Introduction to C

Geared toward programmers

Robert Escriva

Slide heritage: Alin Dobra → Niranjan Nagarajan → Owen Arden

Cornell CS 4411, August 30, 2010

1 Why C?

2 A Quick Example

3 Programmer's Responsibilities

4 Language Basics
- Pointers and Memory Management

5 Program Structure

6 Style

Portability

- Most operating systems have a C compiler available ...
- ... (partially) because most operating systems are written in C.

Linux, {Free,Open,Net,Dragonfly}BSD, Minix, XNU

Predictability

- No unintended memory allocations.
- Control of layout of structures in memory.
- Simple syntax makes determining behavior easy.

Power

- Direct manipulation of control flow.
- Complex datastructure creation.
- Assembly is just a step away.

But great power can corrupt... *

* which will it be, the stack or the heap?

Goals for today

- Provide a broad base for using/learning C.
- Warn you of pitfalls related to C programming.
- Practice reading/writing C.

Hello World

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    printf("Hello World!\n");
}
```

Try it out

- Create `hello.c`[†].
- In a VS2008 command prompt run:
  - `cl hello.c`
- Now run the `hello.exe` binary.

[†] or fetch it from the course webpage

Learning a new language

- Languages are largely the same.
  - Notice the similarities.
  - Control structures (loops, conditionals, etc.)
  - Data types (int, char, classes, etc)
- Embrace diversity.
  - Play on a language's strengths.
  - Don't use a square peg for a round hole.

The Programmer's Responsibilities

- Memory maintenance (no automatic garbage collection here).
- Error handling (no exceptions here).

Language Basics

- Control flow
- Data types
- Operators
- Preprocessor

Control Flow

- `if (x) { } else if (y) { } else { }`
- `for (pre(); loop_invariant(); post()) { }`
- `while (x) { }`
- `do { } while (x);`
- `switch (x) { case 0: break; default: act(); }`

Operators

- Arithmetic: `+`, `-`, `*`, `/`, `%`
- Relational: `<`, `>`, `<=`, `>=`, `==`, `!=`
- Logical: `&&`, `||`, `!`, `?:`
- Bitwise: `&`, `|`, `^`, `!`, `«`, `»`

Primitives

- Integer types:
  - `char` : character, typically one byte.
  - `int`, `short`, `long` : integers of different sizes.
  - Optional prefixes `signed` or `unsigned`.
- Floating point types:
  - `float` : single-precision floating point.
  - `double` : double-precision floating point.
- No booleans:
  - 0 $\Rightarrow$ false
  - $\neq 0 \Rightarrow$ true

Sample Primitive Declarations

```
char a = 'A';
char b = 65;
char c = 0x41;

int i = -2343234;

unsigned int ui = 4294967295;
unsigned int uj = ((long) 1 << 32) - 1;
unsigned int uk = -1;

float pi = 3.14;
double long_pi = 0.31415e+1;
```

Enums

```
enum months
{
    JANUARY,
    FEBRUARY,
    MARCH
};

enum days
{
    SUNDAY,
    TUESDAY = 2,
    WEDNESDAY
};
```

- Values are consecutive integers starting from zero.
- Explicitly assigning a value causes the sequence to continue from that value (e.g., `WEDNESDAY == 3`).

Preprocessor

- Inclusion of other files:
  ```
  #include <header.h>
  ```
- Definition of constants:
  ```
  #define HOURS_PER_DAY 24
  ```
- Creation of macros:
  ```
  #define CELSIUS(F) ((F - 32) * 5/9.)
  ```

Pointers hold memory addresses

- An int: `int foo;`
- Pointer to an int: `int* ptr_foo;`
- Pointer to a pointer to an int: `int** ptr_foo;`

Pointer operations

- Obtain the address of a variable: ‘`&`’
- Dereference a memory address: ‘`*`’

Pointer Example

