Anyone pursuing a degree in CS?

Lecture 3.1: Pointers

CS2308
Gentry Atkinson

Keeping Values in Memory

- A computer’s memory can be thought of as one long list of integers.
- Variables are named locations in memory.
- How does the computer find that location though?

Memory Addresses

- Every byte of memory has one fixed address.
- Most variables occupy more than one byte.
- A variable’s address is the address of its first byte.

Accessing Memory Addresses

- The & operator can be used to access the address of a variable.
- Addresses are show in “hexadecimal”.

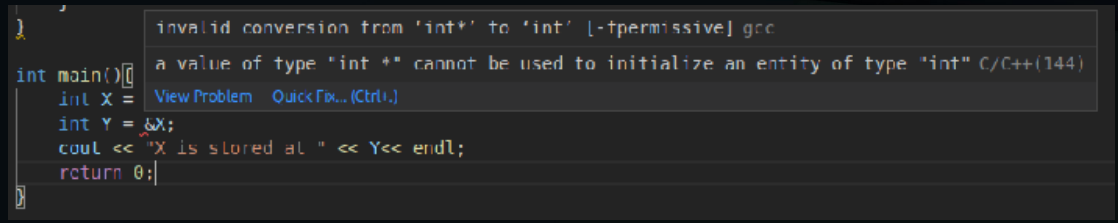
```
int main(){
    int X = 23;
    cout << "X is stored at " << &X << endl;
    return 0;
}
```

```
X is stored at 0x7fffffffdc94
```

Storing an Address in a Variable

- An address is just a number.
- So logically, I should be able to store a memory address in a variable.

```
int main(){
    int X = 23;
    int Y = &X;
    cout << "X is stored at " << Y<< endl;
    return 0;
}
```



Pointers

- Pointers are variables that hold the memory addresses of other variables.
- A pointer can only point to variables of a particular data type.

Pointers are References

- Changing the value stored in a pointer, changes the address it’s pointing to.
- The * operator lets you change the value in the address it’s pointing to.

```
int main(){
    int X = 23;
    int* Y = &X;
    cout << "Value in X: " << *Y << endl;
    return 0;
}
```

```
Value in X: 23
```

Declaring Pointers

- Every pointer should be given a specific data type.
- Adding * between the datatype and the name of a variable makes it a ptr:
  - char * c;

De-referencing Pointers

- Accessing the location that a pointer points to is called “de-referencing”.
- * is called the de-referencing operator.
- You can think of * as meaning “in direction” or “at the location”.

Example 1

```
int main(int argc, char** argv){
    int a = 1;
    int* b = &a;
    *b = 2*a;
    *b = 2*(*b);
    cout << "a is now " << *b << endl;
    return 0;
} //try to predict the output
```

Pointer Arithmetic

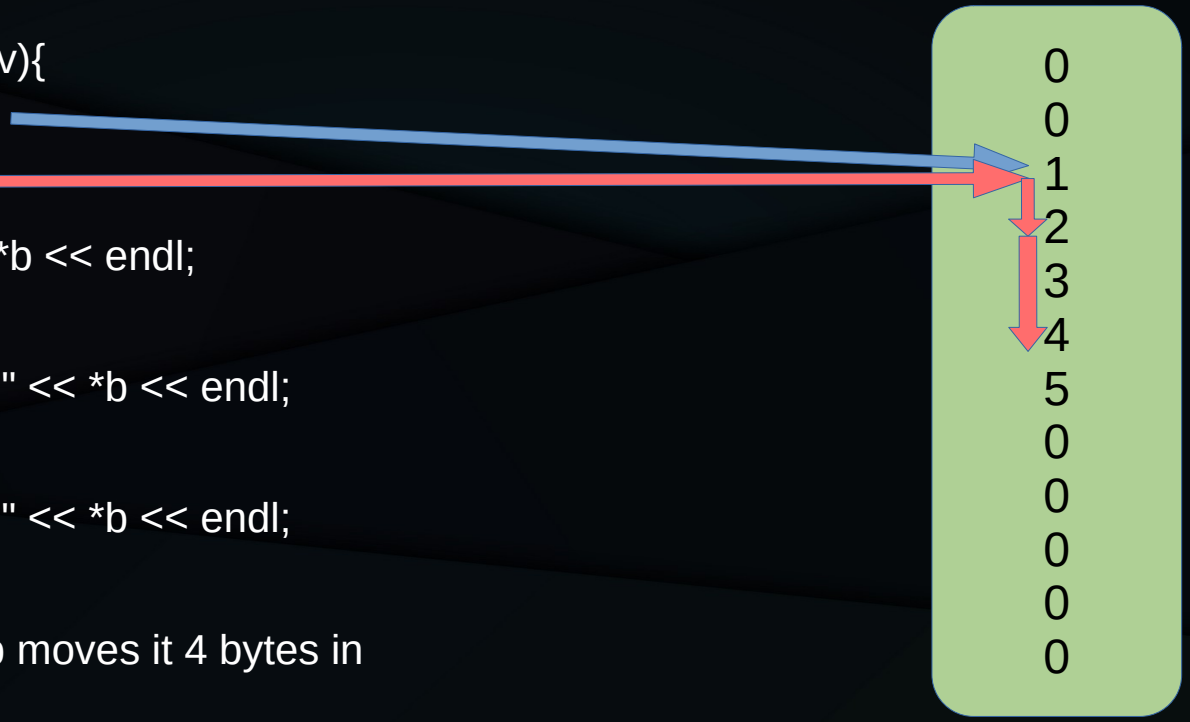
- Using operators to change the value stored in a pointer (the location it points to) is called “pointer arithmetic”.
- Pointers can be made to point to any memory location, meaning that we have to be very careful with with pointer arithmetic.

