Logistics

Reading
For today Zyante
  ▷ Ch 3 (functions)
  ▷ Ch 4 (conditionals)
For Thursday
  ▷ Ch 5 (iteration)
  ▷ Ch 6 (arrays)

Exam 1
  ▷ Next Week Thursday
  ▷ Zyante Ch 1-6
  ▷ This week’s Material Included

HW 1
  ▷ Due tonight by 11:59 pm on Blackboard
  ▷ Don’t forget directory structure: ckauffm2-hw1
  ▷ Don’t forget ID.txt

HW 2
  ▷ Up tomorrow morning, due next Tuesday
  ▷ Conditionals, loops, possibly some arrays
Spec
Questions?
Today

- Comments
- Statements/Expressions
- Variable Types
- Assignment
- Basic Input/Output
- Function Declarations (Session 1)
- Conditionals (if-else) (Session 2)
- Iteration (loops)
- Aggregate data (arrays, structs, objects, etc)
- Library System
Goals

- Write functions
- Vague idea of low level execution
- Zyante Ch 3
  - Many more details in Ch 3 than we’ll discuss
  - Ex: Loops/Conditionals in functions
  - Ex: Pass by pointer
  - Read that material and try to understand
  - These will become clearer in the near future
Functions

- What’s a function?
  - Traditional Math?
  - Programming?
- Why write code with functions?
Terminology

Function ≡ Procedure ≡ Method ≡ Routine ≡ Action Abstraction

Parts: Return Type, Name, Arguments, Body

```c
return_type function_name(arg_type1 arg1, arg_type2 arg2){
    body_line_1;
    body_line_2;
    ...
    return something_of_return_type;
}
```

Examples

```c
int halve(int arg){
    int result = arg / 2;
    return result;
}
```

```c
void print_greeting(){
    printf("Welcome to");
    printf("functionland\n.");
}
```
Whitespace is arbitrary

Other frequent arrangements of functions

```c
return_type function_name_2(arg_type1 arg1,
                            arg_type2 arg2){
    body_line_1;
    body_line_2;
    ...
    return something_of_return_type;
}
```

```c
return_type
function_name_2(arg_type1 arg1,
                arg_type2 arg2)
{
    body_line_1;
    body_line_2;
    ...
    return something_of_return_type;
}
```
Calling Functions

Calling ≡ Invoking ≡ Run Body with Actual Parameters

```c
int halve(int arg){
    int result = arg / 2;
    return result;
}

int main(){
    int n = 12;
    int halved = halve(n);
    printf("n is %d and half n is %d
", n, halved);

    printf("Halve 19 now: %d\n", halve(19));
    return 0;
}
```

- arg is the **formal parameter** to halve
- Takes on **actual value** 12 and 19 during main
Declarations vs Definitions

Prototypes Declare a function exists

- Return Type
- Name
- Number and Types of Arguments

```c
int my_function(double x, int y, char c);
```

Definition of functions involve a body

```c
int my_function(double x, int y, char c){
  do_something;
  do_something_else;
  ...;
  return an_int;
}
```
/* Get some prototypes of mathy stuff*/
#include <math.h>

/* Prototype: name and types only */
int my_function(double x, int y, char c);

/* Definition of function */
int my_function(double x, int y, char c){
    double result = x*2;
    result = result + y;
    result = result - ((int) c);
    return (int) floor(result);
}
Exercises

Write this function

// Normalize a score by subtracting the mean
// and dividing by the standard deviation
double normalize(double score,
                   double mean,
                   double stddev);

// Return the positive root of the quadratic
// defined by a*x^2 + b*x + c; this is found
// by adding the sqrt of the discriminat
// in the quadratic equation rather than
// subtracting it
double pos_root(double a, double b, double c);
Declaration and Definition may be in different files

Often divide function declaration of functions into Header files (.h) and Implementation files (.c).

Declaration numerical.h

```c
// Example header file
#ifndef NUMERICAL_H
#define NUMERICAL 1

// Return half the argument given
int halve(int arg);

// Normalize a score by subtracting the mean
// and dividing by the standard deviation
double normalize(double score,
   double mean,
   double stddev);

// Return the positive root of the quadratic
// defined by a*(x*x) + b*x + c; this is found
// by adding the sqrt of the discriminant
// in the quadratic equation rather than
// subtracting it
double pos_root(double a, double b, double c);

#endif
```

Definitions in numerical.c

```c
#include <math.h>
#include "numerical.h"

int halve(int arg){
   int result = arg / 2;
   return result;
}

double normalize(double score,
   double mean,
   double stddev){
   return (score - mean) / stddev;
}

double pos_root(double a, double b, double c){
   double discriminant = b*b - 4*a*c;
   double rootDiscr = sqrt(discriminant);
   double root1 = (-b + rootDiscr) / (2 * a);
   return root1;
}
```
A very special function: main

Where the action begins - a time-honored C convention

▶ Programs have a main
▶ Libraries (usually) don't
▶ Notice: numerical.c has no main()
▶ Does not comprise a program, only a library of functions
▶ numerical_main.c does have a main but not definitions of numerical functions, only header numerical.h
▶ Compile all together

