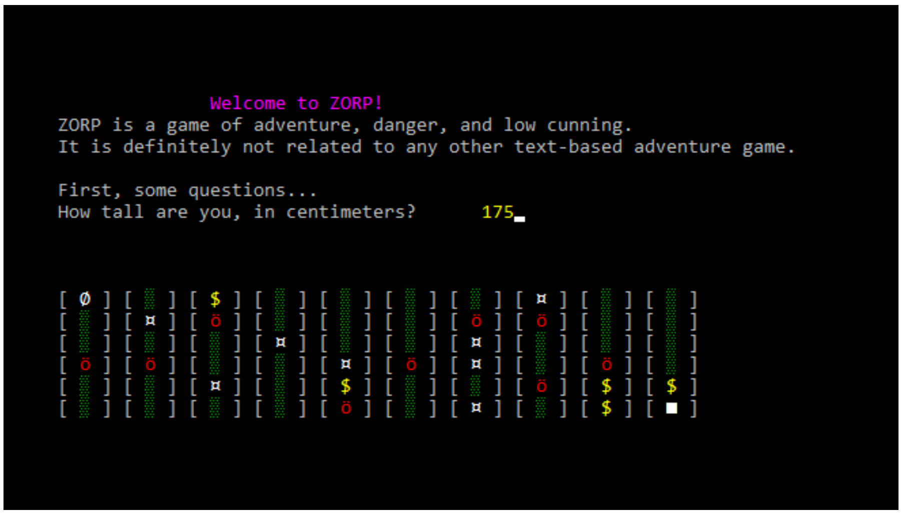
Tutorial – Conditional Statements: Zorp part 4


```
                Welcome to ZORP!
ZORP is a game of adventure, danger, and low cunning.
It is definitely not related to any other text-based adventure game.

First, some questions...
How tall are you, in centimeters?       175_


[ Ø ] [ ░ ] [ $ ] [ ░ ] [ ░ ] [ ░ ] [ ░ ] [ ¤ ] [ ░ ] [ ░ ]
[ ░ ] [ ¤ ] [ ö ] [ ░ ] [ ░ ] [ ░ ] [ ö ] [ ö ] [ ░ ] [ ░ ]
[ ░ ] [ ░ ] [ ░ ] [ ¤ ] [ ░ ] [ ░ ] [ ¤ ] [ ░ ] [ ░ ] [ ░ ]
[ ö ] [ ö ] [ ░ ] [ ░ ] [ ¤ ] [ ö ] [ ¤ ] [ ░ ] [ ö ] [ ░ ]
[ ░ ] [ ░ ] [ ¤ ] [ ░ ] [ $ ] [ ░ ] [ ░ ] [ ö ] [ $ ] [ $ ]
[ ░ ] [ ░ ] [ ░ ] [ ░ ] [ ö ] [ ░ ] [ ¤ ] [ ░ ] [ $ ] [ ■ ]
```

Introduction:

In this tutorial, we will use conditional statements to modify the output of each room according to its type.

Requirements:

You will need to have completed the following previous tutorials:

- Zorp Part 1: Variables. Available under the topic *Variables and Constants.*
- Zorp Part 2: Constants. Available under the topic *Variables and Constants.*
- Zorp Part 3: Arrays. Available under the topic *Arrays.*

ASCII Art:

At the moment, for each room we are only outputting the number that corresponds to the room type. This is a little awkward as we can't easily tell what is at each location in out map.

