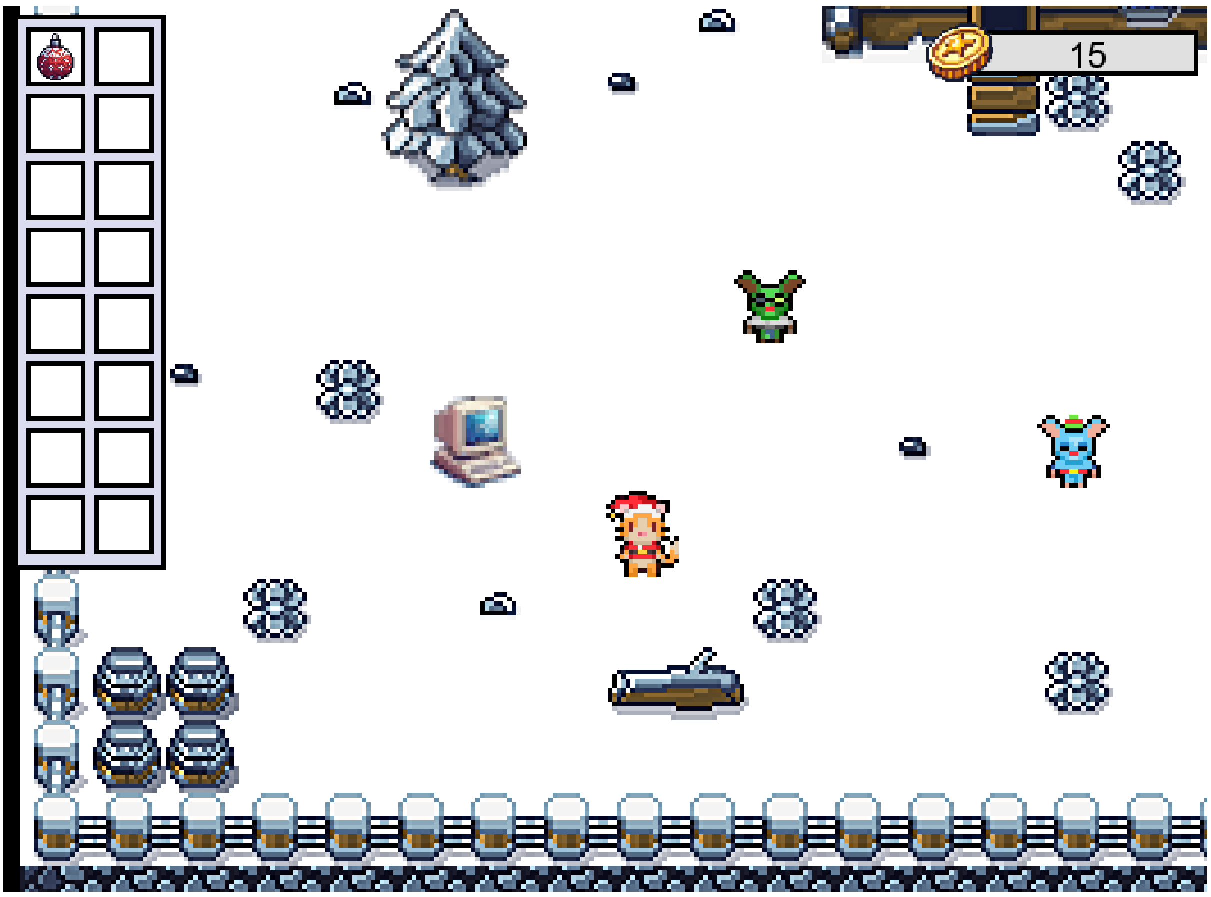
The Game

In this game, you'll play as CatOrMouse. Your objective is to save Christmas by buying the star for your Christmas tree from Van Frosty. In addition to the star, you can buy as many ornaments as you can carry to decorate your tree. To gain money to buy things, you can use the computer to do online freelance programming jobs.

You can also speak to Van Holly to change your name for a fee of 1 coin per character. He says that this is totally *not* a scam. He will actually start calling you by your new name. He is, after all, into identity management.



Is This a Bug

Before the training even starts, Van Jolly approaches McHoneyBell and says that they've been observing some weird behaviours while playing the game. They think the Ghost of Christmas Past is haunting it.