Example 2

```
int main(int argc, char** argv){
    int a[] = {1, 2, 3, 4, 5};
    int* b = a;
    cout << "b points to " << *b << endl;
    b++;
    cout << "Now b points to " << *b << endl;
    b+=2;
    cout << "Now b points to " << *b << endl;
    return 0;
} //try to predict the output
```

Example 2 Walkthrough

```
int main(int argc, char** argv){
    int a[] = {1, 2, 3, 4, 5};
    int* b = a;
    cout << "b points to " << *b << endl;
    b++;
    cout << "Now b points to " << *b << endl;
    b+=2;
    cout << "Now b points to " << *b << endl;
    return 0;
} //notice the incrementing b moves it 4 bytes in
memory.
```



The Terrible Truth about Arrays

```
int main(int argc, char** argv){
    int a[] = {1, 2, 3, 4, 5};
    cout  << "Address of array first element: " << &a[0] << endl;
    cout << "Value stored in a: " << a << endl;
    cout << "First element in a: " << a[0] << endl;
    cout << "Dereferencing a: " << *a << endl;
    cout << "Second element in a: " << a[1] << endl;
    cout << "Dereferencing a+1: " << *(a+1) << endl;
    return 0;
} //try to predict the output
```

Arrays are Secretly Pointers

- Really, they’re just similar.
- An array variable points to the first address in its block of memory.
- This is why arrays are indexed from 0.

```
Address of array first element: 0x7fffffffdc10
Value stored in a: 0x7fffffffdc10
First element in a: 1
Dereferencing a: 1
Second element in a: 2
Dereferencing a+1: 2
```

Does Every Pointer Point to Something?

- Remember that an uninitialized variable will have a “garbage” value left in it.
- An uninitialized pointer is probably still pointing to a valid memory address though.
- **NULL or null_ptr** are reserved pointer values that point to “nothing”.

Example 4

```
int main(int argc, char** argv){
    int* p = NULL;
    cout << "p points to " << p << endl;
    cout << "the value at p " << *p << endl;
    //segmentation fault
    return 0;
} //try to predict the output
```

Always Check for NULL

- Reading a NULL pointer is legal and encouraged:
  - if (ptr != NULL)…
- Writing to a NULL pointer will cause a run time error.

By-reference Using Pointers

```
//Double the value of a by reference parameter
void refFunc(int &X){
    X *= 2;
    return;
}
int main(int argc, char** argv){
    int a =2;
    refFunc(a);
    cout << a << endl;
    return 0;
} //try to predict the output
```

```
//Same thing but with pointers
void ptrFunc(int* X){
    *X *= 2;
    return;
}
int main(int argc, char** argv){
    int a =2;
    ptrFunc(&a);
    cout << a << endl;
    return 0;
} //try to predict the output
```

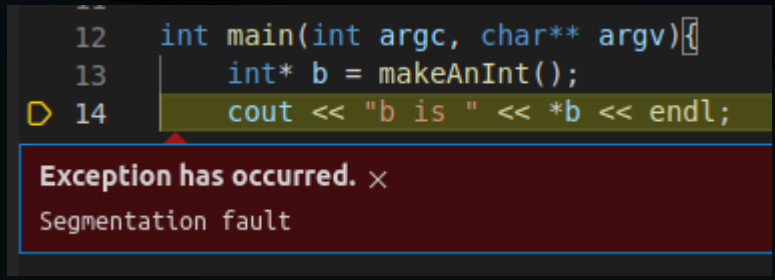
Returning Pointers from Functions

```
int* makeAnInt(){
    int a = 3;
    int* b = &a;
    return b;
}


int main(int argc, char** argv){
    int* b = makeAnInt();
    cout << "b is " << *b << endl;
} //This program causes an error!
```

Returning Pointers from Functions

- The function makeAnInt() creates a variable in memory and then returns the address of that variable.
- That variable goes out of scope when the function returns and can no longer be referenced.



Questions or Comments?