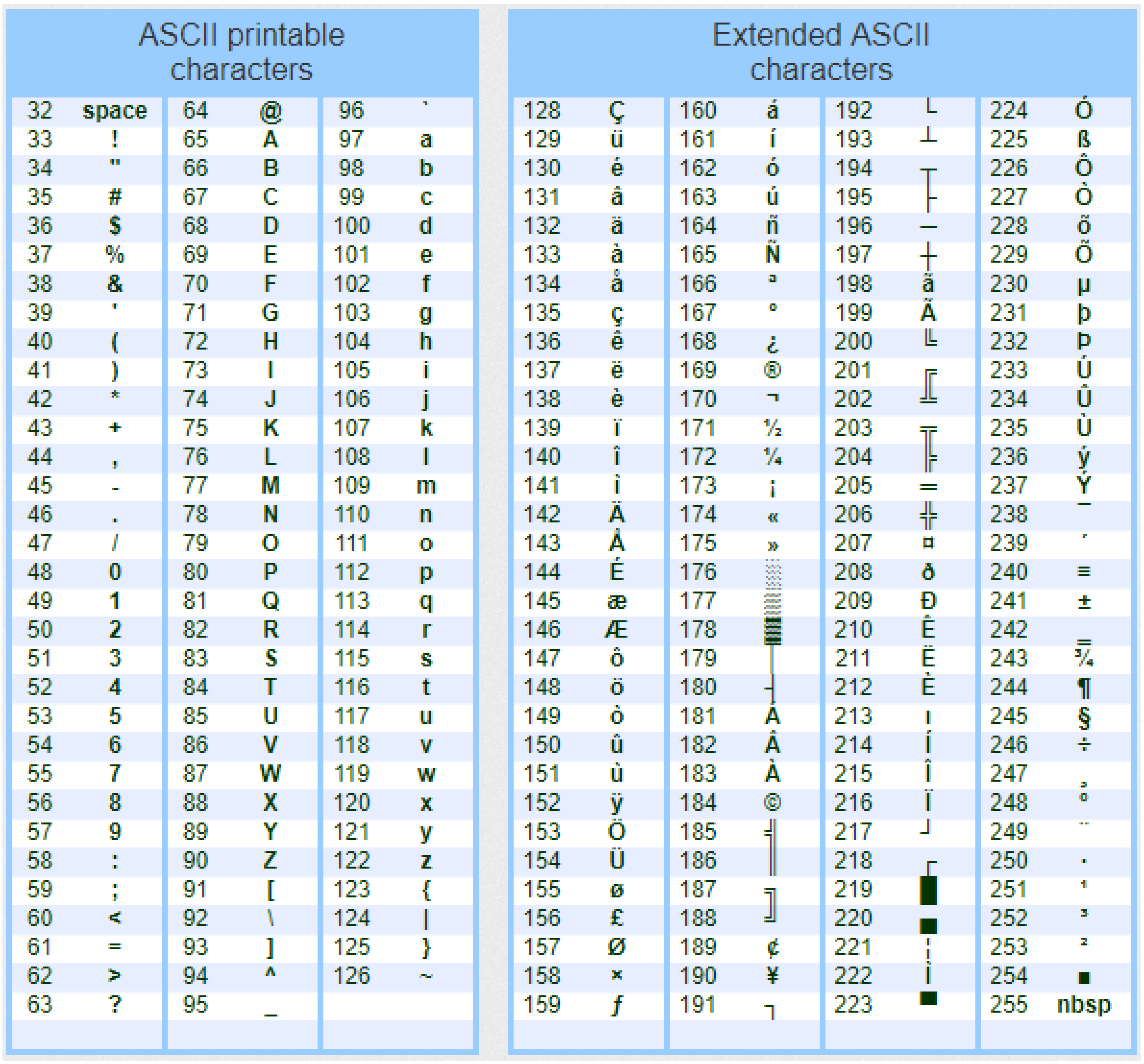
Instead of using numbers to represent the room type, we can use other characters.

ASCII is an acronym standing for *American Standard Code for Information Interchange*. It is a character encoding standard that maps the standard alpha-numeric characters to the values in a byte. It is the base character encoding used by essentially all computers.

Because a byte can store 256 unique numbers, and the alphabet plus numbers plus some punctuation only amounts to about half of that, there are about 127 free spaces.

Instead of leaving these blank, the ASCII standard specifies a collection of other characters and icons that we could also use.

Here is an extract of all of the characters in the ASCII table from values 32 to 256



ASCII printable characters					
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Extended ASCII characters							
128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

The full table is available here: http://www.theasciicode.com.ar/

Using this table we can find some characters that better represent what is in each room. We'll use the following characters for each room:

- Empty room: ░ decimal 176, hexadecimal 0xB0.
  If we change the text colour to green when we draw this character, it will kind of look like grass – so perhaps our empty 'room' will be a meadow.
- Enemy: ö decimal 148, hexadecimal 0x94.
  We'll colour this red so it looks like a face with its mouth open. The perfect enemy.
- Treasure: $ We can simply output this using the character on our keyboard.
- Food: ¤ decimal 169, hexadecimal 0xCF
  There isn't a good character that looks like food. This one is as good as any.

- Entrance: Ø decimal 157, hexadecimal 0x9D
  This sort of looks like the no-entry sign. So we'll use this to represent that the player can't exit through the entrance.
- Exit: ■ decimal 254, hexadecimal 0xFE
  Again, there's no good character to use to represent an exit, so this is as good as any.