McHoneyBell asks them to reproduce what they saw. Van Jolly boots up the game and does the following (which you are free to replicate, too):

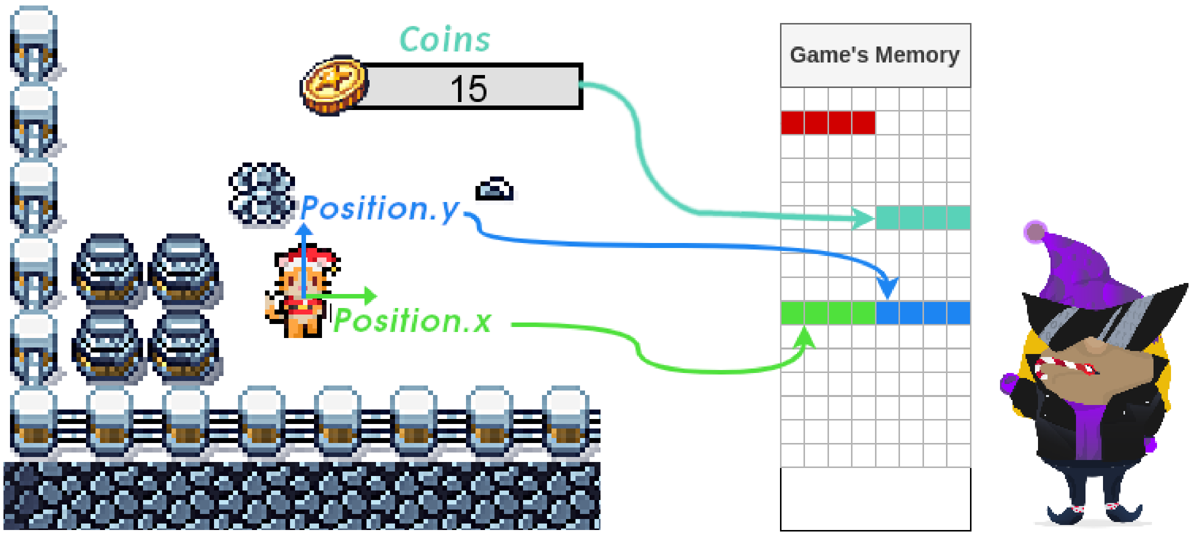
1. Use the computer until you get 13 coins.
2. Ask Van Holly to change your name to `scroogerocks!`
3. Suddenly, you have 33 coins out of nowhere.

Van Jolly explains that when you change your name to anything large enough, the game goes nuts! Sometimes, you'll get random items in your inventory. Or, your coins just disappear. Even the dialogues can stop working and show random gibberish. This must surely be the work of magic!

McHoneyBell doesn't look convinced. After some thinking, she seems to know what this is all about.

Memory Corruption

Remember that whenever we execute a program (this game included), all data will be processed somehow through the computer's **RAM (random access memory)**. In this videogame, your coin count, inventory, position, movement speed, and direction are all stored somewhere in the memory and updated as needed as the game goes on.



Usually, each variable stored in memory can only be manipulated in specific ways as the developers intended. For example, you should only be able to modify your coins by working on the PC or by spending money either in the store or by changing your name. In a well-programmed game, you shouldn't be able to influence your coins in any other way.

But what happens if we can indirectly change the contents of the memory space that holds the coin count? What if the game had a flaw that allows you to overwrite pieces of memory you are not supposed to? Memory corruption vulnerabilities will allow you to do that and much more.

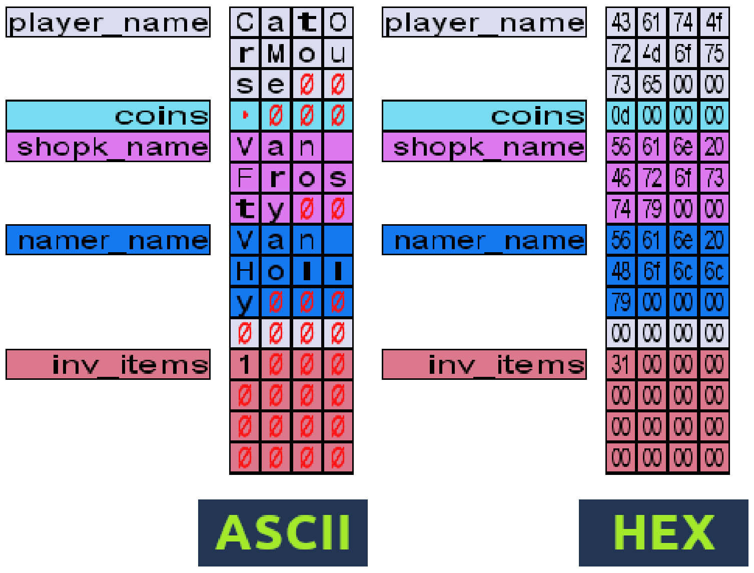
Honeybell says a debugger will be needed to check the memory contents while the game runs. On hearing that, Van Sprinkles says they programmed a debug panel into the game that does exactly that. This will make it easier for us!

