? What is the output of the following program (try manually before looking at the solution or running the code)?^V

```
1. #include <stdio.h>
2. int main(void)
3. {
4.     int a = 1, b = 2, c = 3;
5.     a = b += c;
6.     printf(" %d\n", a);
7. }
```

? What is the output of the following program (try manually before looking at the solution or running the code)?^{vi}

```
1. #include <stdio.h>
2. int main(void)
3. {
4.     int a = 10, b = 15;
5.     a += ++b;
6.     printf("%d\n", a);
7.     a = 10, b = 15;
8.     a += b++;
9.     printf("%d\n", a);
10.    a = 10, b = 15;
11.    a += --b;
12.    printf("%d\n", a);
13.    return 0;
14. }
```

? Tricky question: What is the output of the following program and why? Do your own investigation.

```
1. int main(void){
2.     int x = 10;
3.     int y = printf("%d %d\n", printf("%d ", x), x*4);
4.     printf("%d\n", y);
5.     return 0;
6. }
```

Relational and Logical Operators

Relational operators: == (equal) , != (not equal), >(bigger) , < (smaller), >= (bigger or equal), <= (smaller or equal).

Logical operators: ! (negation), || (or), && (and). (the last two are left-associative).

They all return 1 for true(T) and 0 for false(F).

a	b	!a	a b	a && b
T	T	F	T	T
T	F	F	T	F
F	T	T	T	F
F	F	T	F	F

&& and || operators are short-circuited evaluators; they only evaluate right-hand expressions as necessary.

For example, consider the following part of a program:

```
1. int a = 0;  
2. a != 0 && 4/a > 2;
```

Since a is 0, the first condition after the semicolon is false. Hence, C will not evaluate the second condition (that is, this will compile and run without error even though the second condition would typically give an error if on its own³!)

³ division by zero

C Operator Precedence

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(type){ list }	Compound literal(c99)	
2	++ --	Prefix increment and decrement ^[note 1]	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of ^[note 2]	
	_Alignof	Alignment requirement(c11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10		Bitwise OR (inclusive or)	
11	&&	Logical AND	
12		Logical OR	
13	? :	Ternary conditional ^[note 3]	Right-to-Left
14 ^[note 4]	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= =	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Source: http://en.cppreference.com/w/c/language/operator_precedence

Bit-Wise Operators

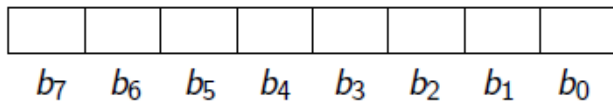
The basics of Bits

- At the smallest scale in the computer, information is stored as bits and bytes.
- A bit can assume either of two values: 1 or 0.
- A byte consists of eight bits.
- All computer data is represented using binary, a number system with 0s and 1s.

Converting from Binary to Decimal.

Let us revisit the unsigned integers:

This is a positional number system that works like a typical binary system.



The value of a number stored in this system is the binary sum that is

$$b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0$$

Examples:

$$01010101 = 2^6 + 2^4 + 2^2 + 2^0 = 64 + 16 + 4 + 1 = 85_{10}$$

$$\begin{aligned} 11111111 &= 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 \\ &= 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 \\ &= 255_{10} \end{aligned}$$

Converting to Binary

One way to do it is to constantly divide by 2 until the result is less than 2. The process keeps track of the integer division and the remainder of dividing by 2. For example, let us convert 38 to a binary value.

Number	Quotient	Remainder
38	19	0
19	9	1
9	4	1
4	2	0
2	1	0
1	0	1


38 in binary (reading bottom to top). This is⁴ $(00100110)_2$.⁵

⁴ using 8 bits, 1 byte of space

⁵ 2 represents the base 2 as 255_{10} means 255 in base 10


Suppose we have unsigned char a=5 (00000101), b=3 (00000011)

Symbol	Operation	Example	Results
&	Bitwise AND	c = a & b;	c = 00000001
	Bitwise Inclusive OR	c = a b;	c = 00000111
^	Bitwise Exclusive OR	c = a ^ b;	c = 00000110
~	Bitwise NOT / Ones Complement	c = ~ a;	c = 11111010
<<	Bitwise Left shift	c = a << 3;	c = 00101000
>>	Bitwise Right shift	c = a >> 2;	c = 00000001