```
int x;
int* ptr; /* ptr points to an
             undefined location */

ptr = &x; /* ptr now points to
             integer x */
*ptr = 3; /* integer pointed to by
             ptr is now 3 */
```

Global Variables

- Declared outside all functions.
- Space allocated before execution.
- Space deallocated at program exit.
- Learn the meanings of `static` and `extern`.

Local Variables

- Declared at the start of the function.
- Space is allocated on the stack.
- Initialized before function execution.
- Deallocated when leaving the function scope.
  - Also known as deleting the stack “frame”.

Heap Variables

- Allocate with `malloc()` and deallocate with `free()`.
  ```
  void* malloc(int)
  void free(void*)
  ```
- It is up to you to manage memory.
- Never calling `free()` leaks memory.
- Calling `free()` more than once will crash your program. ‡

‡ Known as a “double free” bug, and is sometimes exploitable.

Malloc/Free Example

```
int* ptr; /* pointer to an int */

/* allocate space to hold an int */
ptr = (int*) malloc(sizeof(int));

/* check if successful */
if (ptr == NULL) exit(1);

*ptr = 4; /* store value 4 in the int */
printf("ptr: %p %d\n", ptr, *ptr);

/* deallocate memory */
free(ptr);
```

Warning

- Dereferencing an un-initialized pointer can crash your program (or worse)!
- Consider initializing a pointer to NULL and checking before dereferencing.
- Some functions return NULL on error.
  - Pay attention to the function specification!
  - Check return values!

Arrays and Strings

- Arrays

```
for (i = 0; i < 10; ++i)
{
    A[i] = i * i;
}
```

- Strings

```
char name[] = "CS4410";
name[5] = '1';
```

- Functions to operate on strings in `string.h`.
  - `strncpy`, `strncmp`, `strncat`, `strstr`, `strchr`

Functions

- Arguments can be passed:
  - by value: a copy of the parameter is passed into the function.
  - by reference: a pointer is passed into the function.
- Return values are also by value or by reference.

Pass by Value

`swap_value()` only changes `n1` and `n2` within the local scope.

```
void swap_value(int n1, int n2)
{
    int temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
}
```

Pass by Reference

`swap_reference()` changes the values at the memory locations pointed to by `p1` and `p2`.

```
void swap_reference(int* p1, int* p2)
{
    int temp;
    temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

Function Pointers

```
int inc(int i) { return i + 1; }
int dec(int i) { return i - 1; }
int apply(int (*f)(int), int i)
{
    return f(i);
}
int main(int argc, char* argv[])
{
    printf("++: %i\n", apply(inc, 10));
    printf("--: %i\n", apply(dec, 10));

    return 0;
}
```

Structures

```
struct list_elem
{
    int data;
    struct list_elem* next;
};

int main(int argc, char* argv[])
{
    struct list_elem le = { 10, NULL };
    le.data = 20;

    return 0;
}
```

Typedefs

- Create an *alias* for a type.
- Syntax: `typedef type alias`
- Use it like any primitive: `list_elem_t le;`

```
typedef struct list_elem
{
    int data;
    struct list_elem* next;
} list_elem_t;
```

Code Style

- Write comments where appropriate!
  - Bad comment:

    ```
    /* print the value of x */
    ```
  - Good comment:

    ```
    /* take one process off the ready queue */
    ```
- Avoid magic numbers.
  - Create logically named constants.

Code Style : Organization

- Pick a style (e.g., K&R, Allman, BSD, GNU) and stick with it.
- Use a consistent style for naming variables and functions.
- Split large functions into smaller functions.
- Your dependency graph should be more like a spanning tree than a clique.

Build Tools and Version Control

- Build systems:
  - Organizes compilation commands and dependencies.
  - Repeatable, incremental compiling.
  - `make`, `pmake`, `cmake`, `scons`, etc.
- Version control:
  - Keep track of all changes.
  - Simplifies merging changes from all developers.
  - `Git`, `Subverion`, `CVS`, `Mercurial`, etc.

Summary

- C is a great language (especially for systems work).
- Learn by doing. All examples are on the course web page.
- Save yourself some trouble:
  - Initialize variables before use.
  - Don’t return pointers to stack-based variables.
  - Allocate and deallocate memory properly.
  - Check return values.