

CS330 Lab2

Make

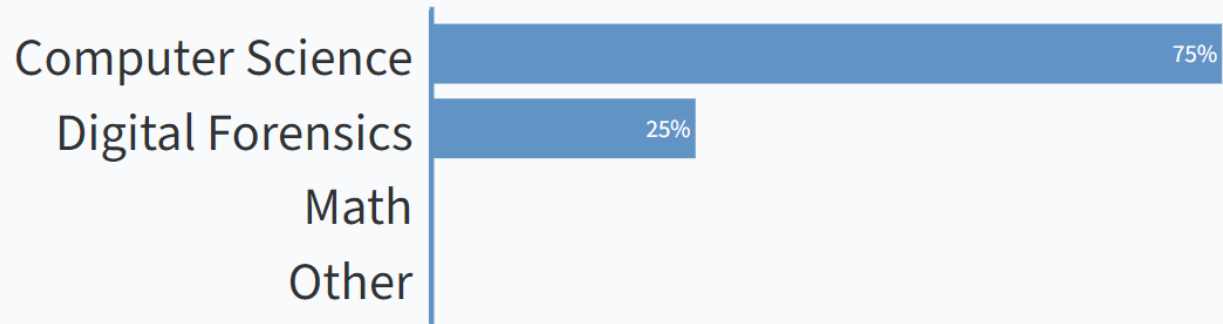
C: Intro, Variables, Loops, Functions

Spring 2022

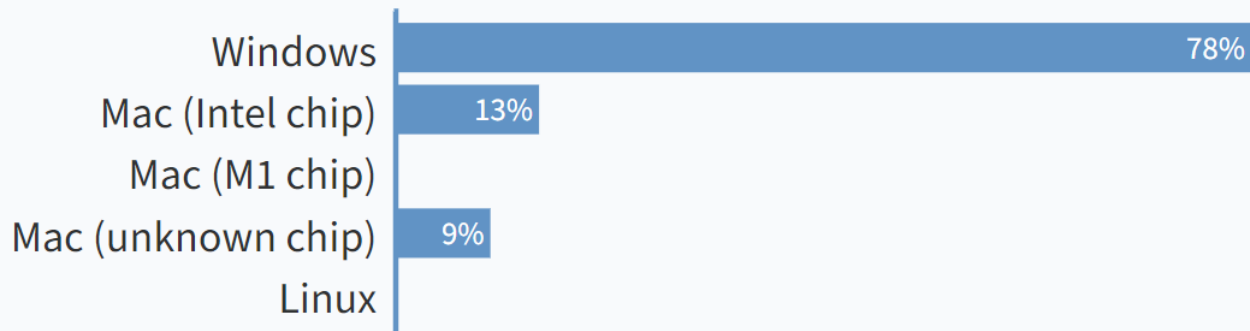
Survey Results (from last Lab) – Mon Lab

2

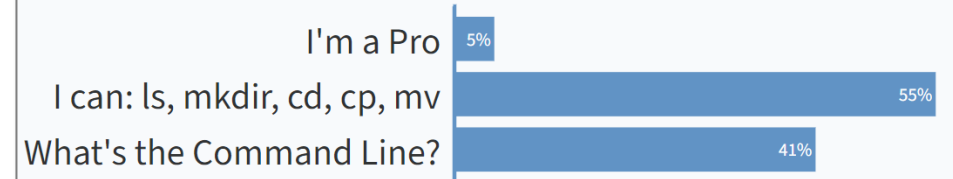
What is your Major?



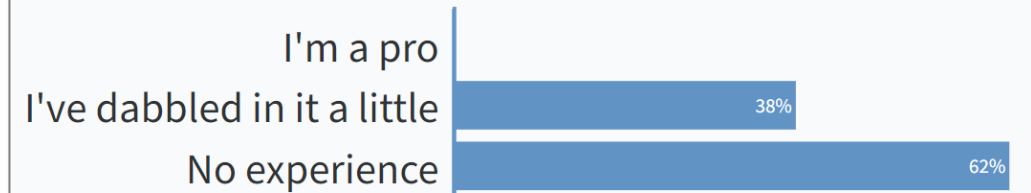
What Operating System will you be using?