 Be careful how you use it with signed values. Try to calculate the expected results before you look at the output, and the tracing on the next page

```

1. #include <stdio.h>
2. int main(void)
3. {
4.     unsigned char a = 5;
5.     unsigned char b = 3;
6.     printf("%d\n", a & b);
7.     printf("%d\n", a | b);
8.     printf("%d\n", a ^ b);
9.     printf("%d\n", ~a);
10.    printf("%d\n", a << 3);
11.    printf("%d\n", a >> 2);
12.    return 0;
13. }
```

 Console program output

```

1
7
6
-6
40
1
Press any key to continue...
```

Explanation:

n_1	n_2	$n_1 \& n_2$	$n_1 n_2$	$n_1 \wedge n_2$	$\sim n_1$
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

$$\begin{array}{r}
 5 = a \quad 000000101 \\
 3 = b \quad 000000011 \\
 \hline
 a \& b \quad 000000001 \\
 = 1
 \end{array}$$

$$\begin{array}{r}
 a \quad 000000101 \\
 b \quad 000000011 \\
 \hline
 a | b \quad 000000111 \\
 = 7
 \end{array}$$

$$\begin{array}{r}
 a \quad 000000101 \\
 b \quad 000000011 \\
 \hline
 a \wedge b \quad 000000110 \\
 = 6
 \end{array}$$

$$\begin{array}{r}
 a \quad 000000101 \\
 \sim a \quad 11111010 \\
 = -6
 \end{array}$$

$$40 = a \ll 3$$

$$\begin{array}{r}
 \text{left}(3) \leftarrow \\
 000000101 \\
 00101000 \quad \text{put zero}
 \end{array}$$

$$1 = a \gg 2$$

$$\begin{array}{r}
 \text{right}(2) \rightarrow \text{removed} \\
 000000101 \\
 \quad \quad \quad \times \times \\
 000000001
 \end{array}$$

Two's complement form

Signed integers are more complicated as they are represented in Two's complement form. To interpret a binary number in this form, convert the number as a signed integer; if the leading bit (The most left) is 0, stop. Otherwise, subtract 2^n when n is the number of bits.

To negate a value:

Take the complement of all bits⁶

Add 1

Example: Let's compute -38_{10} using this notation in one byte of space.

First, write 38 in binary: $38_{10} = 00100110$

Next, take the complement of all the bits: 11011001

Finally, add 1: 11011010

This last value is -38_{10} .

We can convert the binary number by performing

$$2^7 + 2^6 + 2^4 + 2^3 + 2^1 - 2^8 = 128 + 64 + 16 + 8 + 2 - 256 = 218 - 256 = -38$$

A slightly faster way is to locate the rightmost **1** bit and flip all the bits to its left.

For example: $110110**1**0$ Negating $001001**1**0$



Flipping the bits and adding 1 is the same as subtracting 1 and flipping the bits (exercise).

Everything works mod 2^n .

The idea is that the negation of a positive integer k is represented in memory as $2^n - k$ where n is the size of the data type.

⁶ This will ultimately mean that the first bit is a sign (0 if non-negative, 1 if negative).

Arithmetic works naturally, except that any final carryovers are ignored (see the two examples below).

Watch out for overflow errors!

$$\begin{array}{r} \text{1111 1} \\ 0000\ 0100 \quad (+4) \\ + 1111\ 1101 \quad (-3) \\ \hline 0000\ 0001 \end{array}$$

(1)

$$\begin{array}{r} \text{1111 1} \\ 1111\ 1100 \quad (-4) \\ + 1111\ 1101 \quad (-3) \\ \hline 1111\ 1001 \end{array}$$

(-7)

Arithmetic Overflow and Underflow

An integer overflow occurs when you attempt to store a value larger than the maximum value the variable can hold inside an integer variable. The C standard defines this situation as undefined behaviour (anything might happen). In practice, this usually translates to a wrap of the value of an unsigned integer (adding one to the maximum size of an unsigned value gives you 0 always) and a change of the sign and value if a signed integer was used. Due to a lack of validation of the variables involved, integer overflows result from "wild" increments/multiplications.

Most compilers seem to ignore the overflow, resulting in an unexpected or erroneous result being stored.

