#3: Memory
January 18, 2019 Zilles, Fagen-Ulmschneider

Pointers and References
Often, we will have direct access to our object:

---
Cube s1; // A variable of type Cube
---

Occasionally, we have a reference or pointer to our data:

---
Cube & s1; // A reference variable of type Cube
Cube * s1; // A pointer that points to a Cube
---

Pointers
Unlike reference variables, which alias another variable’s memory, pointers are variables with their own memory. Pointers store the memory address of the contents they’re “pointing to”.

Three things to remember on pointers:

1. 
2. 
3. 

main.cpp

```cpp
int main() {
    cs225::Cube c;
    std::cout << "Address storing `c`:" << &c << std::endl;
    cs225::Cube *ptr = &c;
    std::cout << "Addr. storing ptr: " << &ptr << std::endl;
    std::cout << "Contents of ptr: " << ptr << std::endl;
    return 0;
}
```

example1.cpp

```
int main() {
    int a;
    int b = -3;
    int c = 12345;
    int *p = &b;
    return 0;
}
```

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffff00f0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff00e8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff00e0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff00d8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff00d0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

example2.cpp

```
int main() {
    cs225::Cube c;
    cs225::Cube *p = &c;
    return 0;
}
```

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffff00f0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff00e8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff00e0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff00d8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff00d0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Indirection Operators:

- `&v`
- `*v`
- `v->`
Stack Frames
All variables (including parameters to the function) that are part of a function are part of that function’s stack frame. A stack frame:

1.
2.

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffff00f0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff00e8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff00e0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff00d8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff00d0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Puzzle: What happens here?

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffff00f0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff00e8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff00e0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff00d8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff00d0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Heap Memory:
As programmers, we can use heap memory in cases where the lifecycle of the variable exceeds the lifecycle of the function.

1. The only way to create heap memory is with the use of the **new** keyword. Using **new** will:
   
   •
   
   •

2. The only way to free heap memory is with the use of the **delete** keyword. Using **delete** will:
   
   •
   
   •

3. Memory is never automatically reclaimed, even if it goes out of scope. Any memory lost, but not freed, is considered to be “leaked memory”.

CS 225: TTBD ➔ lab_intro due Sunday; MP1 released and due Jan. 28th