CS 161
Introduction to CS I
Lecture 15

• How does memory work in a C++ program?
About Me

• 6th year at OSU, got my Bachelor Degree in Spring, 2018
• Involved in CS 16X since Fall 2017
• Taught CS 161 last term
Week 6 Tips

• Lab 6 – posted
  • Revisit pass by reference
  • Practice on pass by
    • Implementing Hangman
  • Memory model
• Study session this week
  • Thursday 6-7pm at LINC 268
  • Worksheet 6 is posted on the website
Assignment 4: Text Surgeon

- Read in a line of text from the user, and perform analysis and manipulation of that string
- Provides practice with
  - String functions
  - 1-dimensional arrays
  - C-style strings
  - Dynamic memory allocation
- Design Document is due Feb. 16 – go for it!
Review: References and Pointers

• Declare variables:
  • Reference: \texttt{int\& z = n; \quad /* z is an alias to n */}
  • Pointer: \texttt{int* p = \&n; \quad /* p is the address of n */}

• Operators (perform actions):
  • \&: address-of
    • \texttt{p = \&n;}
    • \texttt{\&n = 5234; \quad /* not allowed! (what would it mean?) */}
  • *: dereference (value-of): access the value at memory address
    • \texttt{int\ g = *p; \quad /* read */}
    • \texttt{*p = 27; \quad /* write/change */}
References versus Pointers

• Do not confuse "reference" (a data type) with "pass by reference" (something that happens when you call a function)
• **Reference**: an alias to some variable (permanent)
  • `int& r = s;`
  • Can assign new values to `r` (which is `s`), but cannot make `r` be an alias to another variable later
  • Must be initialized when declared
• **Pointer**: stores the address of some variable
  • `int* p = &s;`
  • Can change what address `r` contains (where it points to) anytime
  • Can be declared, then initialized later
1. `int r = 17;`
2. `int s = -10;`
3. `int* q = NULL;`

\[\begin{array}{|c|c|c|c|}
\hline
Line & r & s & q & *q \\
\hline
3 & & & & \\
4 & & & & \\
5 & & & & \\
6 & & & & \\
7 & & & & \\
8 & & & & \\
\hline
\end{array}\]

\&r = Addr1
\&s = Addr2
1. int r = 17;
2. int s = -10;
3. int* q = NULL;
4. q = &r;

&r = Addr1
&s = Addr2

<table>
<thead>
<tr>
<th>Line</th>
<th>r</th>
<th>s</th>
<th>q</th>
<th>*q</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>17</td>
<td>-10</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Pointer activity

1. `int r = 17;`
2. `int s = -10;`
3. `int* q = NULL;`
4. `q = &r;`
5. `r = -5;`

```c
int r = 17;
int s = -10;
int* q = NULL;
q = &r;
r = -5;
```

```c
&r = Addr1
&s = Addr2
```

<table>
<thead>
<tr>
<th>Line</th>
<th>r</th>
<th>s</th>
<th>q</th>
<th>*q</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>17</td>
<td>-10</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>-10</td>
<td>Addr1</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>Addr1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>Addr1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>Addr1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>Addr1</td>
<td></td>
</tr>
</tbody>
</table>
1. int r = 17;
2. int s = -10;
3. int* q = NULL;
4. q = &r;
5. r = -5;
6. *q = 42;

&r = Addr1
&s = Addr2
1. `int r = 17;`
2. `int s = -10;`
3. `int* q = NULL;`
4. `q = &r;`
5. `r = -5;`
6. `*q = 42;`
7. `q = &s;`

```
&r = Addr1
&s = Addr2
```

<table>
<thead>
<tr>
<th>Line</th>
<th>r</th>
<th>s</th>
<th>q</th>
<th>*q</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>17</td>
<td>-10</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>-10</td>
<td>Addr1</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>-5</td>
<td>-10</td>
<td>Addr1</td>
<td>-5</td>
</tr>
<tr>
<td>6</td>
<td>42</td>
<td>-10</td>
<td>Addr1</td>
<td>42</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. int r = 17;
2. int s = -10;
3. int* q = NULL;
4. q = &r;
5. r = -5;
6. *q = 42;
7. q = &s;
8. s++;

&r = Addr1
&s = Addr2

<table>
<thead>
<tr>
<th>Line</th>
<th>r</th>
<th>s</th>
<th>q</th>
<th>*q</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>17</td>
<td>-10</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>-10</td>
<td>Addr1</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>-5</td>
<td>-10</td>
<td>Addr1</td>
<td>-5</td>
</tr>
<tr>
<td>6</td>
<td>42</td>
<td>-10</td>
<td>Addr1</td>
<td>42</td>
</tr>
<tr>
<td>7</td>
<td>42</td>
<td>-10</td>
<td>Addr2</td>
<td>-10</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. int r = 17;
2. int s = -10;
3. int* q = NULL;
4. q = &r;
5. r = -5;
6. *q = 42;
7. q = &s;
8. s++;

