C Crash Course

UC Berkeley

CS61C

helloworld.c

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("hello world!\n");

  return 0;
}
```

include

```
#include <stdio.h>
```

- Use the `#include` statement to include other C files
- Common includes are stdio.h, stdlib.h, math.h
- Generally include `.h` files to get function and variable declarations

```
#include <stdio.h>
```

vs.

```
#include "stdio.h"
```

- "" looks through current directory, while <> looks through system library folders

main

```
int main(int argc, char *argv[]) {
  /* Code */
}
```

- `main()` is a special function where execution of a C program starts.
- `argc` and `argv` are automatically passed as arguments
- `argc` is the number of arguments
- `argv` is an array containing the arguments

printf()

- `printf()` prints data to the screen
- Takes a variable number of arguments
- First argument is a **format string**
- Other arguments are optional, are inserted into the format string in the place of special sequences of characters

printf()

```
printf("hello world");
hello world
printf("5 == %d", 5);
5 == 5
printf("Char: %c, Double: %f", 'a', 1.2);
Char: a, Double: 1.2
printf("no newline");
printf("causes a run-on");
no newlinecauses a run-on
printf("line1\nline2");
line1
line2
```

Variables

- A variable is a named space in memory to store data
- In C, variables need to be **declared** before you can do anything with them
- After being declared, a variable is usually **initialized** to some initial value before being used
- A variable has a **type** and a **name**

```
int int1; // declares an integer named int1
double d; // declares a double
int1 = 5; // int1 is given the value 5
int int2 = 3; // int2 is declared and initialized
              // in one line
```

C Keywords

Variables/functions/structs may **not** be named after any keyword:

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

Arithmetic Operators

```
int x = 3, y = 4 + 4, z = 12 / 3;
printf("x: %d, y: %d, z: %d", x, y, z);
x: 3, y: 8, z: 4
z = x + y; // z is now 13
z = x + y * y; // z is now 67
double d;
d = x / y; // d is 0, C truncates integer division
d = ((double) x) / y; // now d is .375
```

Key points to keep in mind:

- Numbers in C have min/max values, unlike Scheme
- Remember to cast before dividing if you don't want integer truncation!

Bitwise Operators

```
& bitwise AND                 << left shift
| bitwise OR                  >> right shift
^ bitwise XOR                  ~ bitwise complement

 011010101                     00101101
&000000011                    |11110000
 000000001                     11111101

10000001 << 3 = 00001000
10000001 >> 3 = 00010000 or 11110000 (depending on
  whether 10000001 was a signed number or not)
```

Arrays

- An array is a contiguous segment of memory filled with values of the same type
- Arrays in C must be given a size when declared

```
int arr[10]; // declares an array of size 10
arr[0] = 3; // sets first element of arr to 3
```

Control Statements

```
if(pred) {
  /* code to run if pred is true */
} else {
  /* code to run if pred is false */
}

if(pred1) {
  /* code to run if pred1 is true */
} else if(pred2) {
  /* code to run if pred2 is true */
} else {
  /* code to run if neither is true */
}
```

Control Statements

```
char c;
switch(c) {
  case 'a':
    printf("a\n");
    break;
  case 'b':
    printf("b\n");
  case 'c':
  case 'd':
    printf("after b\n");
    break
  default:
    printf("error\n");
}
```

Control Statements

```
int i = 0;
while( i < 10 ) {
  printf(“i: %d\n”, i);
  i = i + 1;
}

int j;
for( j = 0; j < 10; j = j + 1) {
  printf(“j: %d\n”, j);
}
```

Control Statements

```
int i = 0;
while( 1 ) {
  if(i < 10) {
    continue;
  }
  printf("i reached 10!\n");
  i = i + 1;
  if(i > 10) {
    break;
  }
}
```

```
i reached 10!
```

Functions

- Use functions to break a large task into manageable small chunks
- Functions allow code to be reused (such as printf, atoi, etc.)
- Functions have a **name** and a **return type**
- Functions need to be declared and defined
- Generally happen at the same time, but not necessarily

Functions

```
int foo(); // declares a function foo

// definition of foo
int foo() {
  return 7; // returns something of type int
}

// declare and define a function at the same time
double caster(int x) {
  return (double) x;
}
```

Functions

- Arguments to functions are passed by value; this means that if we pass a variable as an argument to a function, the value of the variable is copied. Changing the copy does nothing to the original