Accessing the Debug Panel

While they were developing this game, the Frostlings added debugging functionality to watch the memory layout of some of the game's variables. They did this because they couldn't understand why the game was suddenly crashing or behaving strangely. To access this hidden memory monitor, just press `TAB` in the game.

You can press `TAB` repeatedly to cycle through two different views of the debugging interface:

- **ASCII view:** The memory contents will be shown in ASCII encoding. It is useful when trying to read data stored as strings.
- **HEX view:** The memory contents will be shown in HEX. This is useful for cases where the data you are trying to monitor is a raw number or other data that can't be represented as ASCII strings.



player_name	C	a	t	O
	r	M	o	u
	s	e	Ø	Ø
coins	·	Ø	Ø	Ø
shopk_name	V	a	n	
	F	r	o	s
	t	y	Ø	Ø
namer_name	V	a	n	
	H	o	l	l
	y	Ø	Ø	Ø
	Ø	Ø	Ø	Ø
inv_items	1	Ø	Ø	Ø
	Ø	Ø	Ø	Ø
	Ø	Ø	Ø	Ø
	Ø	Ø	Ø	Ø

ASCII

player_name	43	61	74	4f
	72	4d	6f	75
	73	65	00	00
coins	0d	00	00	00
shopk_name	56	61	6e	20
	46	72	6f	73
	74	79	00	00
namer_name	56	61	6e	20
	48	6f	6c	6c
	79	00	00	00
	00	00	00	00
inv_items	31	00	00	00
	00	00	00	00
	00	00	00	00
	00	00	00	00

HEX

Viewing the contents in RAM will prove helpful for understanding how memory corruption occurs, so be sure to check the debug panel for each action you make in the game. Remember, you can always hide the debug panel by pressing `TAB` until it closes.

Investigating the "scroogerocks!" Case

Armed with the debugging panel, McHoneyBell starts the lesson. As a first step, she asks you to restart your game (refreshing the website should work) and open the debug interface in HEX mode. The Frostlings have labelled each of the variables stored in memory, making it easy to trace them.

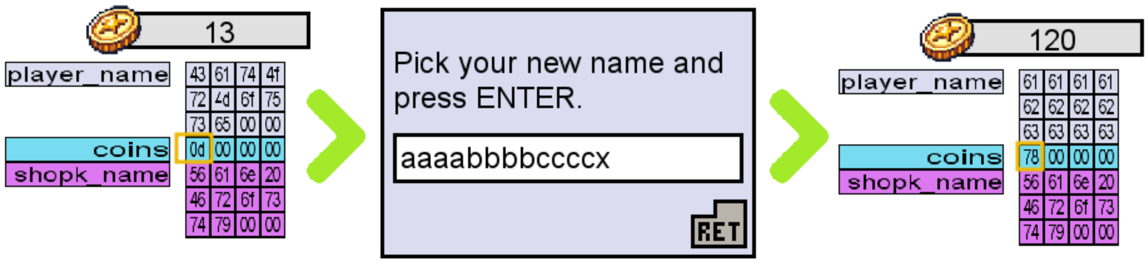
McHoneyBell wants you to focus your attention on the `coins` variable. Go to the computer and generate a coin. As expected, you should see the coin count increase in the user interface and the debug panel simultaneously. We now know where the coin count is stored.

McHoneyBell then points out that right before the `coins` memory space, we have the `player_name` variable. She also notes that the `player_name` variable only has room to accommodate 12 bytes of information.

