Intro to C
Lecture 4
History

• Originally created to build UNIX!
• Circa ~1970
• First ran on the PDP-11
• C is important because it is one of the first high-level languages to expose the machine!
Primitive Types

• Integer types:
  • int (4 bytes typically)
  • unsigned (unsigned 4 bytes)
  • char (signed character, 1 byte)
  • short (2 byte signed integer)
  • long (length depends, but 4 or 8 bytes)
  • long long (8 byte signed integer)
  • float/double (4/8 byte floating point value)
/* these are variable declarations and initializations */
int x;
short y;
char w = 120;
unsigned char z = 255;
float f = 3.1415926;
Control Flow

// basic conditional
if (test expression) {
   // thing to do
}

int x;
// x gets manipulated
if (x > 42) {
    printf("larger than life\n");
} else if (x < 42) {
    printf("smaller than life\n");
} else {
    printf("life\n");
}
Control Flow

```c
int x = 100;

while (x > 0) {
    printf("%d\n", x);
    x--; // same thing as x = x - 1;
}
```
Control Flow

// same thing, more compact
int x;
for (x = 100; x > 0; x--) {
    printf("%d\n", x);
}

// breaking out
int x;
for (x = 100; x > 0; x--) {
    if (x == 33) {
        break;
    }
}
Control Flow

// skipping iterations
int x;
for (x = 100; x > 0; x--) {
    if (x == 33) {
        continue;
    }
}

Comparison Operators

• == testing equivalence
• != testing non-equivalence
• > greater than
• >= greater than or equal to
• <= lte
• < lt
Logical Operators

- `||` Logical Or
- `&&` Logical And
- `!` logical inversion
Arithmetic Operators

• +, +=
• -, -=
• /, /=
• *, *=
• %, %=
Bitwise Operators

- >> right shift (more on this much later)
- << left shift
- & (bitwise and)
- | (bitwise OR)
- ~ (Bitwise negation)
- ^ (bitwise XOR)
Functions

// functions have a return type
int foo (int arg1, char arg2, float arg3) {

    // stuff

    return 0; // <- must be int

} 

void main () { foo(10, ‘b’, 3.0); }
Structs

• *Compound* data types
• Much simpler and more limited than classes
• structs have *fields*
• The C compiler just lays the fields out sequentially in memory (but there are some subtleties here)
struct foo {
    int a;
    short b;
    char c;
};

int main () {
    struct foo f1;
    f1.a = -10;
    f1.b = 20;
    f1.c = ‘z’;
}
Arrays

- Arrays are just sequentially laid out memory locations that store particular types
- `int x[100]; // array of 100 integers`
- `struct foo foo_array[100]; // array of 100 foo structs`
Pointers

• The MOST important feature of C
• Also the most dangerous, confusing
• But really, just a memory address
POINTERS ARE ADDRESSES

• This **forces** you to think about what’s happening under the covers

```c
int * x; // pointer to an integer (address of an integer)
char * y; // pointer to a char (address of a char)
struct foo * f; // pointer to a struct
```
Pointers can point to anything (!)

• including nothing in particular

```c
void * y; // pointer to no type in particular. This really is just an address
int * x = 102948;

*x = 100; // What does this do?
```
Pointer Arithmetic

void * y = 1989303;
int * x = 102948;

x ++; // ??

*y = 0; // ??

*(x+4) = 20; // ??
Sometimes we need to go the other way

• Given some variable, give me its memory address
• Use the address-of operator (&)

```c
int x = 100;

int * ptr_to_x = &x; // the address of an integer

*x += 1; // what happens here?
```
Strings

• “Null terminated” collection of characters
• Typically sit behind a char* (a string is an address of the beginning of a character array)

• char * x = “hello”; // same as prev
Argument passing

• In C, arguments are passed by VALUE. The argument values are copied into the storage area for the function
Macros

• Compiler just does a search and replace BEFORE compilation:

```c
#define THING 10

int main () {
    int x = THING;
    printf("%d\n", x);
    return 0;
}
```