course link
- syllabus
- calendar

goals
- learn imperative language
- learn how to use memory allocation → "fluent programmer"
- learn data structures
- experience w/ Linux
- strengthen programming skills

Piazza → Ed Discussion

always public unless very private

MWF live lectures & 1 hour discussion on M

amr @ cs.uchicago.edu

Work
- Academic Honesty (Gradescope)
- 6 SEs — Mon → Sun
- 6 PAs — Sat → Fri
- 2 TAs — Wks 5 & Wk 9
- labs

Skills > Grades

New Language!

Share characteristics

- assign names — same
- eval. expression
- conditional execution
- defining fns

- level of abstraction — diff.
- Typing: static vs dynamic
- available control abstraction: lazy & strict
- efficiency (algorithmic, time, programmer)

Abstraction

Domain
High Level
Assembly
CPU
Memory I/O

Haskell
Typed Racket, Python
C

Why C?
under the hood code
required for higher level classes

Language	Type	Representation	Operations
Matlab	built-in	built-in	built-in
Python	Optional library	Optional library	Optional library
Java	User-defined class	Built-in	User-defined as part of class
C	User-defined	Built-in	User-defined functions
Assembly Language	None	User-defined	User-defined

terminal vars
code output

```
#include <stdio.h>          seperate lines for each import

                            -> implicit return 0 . returns to terminal
int main ( )
{
    printf("hello world \n");
}
```

clang hello.c → a.out file by default
./a.out

how to interpret warning

clang -g -Wall -O2 hello.c -o hello

-g → include info. to debug
-Wall → give me all warnings, pls
-O2 → efficiency
hello.c → C file
-o → name of executable
hello

echo $? → return value in terminal

Make is fun

<type> <var name> [= <expression>] (optional)

int a = 2 //comment

temp variable: not so descriptive
be descriptive

2 sizes of floats: floats or double

/* multi line
comment */

characters ASCI

```
char x = 'g';
```

single quote for chars

multiple vars in a line

```
int x=0, y = 1;
int b = x+1-y;
double r, s = 5.0;
```

r → undefined, don't rely on auto declaration
don't use var before giving value

format strings

```
printf("x: %d, y: %d", var1, var2)
```

%d → interpret corresponding param as int
%f → " " a double

can't change types, only value

Type	Range of values	Notes
char	$-2^7 \dots 2^7 - 1$	Used to represent characters and small integers
short	$-2^{15} \dots 2^{15} - 1$	
int	$-2^{31} \dots 2^{31} - 1$	
long long	$-2^{63} \dots 2^{63} - 1$	
float	$-3.4 \times 10^{38} \dots 3.4 \times 10^{38}$	IEEE floating point
double	$-1.7 \times 10^{308} \dots 1.7 \times 10^{308}$	IEEE floating point

want only 0 & positive #'s?

unsigned <type>

unsigned int

can assign integers w/ specific sizes

found in <stdint.h>

int8_t (8 bytes) int16_t (16 bytes)
uint8_t unassigned

MIN_INT & MAX_INT → smallest & largest signed ints
→ usr/include

"%[flags][width][.precision][length]specifier"

Specifier:
- c - character
- d - signed integer
- u - unsigned integer
- f - double
- s - string

Length:
- h - short
- ll - long long

Width
minimum number of characters

Precision
number of digits after the decimal place (floating point values)

Flags
padding/justification (see manual)

Operator	Operation	Notes
+	addition	Result type will be a double, unless both operands are integers
-	subtraction	Result type will be a double, unless both operands are integers
*	multiplication	Result type will be a double, unless both operands are integers
/	division	Result type will be a double, unless both operands are integers. **Integer division truncates**
%	integer remainder	Operands must be integers; result is an integer.

if one # is double, answer is double

→ truncates if 2 ints

(double) 10) → cast 10 to double
or multiply by 1.0

Operator	Operation	Examples/Notes
=	assignment	x = y, evaluates y and then updates x with the resulting value
+=	addition assignment	x += y is equivalent to x = x + y
-=	subtraction assignment	x -= y is equivalent to x = x - y
*=	multiplication assignment	x *= y is equivalent to x = x * y
/=	subtraction assignment	x /= y is equivalent to x = x / y

Assignment operators

Operator	Operation	Example:	Notes
-	Unary minus	-x	Result has the same type as x.
++	Postfix increment	x++	Increments x. If used in an expression, the result is the value **before** the increment.
++	Prefix increment	++x	Increments x. If used in an expression, the result is the value **after** the increment.
--	Postfix decrement	x--	Decrements x. If used in an expression, the result is the value **before** the decrement.
--	Prefix decrement	--x	Decrements x. If used in an expression, the result is the value **after** the decrement.