```
void foo(int arg) { arg = 10; }

int main() {
  int x = 17;
  foo(x);
  printf("x:%d\n", x);
}
```

```
x:17
```

Pointers

- A pointer is a variable which points to data at a specific location in memory
- A pointer has a type; this is the type of data it is pointing to
- Key to doing many interesting things in C, such as functions that can change the value of a variable and dynamic memory management (more on memory in lecture)
- Can have a pointer to a pointer (to a pointer to a …)

Pointers

```
int x = 1, y = 2, z = 3;
int *p1, *p2; // declares two pointers to ints
p1 = &x; // p1 contains the address of x
y = *p1; // * dereferences p1, so y = 1
p2 = p1; // p2 points to the same thing as p1
*p2 = 4; // x is now 4
```

Pointers

```
void swap(int x, int y) {
  int tmp = x;
  x = y;
  x = tmp;
}

int a = 1, b = 2;
swap(a, b); // a and b did not get swapped
```

Pointers

```
void swap(int *x, int *y) {
  int tmp = *x;
  *x = *y;
  *x = tmp;
}

int a = 1, b = 2;
swap(&a, &b); // a and b did get swapped
```

Structures

- Used to define compound data types
- Can contain data of different types
- Useful for organizing and packing up related data. For example, in a 2D graphics program, might have structs to represent a point

Structures

```
struct point {
  int x;
  int y;
};

struct point p1, p2; // declares two variables
                     // of type struct point
p1.x = 3; // sets x of p1 to 3
p1.y = 5; // sets y of p1 to 5
```

Structures

- Can `typedef` to shorten the type name

```
typedef struct point point_t;
point_t p3; // equivalent to struct point p3;
```

- Can use user defined types inside a struct

```
struct rect {
  point_t ll; // lower left
  point_t ur; // upper right
}
```

Structures

- Functions can return structures

```
point_t makePoint(int x, int y) {
  point_t p;
  p.x = x;
  p.y = y;
  return p;
}
```

- Can use user defined types inside a struct

```
struct rect {
  point_t ll; // lower left
  point_t ur; // upper right
}
```

Memory Management

- You need to manage your own memory in C!
- Variables can be static, local, or malloc'ed
- Static variables live in special section of program, only 1 copy
- Local variables allocated automatically when a function is called, deallocated automatically when it returns
- Dynamic storage is managed through the function `malloc()`
- Malloc returns a pointer to a chunk of memory in the heap
- Use when we don't know how big an array needs to be, or we need a variable that doesn't disappear when a function returns

Memory Management

```
int main() {
  int x = 5; // x is on the stack
  // y is a pointer to a chunk of memory
  // big enough to hold one int
  int *y = (int *) malloc(sizeof(int));
  // double is a pointer to a chunk of memory
  // big enough to hold 10 doubles
  double *z = (double *) malloc(10 * sizeof(double));
  if(z == NULL) { exit(1);} // something went wrong...
  // we can access the memory z points to
  // as though z was an array
  z[5] = 1.1;
}
```

Memory Management

- What happens to memory given out by malloc when we're done with it?
- Answer: nothing, unless we do something about it!
- Need to say we're done with a chunk of memory when we don't need it anymore
- Use function `free()` to free memory. `free()` takes a pointer given out by malloc, and frees the memory given out so it can be used again
- Forgetting to call free is a cause of a significant percentage of memory leaks…

Memory Management

```
// arrays made without malloc are freed automatically
void ok() {
  int arr[10];
  return;
}

/* arr is never freed; since function returned, we lost
  the only pointer we had to the memory we malloc'ed! */
void leaky() {
  int *arr = (int *) malloc(10*sizeof(int));
  return;
}
```

Useful Data Structures

- Linked List

```
// example with a linked list of integers
struct node {
  int node_value;
  struct node *next; // pointer to next node
};
typedef struct node node_t; // optional
node_t *head = (node_t *) malloc(sizeof(node_t));
head->value = 0;
head->next = (node_t *) malloc(sizeof(node_t));
```

Useful Data Structures

- Binary Tree

```
// example with a linked list of integers
struct node {
  int node_value;
  struct node *left; // pointer to left child
  struct node *right; // pointer to right child
};
typedef struct node node_t; // optional
node_t *head = (node_t *) malloc(sizeof(node_t));
head->value = 0;
head->left = (node_t *) malloc(sizeof(node_t));
head->right = (node_t *) malloc(sizeof(node_t));
```

I/O

- `printf()` is your all-purpose output function to the console
- Reading from standard in:
  - `getchar()` – returns the next character typed in
  - `gets(char *buf)` – reads one line into the given buffer
- Opening a file:

```
FILE *f = fopen("foo.bar", "rw")
```

- Reading/writing from a file:

```
int next_char = getc(f);
putc('a', f);
```

- Remember to close your files when done

```
fclose(f);
```