We'll also use some more colours, so the difference between these ASCII characters is even greater.

Add the following constant variable definitions to your program:

```
const char* RED = "\x1b[91m";
const char* BLUE = "\x1b[94m";
const char* WHITE = "\x1b[97m";
const char* GREEN = "\x1b[92m";
```

These escape sequences can be found on this site: https://msdn.microsoft.com/en-us/library/windows/desktop/mt638032(v=vs.85).aspx#example

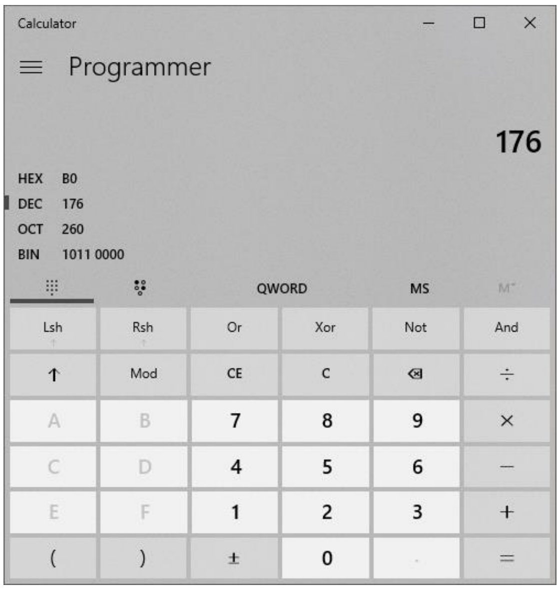
To output the ASCII characters that aren't on our keyboard, we need to use hexadecimal escape sequences. To do this, we use a forward slash followed by *x*, for example:

```
"\xb0"
```

This is the hexadecimal value 0xB0, or decimal 176 - the character ░.

To output a character using it's hexadecimal code, you use *x* followed by the hex value. Note that this must be output as a string (using the double quotation marks).

The ASCII table referenced above doesn't include the hexadecimal values. To find the hex values, you can use the calculator app on your computer by switching it to *programmer* mode:


Calculator
Programmer
176
HEX B0
DEC 176
OCT 260
BIN 1011 0000

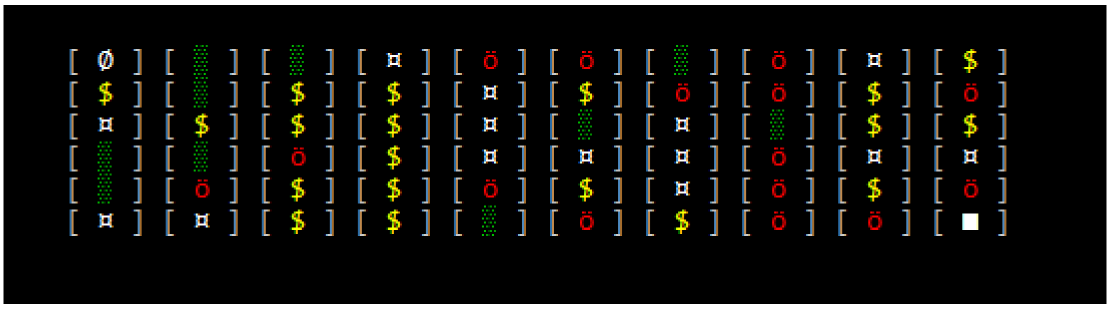
We now have everything we need to update the nested loop that draws the game map.

Update the code that outputs each room to the console as follows:

```
for (int y = 0; y < MAZE_HEIGHT; y++)
{
        std::cout << INDENT;
        for (int x = 0; x < MAZE_WIDTH; x++) {
                switch (rooms[y][x])
                {
                case EMPTY:
                        std::cout << "[ " << GREEN << "\xb0" << RESET_COLOR << " ] ";
                        break;
                case ENEMY:
                        std::cout << "[ " << RED << "\x94" << RESET_COLOR << " ] ";
                        break;
                case TREASURE:
                        std::cout << "[ " << YELLOW << "$" << RESET_COLOR << " ] ";
                        break;
                case FOOD:
                        std::cout << "[ " << WHITE << "\xcf" << RESET_COLOR << " ] ";
                        break;
                case ENTRANCE:
                        std::cout << "[ " << WHITE << "\x9d" << RESET_COLOR << " ] ";
                        break;
                case EXIT:
                        std::cout << "[ " << WHITE << "\xFE" << RESET_COLOR << " ] ";
                        break;
                }
        }
        std::cout << std::endl;
}
```

