Man, I suck at this game. Can you give me a few pointers?

I hate you.

0x3A28213A
0x6339392C
0x7363682E.
Warm-Up

Function: increments_by
Description: increments num1 by the value num2
Input: num1, num2
Return type: void

<table>
<thead>
<tr>
<th></th>
<th>Pass By Value</th>
<th>Pass By Reference</th>
<th>Pass By Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter Listing</td>
<td>int num1, int num2</td>
<td>int &amp; num1, int &amp; num2</td>
<td>int * num1, int * num2</td>
</tr>
<tr>
<td>What is actually being passed?</td>
<td>copy of a value</td>
<td>value and memory address</td>
<td>memory address</td>
</tr>
<tr>
<td>Does the function body need to change? If so, how?</td>
<td>No</td>
<td>No</td>
<td>Yes, add dereference operator to alter values</td>
</tr>
<tr>
<td>Function Call</td>
<td>increments_by(num1, num2)</td>
<td></td>
<td>increments_by((num1, num2))</td>
</tr>
<tr>
<td>How will things change from where the function was called?</td>
<td>They stay the same</td>
<td>They change num1 = num1 + num2</td>
<td></td>
</tr>
</tbody>
</table>
More Pointers

• Declaring pointers

  int *i; //This will hold an int memory address
  int j; //This holds an int
  i = &j; //set the address that i holds to the address of j
  j = 2; //sets the value of j to 2, *i is 2
  (*i)++; //increments the value i points to, j is 3
```cpp
#include <iostream>

using namespace std;

int main() {
    int *i;
    int j;
    cout << endl;
    cout << "int *i address: " << &i << endl;
    cout << "int j address: " << &j << endl;

    i = &j;
    j = 2;
    cout << endl;
    cout << "value stored at i: " << i << endl;
    cout << "value stored at j: " << j << endl;

    cout << "value i points to: " << *i << endl;
    cout << "int *i address: " << &i << endl;
    j++;
    cout << endl;
    cout << "Increment j" << endl;
    cout << "value stored at i: " << i << endl;
}
```

cout << "value i points to: " << *i << endl;
cout << "int *i address: " << &i << endl;
j++;
cout << endl;
cout << "Increment j" << endl;
cout << "value stored at i: " << i << endl;
cout << "value stored at j: " << j << endl;
cout << "value i points to: " << *i << endl;
cout << "int *i address: " << &i << endl;
cout << endl;
(*i)++;
cout << "Increment the value at i" << endl;
cout << "value stored at i: " << i << endl;
cout << "value stored at j: " << j << endl;
cout << "value i points to: " << *i << endl;
cout << "int *i address: " << &i << endl;
cout << endl;
return 0;
}
flip3 ~/teaching/cs161/lectures/week_6 156% a.out

int *i address: 0x7fff64fe3078
int j address: 0x7fff64fe3074

value stored at i: 0x7fff64fe3074
value stored at j: 2
value i points to: 2
int *i address: 0x7fff64fe3078

Increment j
value stored at i: 0x7fff64fe3074
value stored at j: 3
value i points to: 3
int *i address: 0x7fff64fe3078

Increment the value at i
value stored at i: 0x7fff64fe3074
value stored at j: 4
value i points to: 4
int *i address: 0x7fff64fe3078

flip3 ~/teaching/cs161/lectures/week_6 157%
What if we don’t have an address to point to?

- We make one with the `new` keyword (dynamically allocate)
  
  ```c
  int *p;
p = new int; //new returns an address
*p = 10;
  ```
```cpp
#include <iostream>

using namespace std;

int main() {
    int *p;

    cout << "address of int *p: " << &p << endl;

    p = new int;

    cout << "address p points to: " << p << endl;
    cout << "value at address p points to: " << *p << endl;
    *p = 10;
    cout << "Gave *p a value" << endl;
    cout << "value at address p points to: " << *p << endl;
    delete p;

    cout << "addr: " << &p << endl;
    cout << "point to addr: " << p << endl;
    cout << "point to val: " << *p << endl;

    return 0;
}
```

Different Types of Memory

• CPU: central processing unit, “brain” of the computer system

• Main memory
  • where current programs are executed
  • CPU has direct and quick address
  • Volatile: contents are lost when the power goes out

• Secondary Memory
  • Nonvolatile, long term storage
  • Ex. Files, hard drive, USB, etc.
How Main Memory is Structured

• Stack
  • Variables defined at compile time go on the stack (global variables, constants)
  • Functions have their own stack frame
  • When a function ends, the stack frame collapses and cleans up the memory for you -> sometimes referred to as automatic variables

• Heap
  • Variables defined at runtime (new keyword)
  • Variables declared dynamically in a function do not disappear when the function ends as they are on the heap and not the function stack
  • Can run out of heap space
  • Need to free dynamic memory when done with it, otherwise memory leaks
Static vs Dynamic

- Static
  
  \[
  \text{int } *i, j=2;
  \]
  
  \[
  i = &j;
  \]

- Dynamic
  
  \[
  \text{int } *i = \text{NULL};
  \]
  
  \[
  i = \text{new int};
  \]
  
  \[
  *i = 2;
  \]
Fixing Memory Leaks

• How to tell you may have a memory leak
  • You used the new keyword
  • You never used the delete keyword
  • You run valgrind
    • Compile and produce an executable for your program
    • Run valgrind with your executable (valgrind executable_name)

• How to fix memory leaks
  • Delete dynamic memory when you are done with it
    int *i = new int;
    delete i;
Demo

```bash
flip3 ~/teaching/cs161/lectures/week_6 160% a.out
address of int *p: 0x7ffe0235b2e8
address p points to: 0x25a9010
value at address p points to: 0
Gave *p a value
value at address p points to: 10
addr: 0x7ffe0235b2e8
point to addr: 0x25a9010
point to val: 0
flip3 ~/teaching/cs161/lectures/week_6 161%
```
==22471== Invalid read of size 4
==22471== at 0x400AEE: main (in /nfs/stak/users/ernstsh/teaching/cs161/lectures/week_6/a.out)
==22471== Address 0x5a19040 is 0 bytes inside a block of size 4 free'd
==22471== at 0x4C2B18D: operator delete(void*) (vg_replace_malloc.c:576)
==22471== by 0x400A93: main (in /nfs/stak/users/ernstsh/teaching/cs161/lectures/week_6/a.out)
==22471== Block was alloc'd at
==22471== at 0x4C2A203: operator new(unsigned long) (vg_replace_malloc.c:334)
==22471== by 0x4009DA: main (in /nfs/stak/users/ernstsh/teaching/cs161/lectures/week_6/a.out)
==22471== point to val: 10
==22471==
==22471== HEAP SUMMARY:
==22471==     in use at exit: 0 bytes in 0 blocks
==22471==     total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==22471==
==22471== All heap blocks were freed -- no leaks are possible
==22471==
==22471== For counts of detected and suppressed errors, rerun with: -v
==22471== Use --track-origins=yes to see where uninitialised values come from
==22471== ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 0 from 0)
flip3 ~/teaching/cs161/lectures/week_6 1639
Feedback

https://tinyurl.com/y7c79hap