Additional topics on pointers

William Hendrix

Lecture 13

Outline

- Pointer arithmetic
- Shallow vs. deep copying
- Useful functions
- Command-line arguments

Review: pointers

- Variables that store the memory location for other variables
- Syntax: `int* ptr1;`
  - Variable type should precede `*`
  - `char**` is a pointer to a `char*`
- `void*`: "Typeless" pointer
  - Can be assigned and cast, but not dereferenced
  - Useful for functions that return a pointer of unknown type or if you want to "obscure" the type
- Pointer size is the same for all types
- Primary operations
  - Address operator (`&`): returns a pointer to a variable
    - Used to initialize pointers
  - Dereference operator (`*`): returns a variable from a pointer

Pointer arithmetic

- **WARNING:** material on this slide is not recommended
- Pointers can be changed through arithmetic operations
- Addition
  - Adding an integer to a pointer changes the memory location by `value_added * sizeof(pointer_type)`
  - `array[n]` is the same as `*(array + n)`
    - Why arrays always start at 0: `array[0] = *array`
- Subtraction
  - Subtracting two pointers of the same type results in the distance in bytes from the second to the first divided by the size of the type
  - `second[first - second] = *(second + first - second) = *first = first[0]`
    - Pointers are always a multiple of their size
- Comparisons are also valid
  - Never compare two `char*` or `char[]`

Pointers can make your life difficult

- Pointer arithmetic (and array offsets) allow you to change arbitrary values in memory
  - If the pointer location is wrong, you could:
    - crash the program immediately
    - modify another variable or array in your program
    - cause the program to crash when you free an array

- If the value of a variable is incorrect, that variable is wrong and your program may crash
- If the value of a pointer is incorrect, you may modify the wrong value (*very* hard to debug)

Pointer arithmetic example

- What does the following code do to array `arr`?

```
int* end = arr + len;
while (arr < end)
   x += *arr++;
arr -= len;
```

- How does it differ from the following?

```
for (int i = 0; i < len; i++)
   x += arr[i];
```

Historical note: the Morris Worm

- Abused a security flaw in the finger program in UNIX
  - finger allowed user to write values into memory, but didn't check for validity
- Worm used a well-chosen offset to overwrite the instructions for the finger program in memory
- Took control of machine to infect other vulnerable UNIX machines
- Crashed the Internet in 1988

- **Moral of the story**
  - Always check array bounds
  - Always check user input

Shallow vs. deep copying

- Dynamic arrays can be Lvalues (on LHS of assignment)
- However, assignment only copies the pointer into the new variable
  - Still only one array in memory, just two “names”
- Modifying either array will cause both to change
  - Shallow copying
- Example

```
int* arr1 = new int[10];
for (int i = 0; i < 10; i++)
  arr1[i] = i + 1;
int* arr2 = arr1;
arr2[5] = -1;  //Also changes arr1[5]
```

- Deep copying
  - Need to allocate a new array, then copy
  - Allocate with `new` or `malloc`
  - Copy values with loop or function

Pointer/array functions

- `new` and `malloc` allocate memory on the *heap*
  - Separate from the *stack*
  - Pointers do not have scope; “always” reference parameters
  - `new` initializes memory to be zero
- `memcpy`: copies data from one pointer to another
  - Syntax: `memcpy(dest_ptr, src_ptr, num_bytes);`
    - `#include <string.h>`
  - Will cause an error if source and destination overlap
  - `memmove`: same as `memcpy` but allows overlap
    - E.g., inserting a character into a string (shift everything to the right)
    - Somewhat slower
  - `strncpy`: specific to `char*`
    - Won’t copy past `‘\0’` in source string
    - Doesn’t append `‘\0’` unless it reaches end of source string
- `memset`: initializes a memory region
  - Syntax: `memset(ptr, byte_value, num_bytes);`
    - `byte_value` is usually `0` or `0xff`

Command-line arguments

- Alternate form for `main`:

  ```
  int main(int argc, char** argv)
  ```

  - Standard for C programs
- Additional arguments are used for command-line arguments to your program
  - E.g., `C:> mycopy file1.txt file2.txt`
  - `argv`: array of `char` arrays corresponding to arguments
    - `argv[0]` is always the name of the executable
  - `argc`: number of arguments (+1 for program name)
- Example

```
if (argc < 3)
{
  cout << "Usage:  " << argv[0]
       << " [source file] [destination file]\n";
  return 1;
}
ifstream src(argv[1]);
ofstream dest(argv[2]);
//...
```

Tonight

Lab 3 is due Monday at 11:59pm