"But why does this matter at all?" asks a confused Van Twinkle. "Because if you try to change your name to `scroogerocks!`, you would be using 13 characters, which amounts to 13 bytes," replies McHoneyBell. Van Twinkle, still perplexed, interrupts: "So what would happen with that extra byte at the end?" McHoneyBell says: "It will overflow to the first byte of the coins variable."

To prove this point, McHoneyBell proposes replicating the same experiment, but this time, we will get 13 coins and change our names to `aaaabbbbccccx`. Meanwhile, we'll keep our eyes on the debug panel. Let's try this in our game and see what happens.

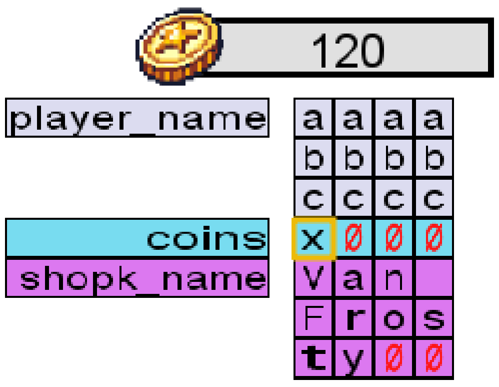
All of a sudden, we have 120 coins! The memory space of the `coins` variable now holds `78`.



```
43 61 74 4f
72 4d 6f 75
73 65 00 00
0d 00 00 00
56 61 6e 20
46 72 6f 73
74 79 00 00
```

```
61 61 61 61
62 62 62 62
63 63 63 63
78 00 00 00
56 61 6e 20
46 72 6f 73
74 79 00 00
```

Remember that `0x78` in hexadecimal equals `120` in decimal. To make this even clearer, let's switch the debug panel to ASCII mode:



```
a a a a
b b b b
c c c c
x Ø Ø Ø
V a n
F r o s
t y Ø Ø
```

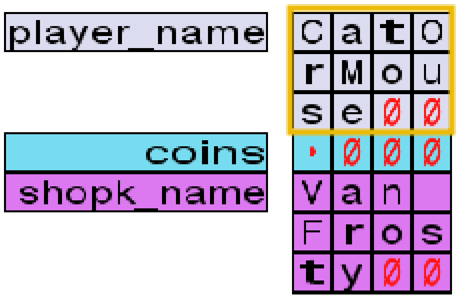
The `x` at the end of our new name spilt over into the `coins` variable. The ASCII hexadecimal value for `x` is `0x78`, so the coin value was changed to `0x78` (or `120` in decimal representation).

As you can see, McHoneyBell's predictions were correct. The game doesn't check if the `player_name` variable has enough space to store the new name. Instead, it keeps writing to adjacent memory, overwriting the values of other variables. This vulnerability is known as a **buffer overflow** and can be used to corrupt memory right next to the vulnerable variable.

Buffer overflows occur in some programming languages, mostly C and C++, where the variables' boundaries aren't strict. If programmers don't check the boundaries themselves, it's possible to abuse a variable to read or write memory beyond the space initially reserved for it. Our game is written in C++.

Strings in More Detail

By now, the Frostlings look baffled. It never occurred to them that they should check the size of a variable before writing to it. Van Twinkle has another question. When the game started, the main character's name was `CatOrMouse`, which only uses 10 characters.

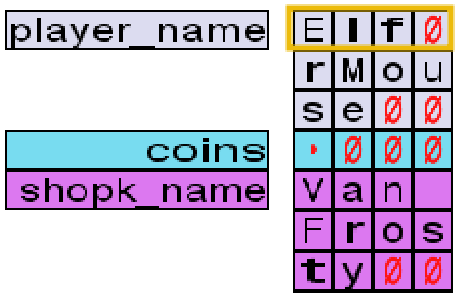


How does the game know the length of a string if no boundary checks are performed on the variable?

To explain this, McHoneyBell asks us to do the following:

1. Restart the game.
2. Get at least 3 coins.
3. Change your name to `Elf`.

As a result, your memory layout should look like this:



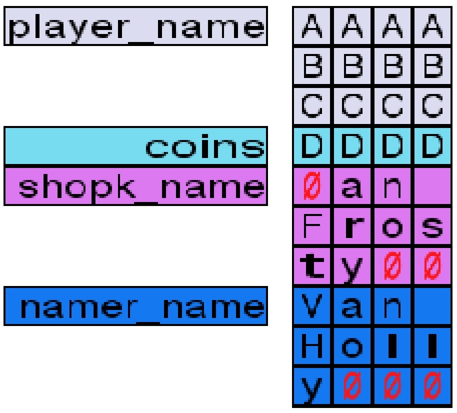
When strings are written to memory, each character is written in order, taking 1 byte each. A NULL character, represented in our game by a red zero, is also concatenated at the end of the string. A **NULL character** is simply a byte with the value 0x00, which can be seen by changing the debug panel to hex mode.

