Administrivia (1)

Reference system: `gcc -Wall -g -std=c99` on attu / workstations / home FC15 64-bit VM

- Linux compiler issues more warnings than some others (OS X)

Lecture exercises

- Grading scale: 2 bits per exercise
  - 0 = (late) or (major incorrectness)
  - 1 = (major style errors) or (minor incorrectness) or (compilation warnings)
  - 2 = minor style errors [most people will get this, I hope]
  - 3 = perfection
Administrivia 1.5

ex0 results

- 3 - 4%
- 2 - 38%
- 1 - 42%
- 0 - 0%
- no submission - 15%
- rounding error - 1%

- Feedback is in your catalyst dropbox
Lecture exercises continued:

- we are posting our solutions
  - my code is a little too overcommented, so be forewarned
    - (e.g. some comments explain the code in ways that make sense for instruction, but not for production code)
- we will get more explicit about good style as the quarter progresses
  - and, we will give you access to a nice “lint” style checker
  - don’t rely on it for perfection
- we will take your best N-4 exercises for your grade
Administrivia (3)

Section exercises

- Grading scale: 1 (satisfactory) / 0 (not satisfactory, late)
- Finished version due by 11 pm Friday night after section
  ‣ No late submissions after that
- Intent is main work done in sections with some polishing / detail work needed afterwards
  ‣ Let us know if the difficulty / time needed gets out of hand
Administrivia (4)

Office hours. Leading possibilities are:
- Mon. right after class or maybe an hour later
- Tue. mornings
- Wed. right after class or 3:30 pm
- Thur. right after section or an hour later or 4 pm
- Fri. right after class or an hour later or later in the afternoon

So what will it be?
Today’s goals:

- functions
- arrays
- refresher on C’s memory model
  ‣ address spaces
  ‣ the stack
  ‣ brief reminder of pointers (more next time)
Defining a function

```
returnType name(type name, ..., type name) {
    statements;
}
```

// sum integers from 1 to max
int sumTo(int max) {
    int i, sum = 0;
    for (i=1; i<=max; i++) {
        sum += i;
    }
    return sum;
}
Problem: ordering

You shouldn’t call a function that hasn’t been declared yet

```c
#include <stdio.h>

int main(int argc, char **argv) {
    printf("sumTo(5) is: %d\n", sumTo(5));
    return 0;
}

// sum integers from 1 to max
int sumTo(int max) {
    int i, sum = 0;
    for (i=1; i<=max; i++) {
        sum += i;
    }
    return sum;
}
```
Problem: ordering

Solution 1: reverse order of definition

#include <stdio.h>

// sum integers from 1 to max
int sumTo(int max) {
  int i, sum = 0;
  for (i=1; i<=max; i++) {
    sum += i;
  }
  return sum;
}

int main(int argc, char **argv) {
  printf("sumTo(5) is: %d\n", sumTo(5));
  return 0;
}
Problem: ordering

Solution 2: provide a declaration of the function

- teaches the compiler the argument and return types of the function

```c
#include <stdio.h>

// this function prototype is a declaration of sumTo
int sumTo(int);

int main(int argc, char **argv) {
    printf("sumTo(5) is: %d\n", sumTo(5));
    return 0;
}

// sum integers from 1 to max
int sumTo(int max) {
    int i, sum = 0;
    for (i=1; i<=max; i++) {
      sum += i;
    }
    return sum;
}
```

sum_declared.c
Declaration vs Definition

C and C++ make a careful distinction between these

**Definition:** The thing itself

- Code for function, or a global variable definition that creates storage
- Must be **exactly one** actual definition of each thing (no dups)

**Declaration:** Description of a thing in files that wish to use it

- Function prototype or external variable declaration
- Should be repeated in each source file that uses it
  - Often written in header files and incorporated via `#include`
  - `#include` declaration in file that defines it to check consistency
Arrays

`type name[size];`

example allocates 100 ints’ worth of memory
- initially, each array element contains garbage data

an array does not know its own size
- `sizeof(scores)` is not reliable; only works in some situations
- recent versions of C allow the array size to be an expression

```
int n=100;
int scores[n];  // OK in C99
```
Initializing and using arrays

type name[size] = {value, value, ..., value};
- allocates and array and fills it with supplied values
- if fewer values are given than the array size, fills rest with 0

name[index] = expression;
- sets the value of an array element

```c
int primes[6] = {2, 3, 5, 6, 11, 13};
primes[3] = 7;
primes[100] = 0;   // smash!

int allZeroes[1000] = {0};
```

