COP 4530 Pointer

TA: Liting Zhang

Florida State University

Oct 19, 2023

What you need to know about pointers

- A pointer is a memory address
- Every location in memory, and therefore every variable, has an address.
- Every address corresponds to a unique location in memory.
- Given a memory address, the computer can find out what value is stored at that location.

Pointer Syntax #1

To declare a pointer of a particular type, use the * (asterisk) symbol

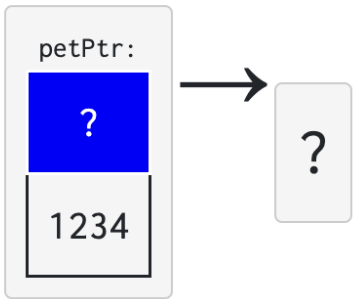
```
1  string* petPtr;  // declare a pointer to a string
2  int* agePtr;     // declare a pointer to an int
3  char* letterPtr; // declare a pointer to a char
```

The type for **petPtr** is a **string*** and not a string. This is important! A pointer type is distinct from the pointee type.

Pointer Syntax #2

To get the address of another variable, use the (ampersand) symbol. This is not a reference! It is the same symbol, but does not mean the same thing.

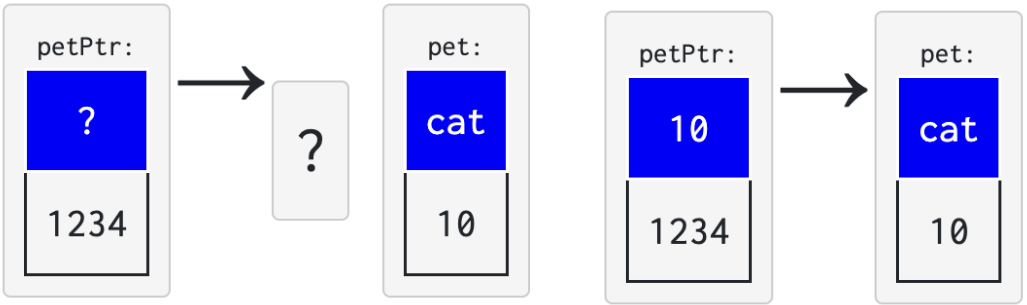
```
1  string* petPtr;
2  // declare a pointer (which will hold a memory address) to a string
3  // at this point, petPtr's value is bogus!
4  // petPtr does have its own address, which we
5  // will pretend is 1234
```




```
1  string pet = "cat";
2  //a string variable, pretend it is at memory location 10
```

Pointer Syntax #2

Now we have

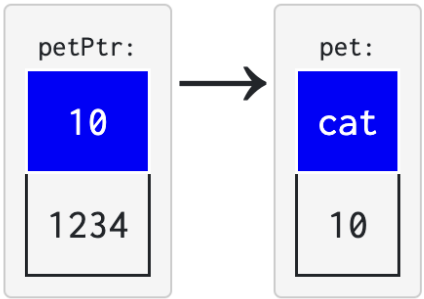


- To get **petPtr** to point to **pet**, we have to assign the address of **pet** to **petPtr**. We do this like any other assignment, except that we use the &symbol:

```
1  petPtr = &pet; // petPtr now holds the address of pet
```

- Notice that **petPtr**'s value is 10, which is the address of pet.
- **petPtr** is a pointer, which means that its value is an address.

Pointer Syntax #3



- To get value of the variable a pointer points to, use the * (asterisk) character (in a different way than before!):

```
1  string* petPtr; // declare a pointer to a string
2  string pet = "cat"; // a string variable, pretend it is at memory location 10
3  petPtr = &pet; // petPtr now holds the address of pet
4  cout << *petPtr << endl; // prints out "cat"
```

- When we use * in this way, we say that we are *dereferencing* the pointer, which follows the address to its location and gets what is at that location.