> gcc numerical.c
(.text+0x20):
  undefined reference to ‘main’
collect2: error:
  ld returned 1 exit status

> gcc numerical_main.c
/tmp/ccIqZXiy.o: In function ‘main’:
numerical_main.c:(.text+0x15):
  undefined reference to ‘halve’
numerical_main.c:(.text+0x39):
  undefined reference to ‘halve’
collect2: error:
  ld returned 1 exit status

> gcc numerical.c numerical_main.c
> ./a.out
n is 12 and half n is 6
Halve 19 now: 9
Input a b c: 12 3 -5
Pos root is 0.5325
Returning Things

With `return`, see `returns.c`

- What about 2 or 3 or more return values?
Blocks and Scope

Blocks defined by \{ \}\ 

- Groups code together
- Defines a scope
  - Variable visibility
  - Hierarchy of scopes
  - Contrast: Python

What can a function see?
Functions have their own scope

- Arguments
- Global data/functions
- Its block variables

See scopes.c, scopes2.c, badscopes.c
Call Stack

Functions call functions call functions
- Compiler/Runtime keeps track
- Easy to draw
Functions are translated to memory manipulations

- Caller \( f \) is executing
- Callee \( g \) is being called by \( f \)
- Caller \( f \): push args onto stack, save registers, jump to \( g \)
- Callee \( g \): execute, put answer on stack, jump back (to \( f \))
- Caller \( f \): restore registers, grab answer, continue

Demonstrate with callstack.c
The Stack

A spot in memory

- Data for each function call
- Arguments, locals, return value
Inlining

Jumping around can be expensive

- Instructions to save registers, push args
- **Inline** means copy definition there
  ```
  inline int max(int a, int b) {
      return a > b ? a : b;
  }
  ```
- Suggests compiler inline a function
- No guarantees of speed
- Compiler may not honor
- May inline without you saying it
How long?

Using functions is good, right?

- How do you decompose a large problem into functions?
- What merits a function?
- How long should a function be?

Try Code Complete by Steve McConnel

- Online At GMU Library

This is a quality of people, not machines.
BREAKTIME

Back in 15 minutes
Goals

- Zyante Chapter 4
- Conditionals
- Comparing Numbers
- `switch/case` (maybe...)
Making Choices

Straightline code is about as interesting as Ikea instructions: rigid.
Simplest Form of if

Always do this;
if(condition)
    sometimes do this;
Always do this;

Always;
if(condition){
    sometimes this;
    and this;
    and this;
}
Always;

See if_test.c
Using Blocks

CK’s preference - always use

if(...){
    ...
    ...
}

Do what works for you
    • Or what your boss forces you to do
Comparing things

= Assignment, NOT comparison
== Equality test
!= Inequality
<> Less / Greater
<= >= Less than equal / Greater than equal

See comparisons.c
Consequence and Alternative

Often have 2 cases, C provides nice syntax

Always;
if (cond) {
    do when true;
}
else {
    do when false;
}
Always;
**Boolean Combinations**

To combine conditions
Test more than one thing at once

```c
&& and
|| or
! not
```

See booleancomb.c

**Truthy/Falsey**

Which things are truthy and falsy in C again?
Combining if elses

**Nesting**  Arbitrary nesting of conditionals, nesting.c

**Chaining**  Mutually exclusive cases, chaining.c
Two very common errors

// Different meaning than intended
if(cond)
    do me;
    do me too;
always;

// Not accepted by compiler
if( 0 <= i <= 10)
Exercises

Define an absolute value function for single integers

```cpp
int a = abs(7); // 7
int b = abs(-2); // 2
int c = abs(0); // 0
```

Define an absolute minimum function for three real numbers

```cpp
double x = absmin3(1.4, 0.5, -2.8); // 0.5
double y = absmin3(-1.4, 0.5, -0.1); // 0.1
double z = absmin3(-1.4, 5.5, -6.1); // 1.4
```
C standard does allow for use of keywords `true` and `false` with type `bool` by including the `stdbool.h` header.

`booleancheck.c`:

```c
/* Demonstrate the use of stdbool.h to define the names true and false */
#include <stdio.h>
#include <stdbool.h>

int main(){
    bool t = true; /* defined to be 1 */
    bool f = false; /* defined to be 0 */
    printf("%d %d\n",t,f);
}
```
Composing

- Conditionals are if/else, switch/case
- Conditionals inside functions
- Conditionals inside other conditionals
  - Nesting if/else
  - Nesting switch/case
- Functions inside conditionals?
  - Sort of - preprocessor as #IF
- Functions inside functions?
  - Go-go gadget gcc
Wrap-Up

- Comments
- Statements/Expressions
- Variable Types
- Assignment
- Basic Input/Output
- Function Declarations
- Conditionals (if-else)
- Iteration (loops)
- Aggregate data (arrays, structs, objects, etc)
  - Sans memory ops
- Library System

Exam 1 Next Week