COSC 301: Monday, 3 September 2012

Note: these are all the remaining notes for the "learn C" portion of class. There are some additional examples that are included below, and other topics that we will get to over the course of week 2 in class.

Aggregate data types: `struct`

C has no classes or objects; instead we have *functions* and *structs* for organizing code and data.

A `struct` is just an aggregate data type:

```
struct record
{
    int id;
    char name[128];
    double gpa;
};
```

Use *dot operator* to access elements in a `struct`; think of `struct` elements as public members of a class. (That's actually *exactly* how they are in C++.)

```
struct record r1;
r1.id = 42;
r1.name = "Dr. Spaceman";
r1.gpa = 2.7;

// another way to initialize
struct record r2 = { 42, "Dr. Spaceman", 2.7 };
```

Arrays of structs

```
struct record reclist[100];
reclist[0].id = 42;
reclist[0].name = "Dr. Spaceman";
reclist[0].gpa = 2.6;
```

What's stored in the rest of the array?

Pointers

C gives you direct access to memory and to addresses in memory

- Data types known as pointers can point to a particular typed location in memory
- `NULL` is a special value which means that a pointer does not point to anything

Pointer is declared with an asterisk:

```
int *x;  // x is a pointer to an int
int y = 0;
x = &y;  // & is "address-of" operator:
         //   yields the memory address of some variable
```

An asterisk is also used to *dereference* (follow) a pointer:

```
int x;           // value in x?
int *y;          // value in y?
y = &x;          // what's this doing?

                 // %p format makes new output of a pointer
printf("The address of x is %p\n", y);

y = 42;         // dereference and assignment
                 // what's the value of x?

y = 42;          // what's the meaning?

int **z = &y;    // what would a memory diagram look like now?
int ***w = &z;   // and now?

// r points to a record in our array declared previously
struct record *r = &reclist[14];
(*r).id = 14; // must dereference pointer, then index the field
r->id = 15;   // shorthand for the above
r = NULL;     // does this affect the array?

struct record *reclist2[100]; // what does this declare?
struct record **reclist[10];  // or this?

char *pc;
pc = 'A';   // any problem here?
```

Functions, function calls, C standard library

Many similarities to Java. Key difference: the formal parameter value receives a *copy* of the actual parameter value.

- Parameters are passed by value in C and Java
- But most data types in Java are references (implicit pointers)
- So, if you make a change to an object that's passed to a Java method, the object will show that change *after* the method call
- If you want to pass a lot of data, or if you want the function to be able to change the data, *pass a pointer* to the data instead
- Functions that don't return anything should be declared `void`
- Functions that don't take any parameters should have `void` as the quasi-formal parameter

Example: Write a function that takes two pointers to ints and swaps their values. Then, write some more code to call that function.

Prototypes. The C compiler needs to know what types a function takes when encountering any calls to that function. For example:

```
int main(int argc, char **argv) {
    foo("something", 45);
    return 0;
}
```

would cause an error because the compiler doesn't know what types the function `foo` takes.

Problem can be solved by declaring the function "prototype" to tell the compiler about the type signature of the function. For library functions, that's why you do a `#include`: all the function prototypes are listed in the header (`.h`) file.

Fixing the above might result in:

```
void foo(char *, int); // function prototype for foo

int main(int argc, char **argv) {
    foo("something", 45);
    return 0;
}
```

C standard library: many (essentially) built-in functions to C. (They're actually in a library called libc.so that is automatically linked to an executable file during the linking process.)

- `stdio.h`: `printf`, file I/O operations
- `stdlib.h`: `strtol`, `strtod`, `malloc`, `free`
- `string.h`: `strlen` and many other string-related functions

item Functions can be declared `static`: this means that function is only callable by other functions in the same .c file

Storage classes and heap allocation

"automatic" storage class Any variables that are automatically allocated on the stack as part of function calls or local variable declarations. Memory is allocated "automatically" by the compiler.

`static` storage class two cases: when a function is declared static in a `.c` file, it means that it can only be called by other functions in the *same* `.c` file. second case: when a variable is declared static, it means that storage for it is allocated statically (once) at compile time.

heap storage Memory that is allocated on the (duh) heap. In C, you must use the `malloc` function to allocate memory from the heap.

You can dynamically allocate memory using `malloc`, and return it to the pool of free memory using `free`:

```
int *buffer;
int i;
buffer =  (int *) malloc(10 * sizeof(int));
for (i = 0; i < 10; i++)
{
    buffer[i] = i;     // NB: array notation works with pointers

    // alternatively: *(buffer+i) = i;
}
```

Running programs (processes) have three places from which memory is allocated: a statically allocated segment, the stack, and the heap. View process address space as a linear array; stack grows "down" and the heap grows "up". (Lots more on address spaces to come...)

Stack

- Includes memory for local variables, parameters

- When functions return, the associated activation record is popped from the stack - *all stack variables are lost!*

Heap

- Must be manually allocated/deallocated using `malloc` and `free`
- Memory allocated from the heap can have an arbitrary lifetime
- It's the programmer's responsibility to keep track of heap references in C!

Example:

```
void func() {
    char buffer[10];
    char *buf = (char *)malloc (sizeof(char)*10);
}

int main(int argc, char **argv) {
    func();
}
```

What do the stack and heap look like:

- when `main` is invoked?
- when `func` is pushed on the stack?
- after `buf` has been allocated?
- after `func` has returned?

What happens with the two return statements?

```
char *func() {
    char buffer[10];
    char *buf = (char *)malloc (sizeof(char)*10);
    // scenario 1: return &buffer[0];
    // scenario 2: return buf;
}

int main(int argc, char **argv) {
    char *c = func();
}
```

`main()` and command-line arguments

`main` takes two parameters: an `int` and an array of pointers to `char`:

```
int main(int argc, char **argv) { ... }
```

Explanation:

- `argc` indicates the number of parameters passed to program, *including* the program name
- `argv` is an array of pointers to C strings, each pointing to individual tokens of the command line; last entry in `argv` is `NULL`
- `argv[0]` is *always* the program name
- `argv[argc]` is always `NULL`

Example: `$ ./test 0 1 2 3`. What are `argc` and `argv`?