Pointers are numbers

Pointers are just numbers that are associated with a type

```
1  int x = 4; // pretend x has some address, which is a number
2  int y;
3  y = x; // what is y's value? It's 4;
4
5  cout << "x: " << x << ", y:" << y << endl;
6
7  int *xPtr;
8  int *yPtr;
9
10 xPtr = &x;   // what is xPtr's value? It is the address of x, some number
11 yPtr = xPtr; // what is yPtr's value? It is also the address of x, the same numbe
12
13 // we need to cast to a size_t below so we print out a regular number
14 cout << "xPtr: " << (size_t)xPtr << ", yPtr:" << (size_t)yPtr << endl;
```

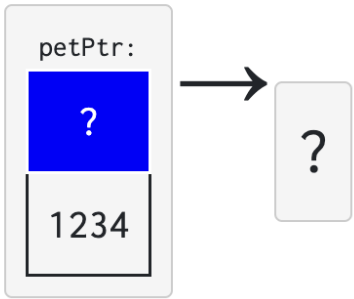
output

```
1  x: 4, y:4
2  xPtr: 123145559043436, yPtr:123145559043436
```

Pointer Tips #1

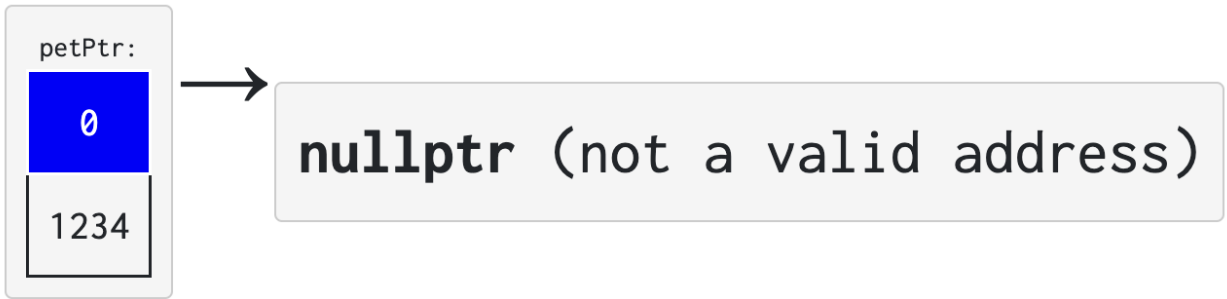
To ensure that we can tell if a pointer has a valid address or not, set your declared pointer to nullptr, which means "no valid address" (it actually is just 0 in C++).

Instead of this




```
1 | string* petPtr; // declare a pointer to a string
```

We do this, instead:




```
1 | string* petPtr = nullptr;
2 | // declare a pointer to a string that points to nullptr
```

Pointer Tips #2

If you are unsure if your pointer holds a valid address, you should check for nullptr

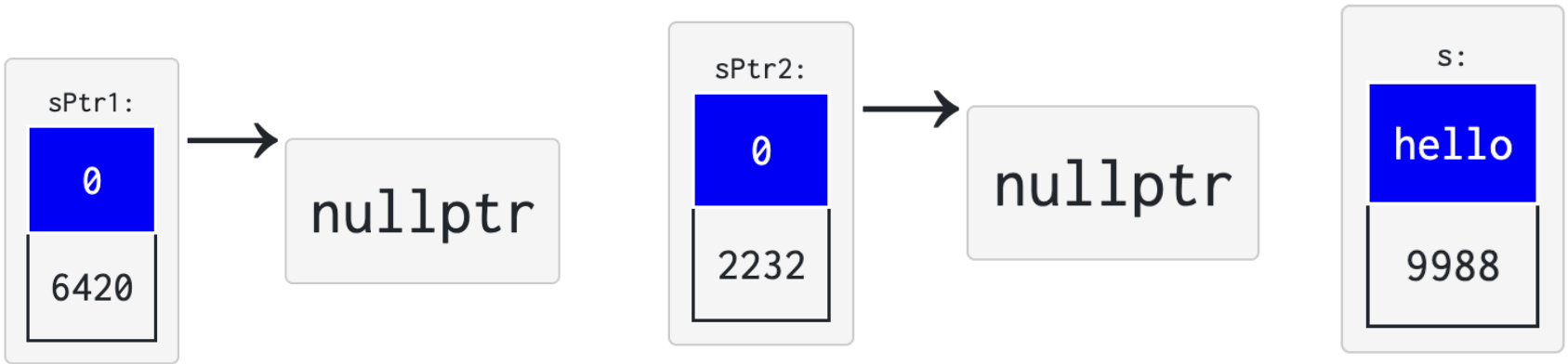
```
1  void printPetName(string* petPtr) {
2      if (petPtr != nullptr) {
3          cout << *petPtr << endl; // prints out the value pointed to by petPtr
4                                   // if it is not nullptr
5      } else {
6          cout << "petPtr is not valid!" << endl;
7      }
8  }
```

When you dereference a nullptr, you seg fault!