Unary operators

Operator	Operation	Notes
==	equals	true iff the values of the two operands are the equal.
!=	not equals	true iff the values of the two operands are not equal
>	greater than	true iff the value of the first operand is **greater than** the value of the second operand
<	less than	true iff the value of the first operand is **less than** the value of the second operand
>=	greater than or equal to	true iff the value of the first operand is **greater than or equal to** the value of the second operand
<=	less than or equal to	true iff the value of the first operand is **less than or equal to** the value of the second operand

Relational

Operator	Operation	Example	Notes
&&	logical and	x && y	**true** iff the values of both operands are **non-zero**. *Short circuits:* will only evaluate the second operand if the first operand is non-zero.
||	logical or	x || y	**true** iff the values of one or both operands evaluate to **non-zero**. *Short circuits:* will only evaluate the second operand if the first operand is zero.
!	logical not	!x	**true** if the value of the operand is **zero** and false if the value of the operand is **non-zero**.

Logical Operators

bool not built in
<stdbool.h>
use this over 0 & 1
type bool, true, false → all defined

printf("has no way to print bool
status ? "yes" : "no"
true
false

more README .. cat README
↳ page by page
conditional expression

vim edits

Basics of Functions

Ed → site has asking etiquette
whole warnings

No code posting & screenshots
→ can be void → can be empty

```
<return-type> <name>( <param_list> )
{
    body
}
```

<name> → verbs

%s → string directive

call-by-value

evaluate arg. expression @ call site, pass thru value

declare a function:
<return-type> <name>(<param_list>)

"hey! there will be a fn w/ this name" don't define till later

useful b/c C compiler reads top to bottom
doesn't require naming params
provide type info (return & param) & name

declare ≠ define

multiple files

System header
my file directory #include "mycode.h"

mycode.c
all impl. stuff
includes mycode.h for declaration

mainlink
mycode.h
declare & types matchup

mymain.c
includes mycode.h
type signature

includes → header files
src → .c files
bin → executable files

Numbers (supplemental)

decimal, base 10 — using digits 0→9

512_{10}

↳ $2 \cdot 10^0 + 1 \cdot 10^1 + 5 \cdot 10^2$

binary, base 2 — absence or presence of current — "bit"

10011_2

↳ $1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4$

5 bits

byte - 8 bits

unsigned char (8 bits)	a = 5	00000101
unsigned char (8 bits)	b = 200	11001000

int is 32 bits

	Base	Digits
Decimal	10	0→9
Binary	2	0→1
Octal	8	0→7
Hexadecimal	16	0→9→A→F

A = 10 B = 11 C = 12 D = 13 E = 14 F = 15

to convert base X to decimal, multiply n^{th} digit from right by X^n

$A3F_{16} \rightarrow 15 \cdot 16^0 + 3 \cdot 16^1 + 10 \cdot 16^2 = 2623_{10}$

from base X to Y

$X = (X/Y) \cdot Y + (X \% Y)$

$165_{10} = (165/8) \cdot 8 + 165 \% 8$

$= 20 \cdot 8 + 5$

$= ((20/8) \cdot 8 + (20 \% 8)) 8 + 5$

$= (2 \cdot 8 + 4) 8 + 5$

↳ octal digit? ✓

$= 2 \cdot 8 \cdot 8 + 4 \cdot 8 + 5$

$= 2 \cdot 8^2 + 4 \cdot 8^1 + 5 \cdot 8^0$

binary to hex

pad w/ zero

$0110\ 1101\ 0011_2$ = 0110 | 1101 | 0011

6 | D | 3

$= 6D3_{16}$

hex to binary

$2CE1_{16}$ = 0010 1100 1110 0001

= 0010110011100001

= 10110011100001

leading zero: octal — 032 → "hey, interpret rest in octal"

leading zero-x: hex — 0x8AF

%u → unsigned int — decimal

%o → octal

%x → hex

Control Constructs

```
if (<expr1>) {
    <statements>
} else if (<expr2>) {
    <statements>
} else if (<exprN>) {
    <statements>
} else {
    <statements>
}
```

else: optional

no {}? → compiler thinks it's only 1 statement

```
while (<expr>) {
    <statements>
}
```

evaluate

body never executes w/ false (zero)

true → execute & repeat

```
do {
    <statements>
} while (<expr>);
```

always go at least once

optional

```
for ( <init statement> ; <test expr> ; <update> )
{
    <statements>
}
```

run init
evaluate test
- false → resume execution after loop
- true (nonzero)
  - execute statements
  - update
  - test

return statements in loop should be guarded by conditional

break — exit inner-most enclosing loop
continue — jump to next iteration of inner-most enclosing loop

for vs while

```
<init>
while ( <test> ){
    <body>
    <update>
}
```

better for unknown # of loops

```
switch ( <expression> ) {
    case <value1> :
        <statements>
        break ;
    case <value3> :
        <statements>
    case <value2> :
        <statements>
        break ;
    default :
        break ;
}
```

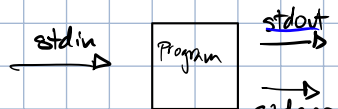
<expression> → must be integer

evaluates where expr == case

"falls through"

default: optional, not really

5 IO streams



(so that no errors go to unintended places)

stderr

printf → stdout

fprintf (stream , <string>)

exit (exit status) → "im done", does no more

↳ 1, or any nonzero number

enumerated types

```
enum <name for type> {
    <name A>,
    <name B>,
    :
    <name N>
}
```

can specify value