// 1000 zeroes
Multi-dimensional arrays

type name[rows][columns] = {{values}, ..., {values}};

- allocates a 2D array and fills it with predefined values

// a 2 row, 3 column array of doubles
double grid[2][3];

// a 3 row, 5 column array of ints
int matrix[3][5] = {
    {0, 1, 2, 3, 4},
    {0, 2, 4, 6, 8},
    {1, 3, 5, 7, 9}
};

grid[0][2] = (double) matrix[2][4];  // which val?
Arrays as parameters

It’s tricky to use arrays as parameters

- arrays are effectively passed by reference (not copied)
- arrays do not know their own size

```
int sumAll(int a[]); // prototype declaration

int main(int argc, char **argv) {
    int numbers[5] = {3, 4, 1, 7, 4};
    int sum = sumAll(numbers);
    return 0;
}

int sumAll(int a[]) {
    int i, sum = 0;
    for (i = 0; i < ...???
```
Arrays as parameters

Solution 1: declare the array size in the function
- problem: code isn’t very flexible

```c
int sumAll(int a[5]); // prototype declaration

int main(int argc, char **argv) {
    int numbers[5] = {3, 4, 1, 7, 4};
    int sum = sumAll(numbers);
    return 0;
}

int sumAll(int a[5]) {
    int i, sum = 0;
    for (i = 0; i < 5; i++) {
        sum += a[i];
    }
    return sum;
}
```
Arrays as parameters

Solution 2: pass the size as a parameter

```c
int sumAll(int a[], int size);

int main(int argc, char **argv) {
    int numbers[5] = {3, 4, 1, 7, 4};
    int sum = sumAll(numbers, 5);
    printf("sum is: %d\n", sum);
    return 0;
}

int sumAll(int a[], int size) {
    int i, sum = 0;
    for (i = 0; i <= size; i++) {  // CAN YOU SPOT THE BUG?
        sum += a[i];
    }
    return sum;
}
```

arraysum.c
Returning an array

Local variables, including arrays, are stack allocated

- they disappear when a function returns
- therefore, local arrays can’t be safely returned from functions

```c
int[] copyarray(int src[], int size) {
    int i, dst[size];   // OK in C99
    for (i = 0; i < size; i++) {
        dst[i] = src[i];
    }
    return dst;   // NO -- bug
}
```

`buggy_copyarray.c`
Solution: an output parameter

Create the “returned” array in the caller

- pass it as an **output parameter** to copyarray

- works because arrays are effectively passed by reference

```c
void copyarray(int src[], int dst[], int size) {
    int i;
    for (i = 0; i < size; i++) {
        dst[i] = src[i];
    }
}
copyarray.c
```
OS and processes

The OS lets you run multiple applications at once

- an application runs within an OS “process”
- the OS timeslices each CPU between runnable processes
  - happens very fast; ~100 times per second!
Processes and virtual memory

OS gives each process the illusion of its own, private memory

- this is called the process’ *address space*
- contains the process’ virtual memory, visible only to it
- $2^{32}$ bytes on 32 bit host
- $2^{64}$ bytes on 64 bit host

(process’ address space)

contains code, data, libraries, stack, etc.

0x00000000

0xFFFFFFFF
Loading

When the OS loads a program, it:

- creates an address space
- inspects the executable file to see what’s in it
- (lazily) copies regions of the file into the right place in the address space
- does any final linking, relocation, or other needed preparation
The stack

Used to store data associated with function calls

- when you call a function, compiler-inserted code will allocate a stack frame to store:
  ‣ the function call arguments
  ‣ the address to return to
  ‣ local variables used by the function
  ‣ a few other pieces of bookkeeping

```c
int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    return x;
}
```

<table>
<thead>
<tr>
<th>offset</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>p2</td>
</tr>
<tr>
<td>20</td>
<td>p1</td>
</tr>
<tr>
<td>16</td>
<td>return address</td>
</tr>
<tr>
<td>12</td>
<td>a[2]</td>
</tr>
<tr>
<td>8</td>
<td>a[1]</td>
</tr>
<tr>
<td>4</td>
<td>a[0]</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
</tr>
</tbody>
</table>

a stack frame
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```

OS kernel [protected]

- stack
- heap (malloc/free)
- read/write segment
- globals
- read-only segment
  - (main, f, g)
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```

OS kernel [protected]

- stack
- main
  - argc, argv, n1
- heap (malloc/free)
- read/write segment
  - globals
- read-only segment
  - (main, f, g)
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```