When reading a variable as a string, the game will stop at the first NULL character it finds. This allows programmers to store smaller strings into variables with larger capacities. Any character appearing after the NULL byte is ignored, even if it has a value.

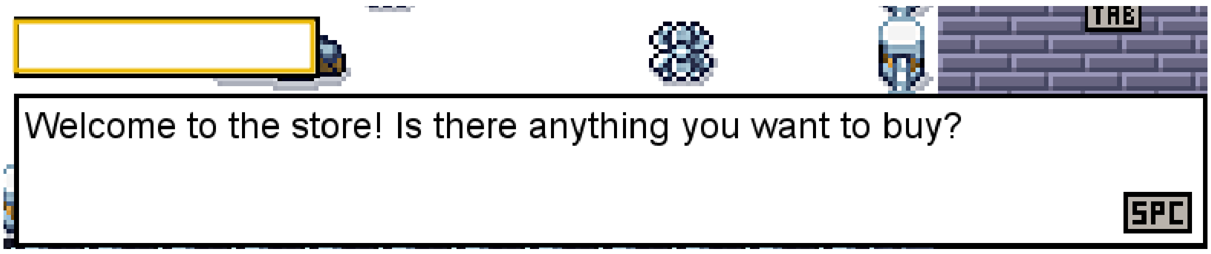
To better explain all of this, McHoneyBell proposes a second experiment on strings:

1. Get 16 coins.
2. Rename yourself to `AAAABBBBCCCCDDDD` (16 characters).

Now, your memory layout should look like this:

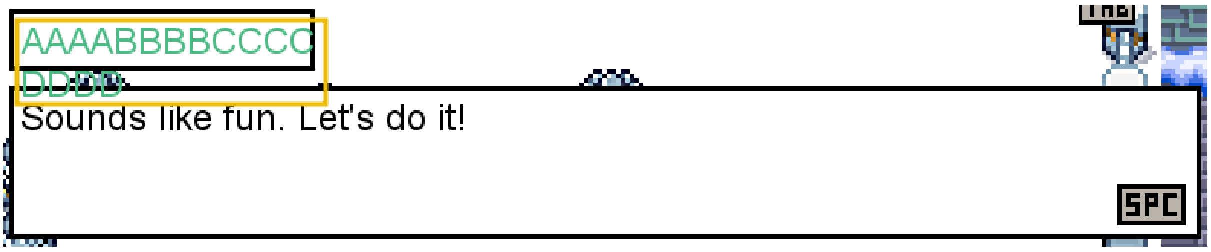


Notice how the game adds a NULL character after your 16 bytes, which overwrites the `shopk_name` variable. If you talk to the shopkeeper, you should see his name is empty.


Welcome to the store! Is there anything you want to buy?

This happens because the game reads from the start of the variable up to the first NULL byte, which appears in the first byte in our example. Therefore, this is equivalent to having an empty string.

On the other hand, if you talk to Van Holly, you should see your own name is now `AAAABBBBCCCCDDDD`, which is 16 characters long.



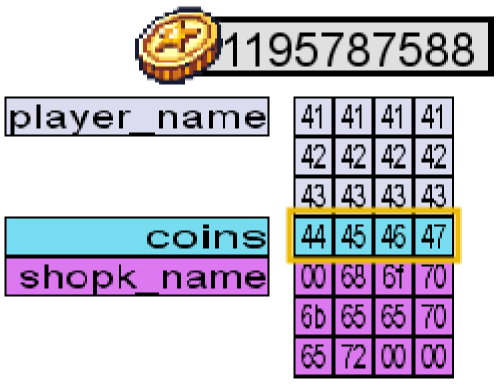
Sounds like fun. Let's do it!

Since C++ doesn't check variable boundaries, it reads your name from the start of the `player_name` variable to the first NULL byte it finds. That's why your name is now 16 characters long, even though the `player_name` variable should only fit 12 bytes.