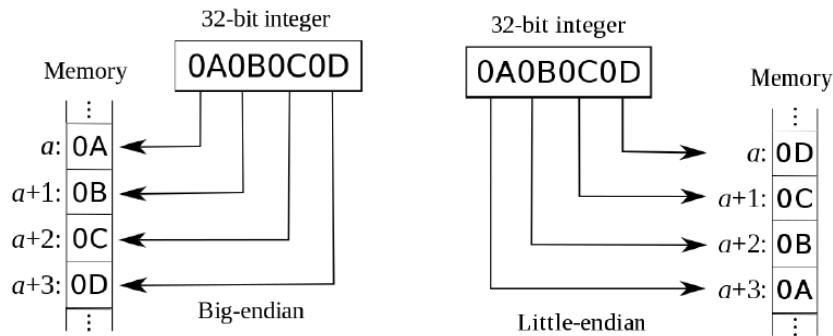
Unfortunately, integer overflows **cannot** be detected after they have happened, so there is no way for an application to tell if the result it has calculated previously is correct.

 Watch the following video to see a real problem example related to overflow then look at the explanation here^{vii}.

<https://youtu.be/jm4wqRj4Lm8?t=3h17m12s>

Endianness

Integers are usually 4 bytes. There are two ways that an integer could be stored. Either they could be big-endian (like in MIPS) where the most significant byte (leftmost) is at the lowest memory address [usual left-right reading; store from least to largest].



(Images Courtesy Wikipedia)

Or it could be little-endian (like in x86), namely, the most significant byte is at the highest memory address. In C, the implementation is computer/compiler dependent and usually irrelevant.

De Morgan's Theorem

De Morgan's Theorem in C

Let P and Q be logical statements. Then

$$\!(P \&\& Q) = \!P \|\! \!Q$$

$$\!(P \|\! Q) = \!P \&\&\!Q$$

Proof:

P	Q	P && Q	!(P && Q)	!P	!Q	!P !Q
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

This is helpful when you will write Boolean expressions in the future and how you can simplify them.

For example, the following two C expressions are equivalent

```
!(a>0) || !(b>0)
```

```
!(a>0 && b>0)
```

Input – scanf

Consider the following statement:

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

`scanf` looks to the standard input (stdin) to read an integer. The integer read will be stored into `a`. If the user doesn't enter an integer, the previous value for the variable `a` remains, and the `scanf` fails. The `&` refers to the memory address of `a`. We'll talk a lot more about this later, so accept that you need the `&` symbol to make the `scanf` work. `scanf` ignores all whitespace/newline characters.

Other features: Consider the following statement:

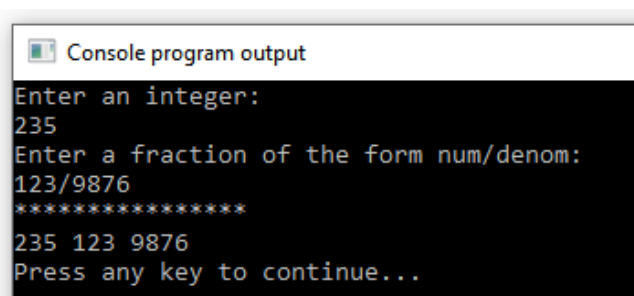
```
scanf ("%d/%d", &num ,& denom );
```

The above code will look for user input of fractions of the form `num/denom` and store the numerator and denominator in the appropriate variables. **This idea will work not just for / but for any special type of inputted format.** The inputted format, however, must match exactly to work. This includes white spaces in the matching.

Example:

```
1. #include <stdio.h>
2. int main(void)
3. {
4.     int a,b,c;
5.     printf("Enter an integer: \n");
6.     scanf("%d", &a);
7.     printf("Enter a fraction of the form num/denom: \n");
8.     scanf("%d/%d", &b, &c);
9.     printf("*****\n");
10.    printf("%d %d %d\n", a, b, c);
11.    return 0;
12. }
```

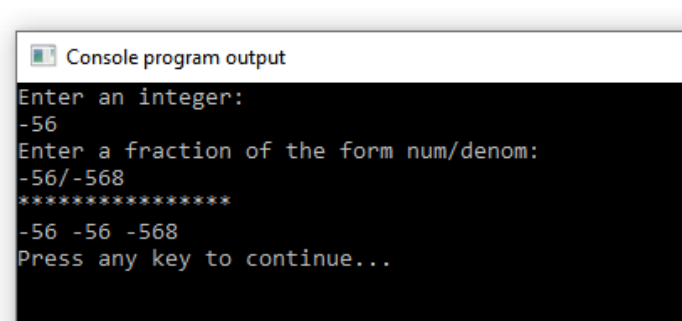
Execution1:



Console program output

```
Enter an integer:
235
Enter a fraction of the form num/denom:
123/9876
*****
235 123 9876
Press any key to continue...
```

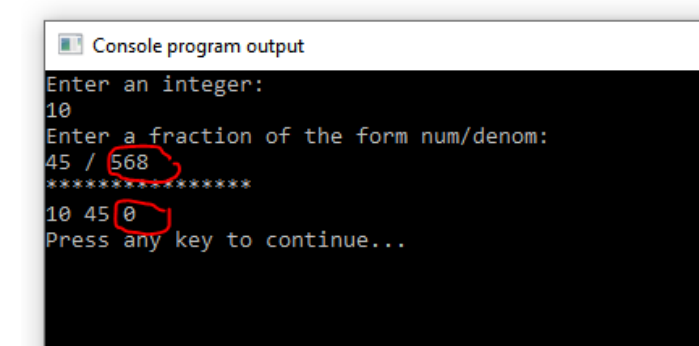
Execution2:



Console program output

```
Enter an integer:
-56
Enter a fraction of the form num/denom:
-56/-568
*****
-56 -56 -568
Press any key to continue...
```

Execution3:



Console program output

```
Enter an integer:
10
Enter a fraction of the form num/denom:
45 / 568
*****
10 45 0
Press any key to continue...
```



Why is the output incorrect?

⁷ The inputted format must match exactly in order to work.

`scanf` returns how many inputs were entered successfully.

Example:

```
1. int main(void) {  
2.     int x,y;  
3.     int z = scanf("%d %d", &x, &y);  
4.     printf("%d\n", z);  
5.     return 0;  
6. }
```

Sample input:

1254 654

Output:

2

Additional Examples (Enrichment)

```
/*
 * Calculates and displays the area and circumference of a metal disc
 * given the radius in centimeters.
 */

#include <stdio.h>
#define PI 3.14159

int main(void)
{
    char part_id1, part_id2, /* input - 3-character part id */
        part_id3;
    double radius, /* input - radius of disc in cm */
        area, /* output - area of disc in cm^2 */
        circum; /* output - circumference of disc (cm) */

    /* Get the part identifier */
    printf("Enter a 3-character part i.d.> ");
    scanf("%c%c%c", &part_id1, &part_id2, &part_id3);

    /* Get the disc radius */
    printf("Enter radius (in centimeters)> ");
    scanf("%lf", &radius);

    /* Calculate the area */
    area = PI * radius * radius;

    /* Calculate the circumference */
    circum = 2 * PI * radius;

    /* Display the part i.d., area and circumference */
    printf("For part %c%c%c, ", part_id1, part_id2, part_id3);
    printf("the disc area is %f cm^2.\n", area);
    printf("The circumference is %f cm.\n", circum);
    return (0);
}
```

Console program output

```
Enter a 3-character part i.d.> DK2
Enter radius (in centimeters)> 5
For part DK2, the disc area is 78.539750 cm^2.
The circumference is 31.415900 cm.
Press any key to continue...
```

```
/*  
 * Using Casts to Prevent Integer Division  
 */  
  
#include <stdio.h>  
  
int main(void)  
{  
    int total_score, num_students;  
    double average;  
    printf("Enter sum of students' scores> ");  
    scanf("%d", &total_score);  
    printf("Enter number of students> ");  
    scanf("%d", &num_students);  
    average = (double)total_score / (double)num_students;  
    printf("Average score is %.2f\n", average);  
    return (0);  
}
```

Console program output

```
Enter sum of students' scores> 1822  
Enter number of students> 25  
Average score is 72.88  
Press any key to continue...
```

Extra Practice Problems

[Try to manually figure out the answers before looking at the provided ones or before running the code]

1) What is the output of the following program^{viii}?

```
1. #include <stdio.h>
2. int main(void)
3. {
4.     int a,b;
5.     a = b = 10;
6.     printf("a    = %d\tb    = %d\n", a, b);
7.     printf("a++ = %d\t++b = %d\n", a++, ++b);
8.     printf("a    = %d\tb    = %d\n", a, b);
9.     return(0);
10. }
```

2) What is the output of the following program^{ix}?

```
1. #include <stdio.h>
2. int main(void)
3. {
4.     int a,b;
5.     a = b = 10;
6.     printf("%d\n", a == b);
7.     printf("%d\n", a != b);
8.     printf("%d\n", a << 1);
9.     printf("%d\n", a >> 2);
10.    printf("%d\n", a >> 3);
11.    printf("%d\n", b >> 5);
12.    return(0);
13. }
```

3) Complete the following program to print the minimum number of buses required to accommodate any number of people planning a road trip together. Each bus can accommodate 50 people.

