Memory and arrays

Jack Garner

August 23, 2020

Stack

main: {int a, double x, bool b}

Stack

main: {int a, double x, bool b}

function1: {int c, double d, string s}

Stack

main: {int a, double x, bool b}

function1: {int c, double d, string s}

function2: {string st}

Stack

main: {int a, double x, bool b}

function1: {int c, double d, string s}

function2: {string st}

- Everything on the stack must have a known size at compile time
- Every data type has a known size and can be put on the stack

Pointers

Memory in your computer is divided into addresses. We can get the value of a variable or the address of the value.

```
1  int y = 5
2  int *x = &y;
3  cout << y << endl; // 5
4  cout << x << endl; // Some number n
5  cout << &y << endl; // Some number n
6  cout << *x << endl; // 5
```

Pointers

Memory in your computer is divided into addresses. We can get the value of a variable or the address of the value.

```
1  int y = 5
2  int *x = &y;
3  cout << y << endl; // 5
4  cout << x << endl; // Some number n
5  cout << &y << endl; // Some number n
6  cout << *x << endl; // 5
```

main: {int y, int* x}

Arrays

```
1  int main() {
2    int x[5] = {0, 1, 2, 3, 4};
3    cout << x[0] << endl; // return 0
4    cout << x[4] << endl; // return 4
5    cout << x[10] << endl; // best case scenario: it will crash
6  }
```

Arrays

```
1  int main() {
2    int x[5] = {0, 1, 2, 3, 4};
3    cout << x[0] << endl; // return 0
4    cout << x[4] << endl; // return 4
5    cout << x[10] << endl; // best case scenario: it will crash
6  }
```

main: {int[] x, int, int, int, int, int}

Arrays

```
1  int main() {
2    int x[5] = {0, 1, 2, 3, 4};
3    int *p = x;
4    cout << p[0] << endl; // return 0
5    cout << p[4] << endl; // return 4
6    cout << p[10] << endl; // best case scenario: it will crash
7  }
```

Arrays

```
1  int main() {
2    int x[5] = {0, 1, 2, 3, 4};
3    int *p = x;
4    cout << *p << endl; // return 0
5    cout << *(p+4) << endl; // return 4
6    cout << *(p+10) << endl; // best case scenario: it will crash
7  }
```

Arrays

```
1  int main() {
2    int x[5] = {0, 1, 2, 3, 4};
3    int *p = x;
4    cout << *p << endl; // return 0
5    cout << *(p+4) << endl; // return 4
6    cout << *(p+10) << endl; // best case scenario: it will crash
7  }
```

If arrays are pointers in disguise, and the stack needs to have a known size at compile time, how do we handle arrays of an unknown length?

Heap

The heap has a structure that doesn't require knowing the size at compile time.

```
1  int main() {
2    int x;
3    cin >> x;
4    int *p = new int[x];
5    ...
6  }
```

Heap

The heap has a structure that doesn't require knowing the size at compile time.

```
1  int main() {
2    int x;
3    cin >> x;
4    int *p = new int[x];
5    ...
6  }
```

Any variable can be prefixed with new to put it on the heap. If you remember new from Java, it actually does pretty much the exact same thing.

Heap

Unlike Java, C++ doesn't have a garbage collector. The heap must be cleaned up after being used.

```
1  int main() {
2    int x;
3    cin >> x;
4    int *p = new int[x];
5    ...
6    free(p); // without this, you can leak memory
7  }
```

Heap

Vectors (as we saw last week) wrap all of the memory stuff for you.
Vectors are kind of like ArrayList in Java.

Try it!

Write a program which takes in ints until the user gives you the int 0. After the user types in 0, all of the ints should be in an array. How do you deal with having an unknown number of elements in your array?