Part of your name now overlaps with the coins variable, so, if you spend some money in the shop, your visible name will also change. Buy some items and see what happens!

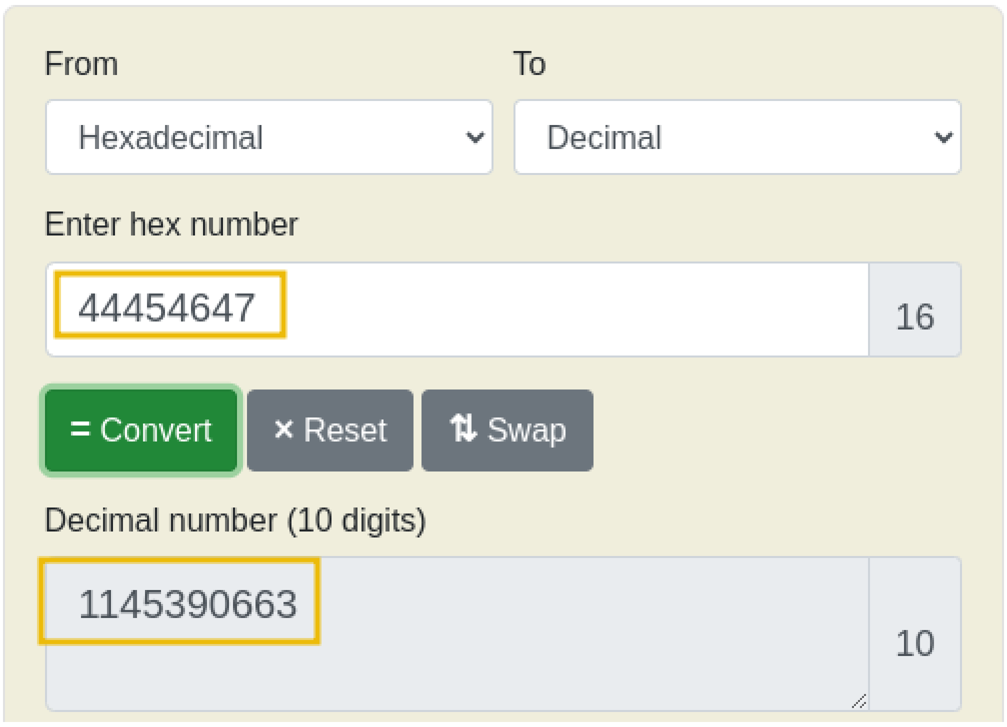
Integers and the Coins Variable

Van Twinkle mistyped the name during the previous experiment and ended up with `AAAABBBBCCCCDEFG`. They then noticed that they had `1195787588` coins in the upper right corner, shown as follows in the debug panel:



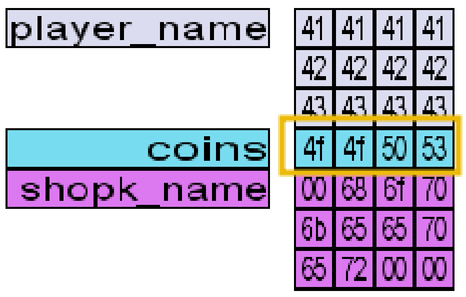
player_name	41	41	41	41
	42	42	42	42
	43	43	43	43
coins	44	45	46	47
shopk_name	00	68	6f	70
	6b	65	65	70
	65	72	00	00

Out of curiosity, they used an [online tool](#) that converts hexadecimal to decimal numbers to check if the hexadecimal number from the debug panel matched their coin count. To their surprise, the numbers were different:



McHoneyBell explains that integers in C++ are stored in a very particular way in memory. First, integers have a fixed memory space of 4 bytes, as seen in the debug panel. Secondly, an integer's bytes are stored in reverse order in most desktop machines. This is known as the **little-endian** byte order.

Let's use an example to understand this better. If you take your current coin count of `1195787588` and convert that number to hex, you'll obtain `0x[47 46 45 44]`, corresponding to what's shown by the debug panel but backwards. How many coins would you have if the hex value of the coins variable was showing in memory as follows? Input your answer at the end of the task!


player_name
41 41 41 41
42 42 42 42
43 43 43 43
coins
4f 4f 50 53
shopk_name
00 68 6f 70
6b 65 65 70
65 72 00 00