Examples:

Number of people(participants)	Number of busses required
0	0
45	1
50	1
51	2
99	2
100	2
113	3

The partially provided program:

```
int main(void)
{
    int par, busses;
    printf("Enter The number of participants: \n");
    scanf("%d", &par);

    // You are to complete the code herex

    printf("Number of busses required: %d\n", busses);
    return(0);
}
```

4) Convert the following three numbers to binary: 27, 78, 99^{xi}.

5) Convert the three following three binary numbers to decimals: 01001101 11001101^{xii} 01011110

6) There are two errors in the following program. Find^{xiii} them!

```
1. double sqrt(double num)
2. {
3.     int i;
4.     double result;
5.
6.     for (i=0; i<1000; i++)
7.         result = (result + num/result)\2;
8.
9.     return result;
10. }
```

7) What is the purpose of the following function?

```
int fun_1(int num)
{
    if ((num & 3) == 3)
        return 1;
    return 0;
}
```

Answer ^{xiv}

8) What does this program do?

```
int fun_1(int num)
{
    if ((num & (num - 1)) != 0 )
        Return 1;
    return 0;
}
```

9) Write a program that reads two integers to a and n and rotates the bits of a to the right n times.

Examples:

Input 123 23

Output 62976

Input: 5 14

Output: 1310720

10) Implement the following functions using a limited set of bitwise operations:

1. **bitXor** - Implement the XOR operation (^) using only ~ (bitwise NOT) and & (bitwise AND) operators.
 - **Prototype:** int bitXor(int x, int y);
 - **Description:** Given two integers x and y, compute the result of $x \oplus y$ (XOR) using only ~ and &.
 - **Example:** bitXor(4, 5) = 1
 - **Legal operations:** ~, &
2. **tmin** - Return the minimum two's complement integer.
 - **Prototype:** int tmin(void);
 - **Description:** Return the smallest possible integer for a 32-bit two's complement system.
 - **Example:** tmin() = -2147483648
 - **Legal operations:** !, ~, &, ^, |, +, <<, >>
3. **isTmax** - Return 1 if the given integer x is the maximum two's complement number, and 0 otherwise.
 - **Prototype:** int isTmax(int x);
 - **Description:** Determine if x is equal to the maximum possible integer for a 32-bit two's complement system, which is 2147483647. Return 1 if true, otherwise return 0.
 - **Example:**
 - isTmax(2147483647) = 1
 - isTmax(1) = 0
 - **Legal operations:** !, ~, &, ^, |, +

Constraints:

- For each function, you can only use the specified legal bitwise operations.
- You cannot use loops, conditionals, or function calls.

Answers:

i

Programming is fun. And programming in C is even more fun.

This

is

even

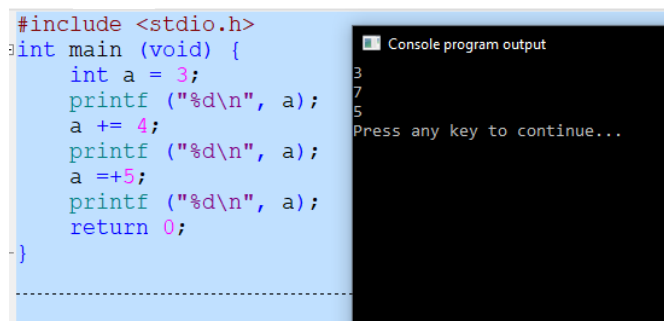
more

fun!

ii In the first case, arbitrary values are printed and depends on the compiler you might get a warning message, while in the second case the extra arguments are ignored. Thus, you should pay a close attention as you won't get an error message.