&r = Addr1
&s = Addr2

<table>
<thead>
<tr>
<th>Line</th>
<th>r</th>
<th>s</th>
<th>q</th>
<th>*q</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>17</td>
<td>-10</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>-10</td>
<td>Addr1</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>-5</td>
<td>-10</td>
<td>Addr1</td>
<td>-5</td>
</tr>
<tr>
<td>6</td>
<td>42</td>
<td>-10</td>
<td>Addr1</td>
<td>42</td>
</tr>
<tr>
<td>7</td>
<td>42</td>
<td>-10</td>
<td>Addr2</td>
<td>-10</td>
</tr>
<tr>
<td>8</td>
<td>42</td>
<td>-9</td>
<td>Addr2</td>
<td>-9</td>
</tr>
</tbody>
</table>
Passing pointers into functions

- \(\text{int } v = 3;\)  \(\text{int* } p = &v;\)
- \(\text{void fn1(int } w);\)  \(\text{void pfn1(int* } q);\)
- \(\text{void fn2(int& } w);\)  \(\text{void pfn2(int* & q);}\)

- **Pass by value:** make a copy:
  - Same for pointers: make a copy of the address inside the pointer variable; changes to \(q\) do not change \(p\)
- **Pass by reference:** pass the address of the variable:
  - Same for pointers: pass the address of the pointer variable; changes to \(q\) DO change \(p\)
Challenge questions

• What if you made a pointer (p2) that points to a pointer (p1) that points to an int (x)?
  • What would the picture look like?
  • Write the code for this picture.

• Can you make this same picture for references?
  • What if you had two references, r1 and r2?
  
  ```
  int var = 50;
  int &r1 = var;
  int &r2 = var;  // You cannot say: int &&r2 = var;
  ```

  var, r1, r2
Memory Model/Layout
What we have seen so far: Variables vs. Pointers

• Value
  – Values stored directly
  – Copy of value is passed
    \[
    \text{int } i, j=2; \\
i=j;
    \]

• Pointer
  – Address to variable is stored
  – Copy of address is passed
    \[
    \text{int } *i = \text{NULL}, j=2; \\
i=&j;
    \]
Stack – Static Memory

• Stack
  • Variables known in advance (global/local variables, constants), always allocated at compile time
  • Functions have their own stack frame
  • When a function ends, the stack frame collapses and cleans up the memory for you
What if we don’t have the j?

- We need to **create the address space**
- How do we do this?
  - **new** type;
- For example:
  ```
  int *i = NULL;
  i = new int; //new returns an address
  *i = 10;
  ```
Heap – Dynamic Memory

- Heap
  - Variables defined at runtime (use `new` keyword), do not need to be known in advance
  - Variables declared dynamically in a function do not disappear when the function ends as they are on the heap and not the function stack
  - Need to free dynamic memory when done with it, otherwise memory leaks
Static vs. Dynamic

• Static
  • Assign address of variable
    ```
    int *i=NULL, j=2;
    i=&j;
    ```

• Dynamic
  • Create memory
  • Assign memory to pointer
    ```
    int *i=new int;
    *i=2;
    ```
How to avoid Memory Leaks?
A: Deleting items from the heap

• Delete operator: delete
• (delete does not clear the memory contents, just lets it be reused)

For example:
```cpp
int main () {
    int *i = NULL;
    i = new int;
    *i = 2;
    delete i;
    i = NULL;  // set the pointer back to NULL
    return 0;
}
```
Segmentation Fault (aka segfault)

- Something that causes programs to crash
- Often caused by program trying to read or write an illegal memory location

For example, what’s wrong with this:
```c
int main () {
    int *i = NULL;
    i = new int; //if forget this, segfault
    *i = 2;
    delete i;
    i = NULL; // set the pointer to NULL
    return 0;
}
```
Memory allocation tips

• new can fail – throws exception
• after delete, set your ptr to NULL (explicitly)
• you can delete a NULL ptr with no adverse effects
• Gotchas:
  • forget to delete: memory leak
  • forget to set to NULL: dangling pointers
• tool: valgrind
What vocabulary did we learn today?

- Static memory
- Dynamic memory
- Stack
- Heap
- Segmentation fault
- Dynamic memory operators: new and delete
- Memory leak
- Dangling pointer
What ideas and skills did we learn today?

- Memory model: where the stack and the heap are
- How to dynamically allocate memory
- How to delete dynamic memory
- How to check for memory leaks (valgrind)
Week 6 begins!

- Attend lab (laptop required)
- Read Rao Lesson 8 (pp. 187-204)
- Start design for Assignment 4 (due Sunday, Feb. 16)