Pointer Pratice

If you set one pointer equal to another pointer, they both point to the same variable.

```
1  string* sPtr1 = nullptr;
2  string* sPtr2 = nullptr;
3  string s = "hello";
```



Pointer Pratice

If you set one pointer equal to another pointer, they both point to the same variable.

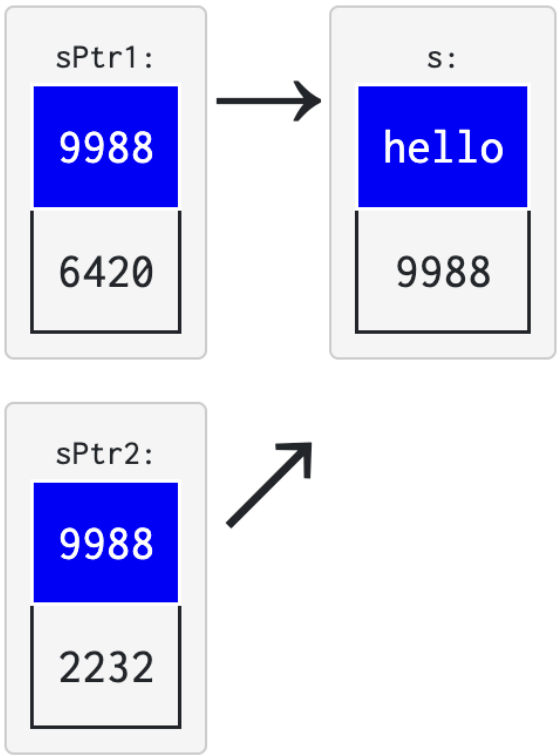
```
1  string* sPtr1 = nullptr;
2  string* sPtr2 = nullptr;
3  string s = "hello";
4  sPtr1 = &s;
5  cout << *sPtr1 << endl;
```

output

```
1  hello
```

Pointer Pratice

```
1  string* sPtr1 = nullptr;
2  string* sPtr2 = nullptr;
3  string s = "hello";
4  sPtr1 = &s;
5  cout << *sPtr1 << endl;
6  sPtr2 = sPtr1;
7  cout << *sPtr2 << endl;
```

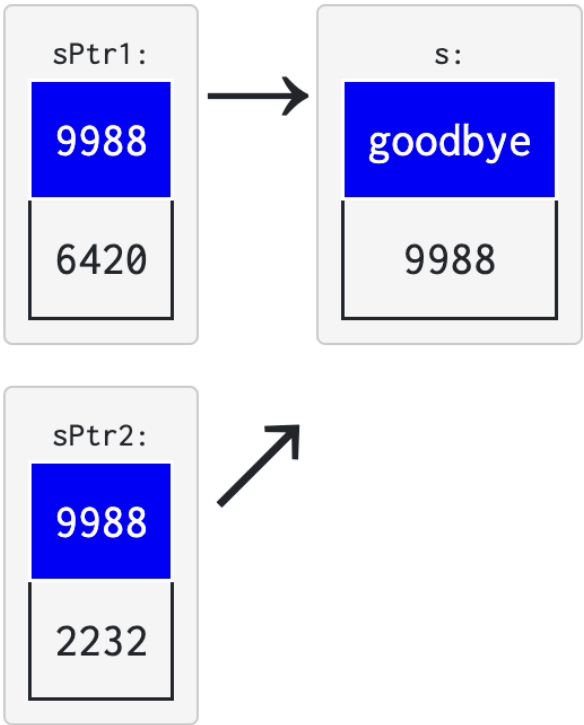


output

```
1  hello
```

Pointer Pratice

```
1  string* sPtr1 = nullptr;
2  string* sPtr2 = nullptr;
3  sPtr1 = &s;
4  cout << *sPtr1 << endl;
5  sPtr2 = sPtr1;
6  *sPtr1 = "goodbye";
7  cout << *sPtr1 << " " <<  *sPtr2 << endl;
```



output

```
1  goodbye goodbye
```

More info about Addresses

Addresses are just numbers, as we have seen. However, you will often see an address listed like this:

```
0x7fff3889b4b4
```

or this:

```
0x602a10
```

This is a base-16, hexadecimal representation. The 0x just means "the following number is in hexadecimal notation."
The letters are used because base 16 needs 16 digits:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
```

Reference:
https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1206/lectures/pointers/