iii a

iv



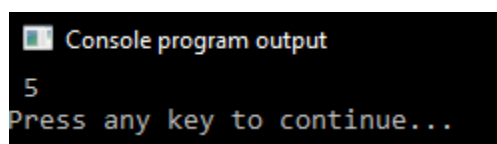
The screenshot shows a C program in a code editor on the left and its console output on the right. The code defines a variable 'a' as 3, prints it, increments it by 4 to 7, prints it, increments it by 5 to 12, and prints it again. The console output shows the values 3, 7, and 12 printed on separate lines, followed by the prompt 'Press any key to continue...'.

```
#include <stdio.h>
int main (void) {
    int a = 3;
    printf ("%d\n", a);
    a += 4;
    printf ("%d\n", a);
    a += 5;
    printf ("%d\n", a);
    return 0;
}
```

Console program output

```
3
7
12
Press any key to continue...
```

v



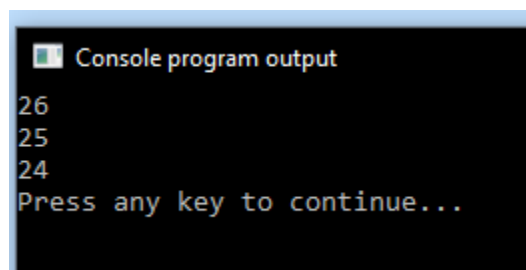
The screenshot shows a console window with the title 'Console program output'. It displays the number 5 on the first line and the prompt 'Press any key to continue...' on the second line.

Console program output

```
5
Press any key to continue...
```

c is 3 b+c is 5 and assigned back to b then assigned to a

vi



The screenshot shows a console window with the title 'Console program output'. It displays the numbers 26, 25, and 24 on three separate lines, followed by the prompt 'Press any key to continue...'.

Console program output

```
26
25
24
Press any key to continue...
```

vii

The boss has 20000 health points (HP)

SNES was a 16 bit console so this value was stored essentially in a short int.

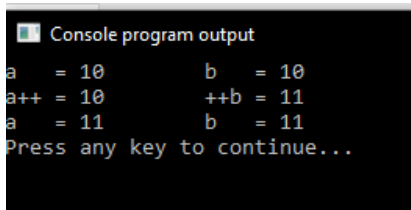
Thus, the boss' health is 0100 1110 0010 0000 in binary

The Elixir is a full heal item which adds the original maximum health to the current health then checks if the health is greater than max and adjusts current health to max health.

Thus, the final bosses HP becomes 1001 1100 0100 0000 which as a signed integer is -25536.

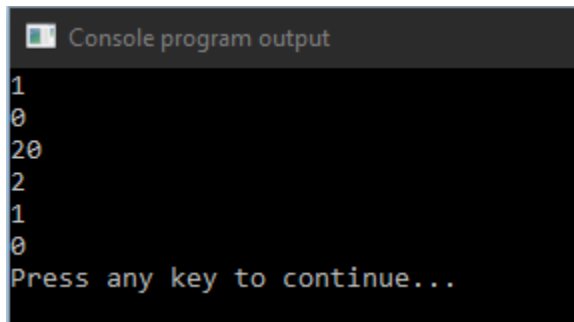
This is less than 0 hence the boss is dead.

viii



```
Console program output
a  = 10      b  = 10
a++ = 10     ++b = 11
a  = 11      b  = 11
Press any key to continue...
```

ix



```
Console program output
1
0
20
2
1
0
Press any key to continue...
```

x

```
busses = par/50 + ((par % 50) != 0);
```

xi 01100011

xii -51

xiii

- 1) Compilation error in line#7 \ should be /
- 2) Run time error in line #4 (or #7) result was not initialized.

xiv the function returns 1 if num equals $4*n+3$ when n is an integer.

Q10 of practice problems:

```
/*
 * bitXor - x^y using only ~ and &
 *   Example: bitXor(4, 5) = 1
 *   Legal ops: ~ &
 */
int bitXor(int x, int y) {
    return ~(x & y) & ~(~x & ~y);
}

/*
 * tmin - return minimum two's complement integer
 *   Legal ops: ! ~ & ^ | + << >>
 */
int tmin(void) {
    return 1 << 31;
}

/*
 * isTmax - returns 1 if x is the maximum, two's complement number,
 *   and 0 otherwise
 *   Legal ops: ! ~ & ^ | +
 */
int isTmax(int x) {
    return !((~x) ^ (x + 1)) & !(x + 1);
}
```