Tweaking Probabilities:

Running our game now will give us a map that looks like this:



```
[ Ø ] [ ░ ] [ ░ ] [ ¤ ] [ ö ] [ ö ] [ ░ ] [ ö ] [ ¤ ] [ $ ]
[ $ ] [ ░ ] [ $ ] [ $ ] [ ¤ ] [ $ ] [ ö ] [ ö ] [ $ ] [ ö ]
[ ¤ ] [ $ ] [ $ ] [ $ ] [ ¤ ] [ ░ ] [ ¤ ] [ ░ ] [ $ ] [ $ ]
[ ░ ] [ ░ ] [ ö ] [ $ ] [ ¤ ] [ ¤ ] [ ¤ ] [ ö ] [ ¤ ] [ ¤ ]
[ ░ ] [ ö ] [ $ ] [ $ ] [ ö ] [ $ ] [ ¤ ] [ ö ] [ $ ] [ ö ]
[ ¤ ] [ ¤ ] [ $ ] [ $ ] [ ░ ] [ ö ] [ $ ] [ ö ] [ ö ] [ ■ ]
```

This looks a bit busy. We want more empty rooms so that getting treasure or food is more meaningful. Also, by making fewer enemies then each enemy encounter can be more dangerous and exciting.

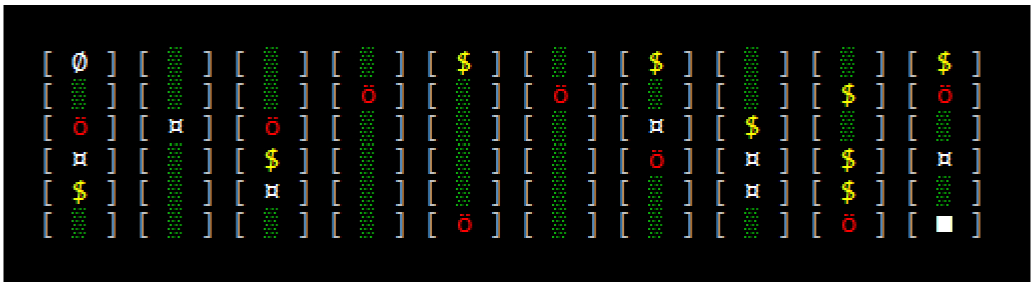
What we want is a way to tweak the random generation so that we have a higher probability of getting an empty room.

We'll handle this in a very simple way. We'll double the range of values that the random function generates, and if the random function gives us a number that isn't in our range of room types then we'll make those rooms empty rooms too.

Here is the code that achieves this:

```
	// fill the arrays with random room types
	for (int y = 0; y < MAZE_HEIGHT; y++)
	{
		for (int x = 0; x < MAZE_WIDTH; x++) {
			int type = rand() % (MAX_RANDOM_TYPE * 2);
			if (type < MAX_RANDOM_TYPE)
				rooms[y][x] = type;
			else
				rooms[y][x] = EMPTY;
		}
	}
```

Here are the results of our update:



```
[ Ø ] [ ▒ ] [ ▒ ] [ ▒ ] [ $ ] [ ▒ ] [ $ ] [ ▒ ] [ ▒ ] [ $ ]
[ ▒ ] [ ▒ ] [ ▒ ] [ ö ] [ ▒ ] [ ö ] [ ▒ ] [ ▒ ] [ $ ] [ ö ]
[ ö ] [ ¤ ] [ ö ] [ ▒ ] [ ▒ ] [ ▒ ] [ ¤ ] [ $ ] [ ▒ ] [ ▒ ]
[ ¤ ] [ ▒ ] [ $ ] [ ▒ ] [ ▒ ] [ ▒ ] [ ö ] [ ¤ ] [ $ ] [ ¤ ]
[ $ ] [ ▒ ] [ ¤ ] [ ▒ ] [ ▒ ] [ ▒ ] [ ▒ ] [ ¤ ] [ $ ] [ ▒ ]
[ ▒ ] [ ▒ ] [ ▒ ] [ ▒ ] [ ö ] [ ▒ ] [ ▒ ] [ ▒ ] [ ö ] [ ■ ]
```

This gives us a much nicer level map that is now nicely coloured with some easy to understand symbols indicating exactly what is in each room of the map.