CSE 333
Lecture 2 - arrays, memory, pointers

Administrivia

ex0, hw0 were due 15 minutes ago!
- let me know if you had any logistical issues with either

ex1 is out today, due on Friday
- hw1 will be handed out on Friday, due two weeks later
Today’s agenda

More C details
- functions
- arrays
- refresher on C’s memory model
  - address spaces
  - the stack
  - brief reminder of pointers

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/
Why?

```c
#include <stdio.h>

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

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

sum_badorder.c

Problem: ordering

Solution 1: find an ordering that respects the restriction

```c
#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;
}
```

sum_betterorder.c

Of course, this isn’t always possible.
Problem: ordering

Solution 2:

- Separate notions of declaration and definition
- Place declaration before use

```c
#include <stdio.h>

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

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

// This is the definition of sumTo
int sumTo(int max) {
  int i, sum = 0;
  for (i=1; i<=max; i++) {
    sum += i;
  }
  return sum;
}
```

Arrays

type name[size];

Example:

- allocates 100 ints' worth of memory
  - initially, each array element contains garbage data
  - associates the name scores with that memory

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

```c
int[] vecAdd(int[] A, int[] B, int n) {
  int result[n]; // OK in C99
  ... 
} 
```
Array initialization

\textit{type name[size]} = \{\textit{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

\textit{name[index]} = \textit{expression};

- sets the value of an array element

\begin{verbatim}
int primes[6] = \{2, 3, 5, 6, 11, 13\};
primes[3] = 7;
primes[100] = 0; // smash!
\end{verbatim}

\begin{verbatim}
// 1000 zeroes
int allZeroes[1000] = \{0\};
\end{verbatim}

Multi-dimensional arrays

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

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

\begin{verbatim}
// 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\}
};
\end{verbatim}
Arrays as parameters

It’s tricky to use arrays as parameters

- **Array names** are passed by value
  - which means that **array contents** are always passed by reference

- The language doesn’t provide any way to determine the length of an array (you have to write code if you want that)

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

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

```c
int sumAll(int a[]) {
    int i, sum = 0;
    for (i = 0; i < ...??? // there isn't anything you can write here to mean "a's length"
    }
}
```

Arrays as parameters

Solution 1: declare the array size in the function

- problem: this isn’t really a solution at all!
  - but, what does it do?

```c
int sumAll(int a[5]);
```

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

```c
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++) {
        sum += a[i];
    }
    return sum;
}
```

Pop quiz 1:
Can you spot the bug in this code?

Pop quiz 2:
What do you think happens when you run it?

Religious battle 1:
Which is better, C arrays or Java arrays?

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! -- buggy
}
```

But I thought C always passes & returns by value?
Stopgap Solution: an output parameter

Create the “returned” array in the caller
- pass it as an output parameter to `copyarray`
- we’ll see a better way later in the course

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

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!

```
process 1       process 2         ⋯         process N
---------------------------------------------------------------
operating system
```
Program memory: 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

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 allocate 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

```
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

```
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;
}
```
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>
</table>

<table>
<thead>
<tr>
<th>main</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc, argv, n1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1, p2, x, a</td>
</tr>
</tbody>
</table>

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]

<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>
</tbody>
</table>

heap (malloc/free)

read/write segment
globals

read-only segment
(main, f, g)
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

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]

- **main**
  - argc, argv, n1
- **g**
  - param
  - 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;
}
```
Addresses and the & operator

&foo produces “the 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;
}
```

```
$ ./addresses
x is at 0x7fffff4259338
y is at 0x7fffff425933c
a[0] is at 0x7fffff4259330
a[1] is at 0x7fffff4259334
foo is at 0x4004f4
main is at 0x400503
```

Pointers

```c
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

```c
int* p1;     // these three are all basically the same
int * p2;
int *p3;

int *p4, *p5; // these two are basically the same
int* p6, *p7;

int* p8, p9; // bug?; equivalent to int *p8; int p9;
```

Dereferencing pointers

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

dereference: access the memory referred to by a pointer

```c
#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

deref.c
    printf("x is %d\n", x);
    *p = 99;
    printf("x is %d\n", x);
    return 0;
}
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
Self exercise #3

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 Friday!