OS kernel [protected]

<table>
<thead>
<tr>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
</tr>
<tr>
<td>argc, argv, n1</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>p1, p2, x, a</td>
</tr>
<tr>
<td>heap (malloc/free)</td>
</tr>
<tr>
<td>read/write segment</td>
</tr>
<tr>
<td>globals</td>
</tr>
<tr>
<td>read-only segment</td>
</tr>
<tr>
<td>(main, f, g)</td>
</tr>
</tbody>
</table>
The stack in action

```
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```

OS kernel [protected]

- stack
- main
  - argc, argv, n1
- f
  - p1, p2, x, a
- heap (malloc/free)
- read/write segment
  - globals
- read-only segment
  - (main, f, g)
The stack in action

int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...  
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```

OS kernel [protected]

- stack
- main
  - argc, argv, n1
- f
  - p1, p2, x, a
- g
  - param
- heap (malloc/free)
- read/write segment
  - globals
- read-only segment
  - (main, f, g)
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...  
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```

OS kernel [protected]
- stack
- main
  - argc, argv, n1
- f
  - p1, p2, x, a
- g
  - param
- heap (malloc/free)
- read/write segment
  - globals
- read-only segment
  - (main, f, g)
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```

- **OS kernel [protected]**
  - stack
  - **main**
    - argc, argv, n1
  - **f**
    - p1, p2, x, a
  - **g**
    - param
  - heap (*malloc/free*)
  - read/write segment
    - **globals**
  - read-only segment
    - (main, f, g)
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}
int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}
int g(int param) {
    return param * 2;
}
```
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```
The stack in action

```
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```

OS kernel [protected]
- stack
- main
  - argc, argv, n1
- heap (malloc/free)
- read/write segment 
  - globals
- read-only segment 
  - (main, f, g)
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```
The stack in action

```
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```

OS kernel [protected]
- stack
- main
  argc, argv, n1
- g
  param
- heap (malloc/free)
- read/write segment
  globals
- read-only segment
  (main, f, g)
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```

OS kernel [protected]
- stack
- main
  argc, argv, n1
- heap (malloc/free)
- read/write segment globals
- read-only segment (main, f, g)
The stack in action

```c
int main(int argc, char **argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```
Addresses and 

&foo produces the virtual address of foo

```c
#include <stdio.h>

int foo(int x) {
    return x + 1;
}

int main(int argc, char **argv) {
    int x, y;
    int a[2];
    printf("x is at \%p\n", &x);
    printf("y is at \%p\n", &y);
    printf("a[0] is at \%p\n", &a[0]);
    printf("a[1] is at \%p\n", &a[1]);
    printf("foo is at \%p\n", &foo);
    printf("main is at \%p\n", &main);
    return 0;
}
```
Pointers

type *name;       // declare a pointer

Type *name = address;  // declare + initialize a pointer

A pointer is a variable that contains a memory address

- It points to somewhere in the process’ virtual address space

```c
int main(int argc, char **argv) {
  int x = 42;
  int *p;       // p is a pointer to an integer
  p = &x;       // p now stores the address of x
  printf("x is %d\n", x);
  printf("&x is %p\n", &x);
  printf("p is %p\n", p);

  return 0;
}
```

A stylistic choice

C gives you flexibility in how you declare pointers

- one way can lead to visual trouble when declaring multiple pointers on a single line
- the other way is what I prefer

```c
int* p1, p2;  // bug?; equivalent to int *p1; int p2;
int* p1, *p2; // correct
```

or

```c
int *p1; // correct, i prefer
int *p2;
```
Dereferencing pointers

*pointer                // dereference a pointer
*pointer = value;  // dereference / assign

dereference: access the memory referred to by a pointer

#include <stdio.h>

int main(int argc, char **argv) {
  int x = 42;
  int *p;       // p is a pointer to an integer
  p = &x;       // p now stores the address of x
  printf("x  is %d
", x);
  *p = 99;
  printf("x  is %d
", x);
  return 0;
}

deref.c
Self exercise #1

Write a function that:

- accepts an array of 32-bit unsigned integers, and a length
- reverses the elements of the array in place
- returns void (nothing)
Self exercise #2

Write a function that:

- accepts a function pointer and an integer as an argument
- invokes the pointed-to function
  ‣ with the integer as its argument
Write a function that:

- accepts a string as a parameter
- returns
  ‣ the first whitespace-separated word in the string (as a newly allocated string)
  ‣ and, the size of that word
See you on Monday!