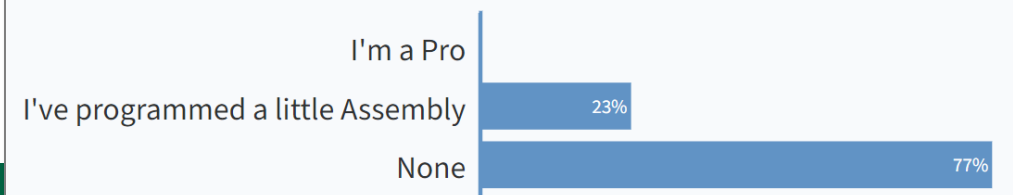
How much experience do you have working with the Command Line?



How much experience do you have with the C or C++ language (not C#)?



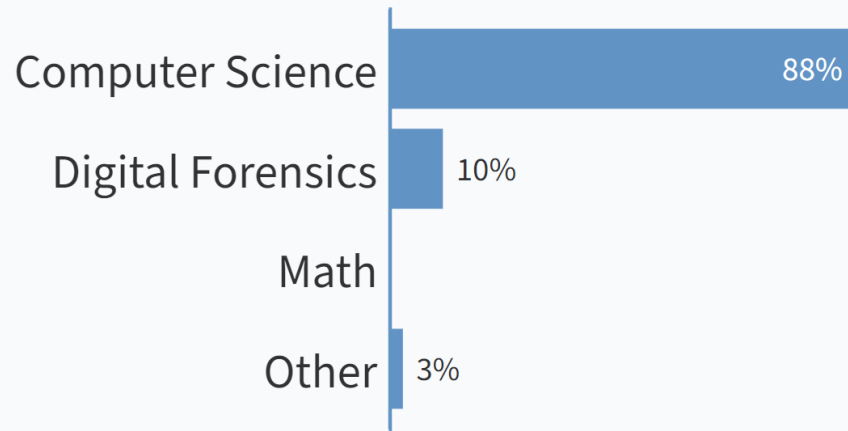
How much experience do you have with Assembly Language?



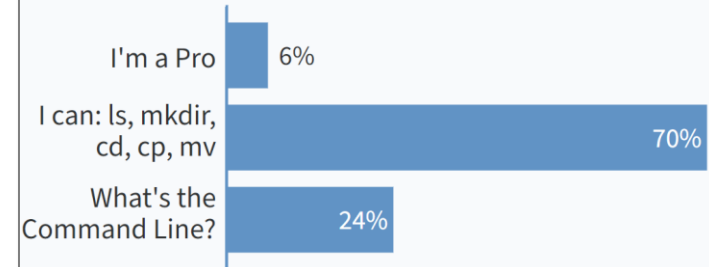
Survey Results (from last Lab) – Wed Lab

3

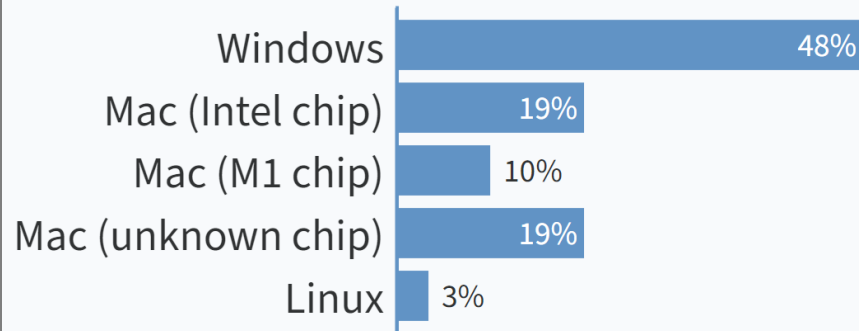
What is your Major?



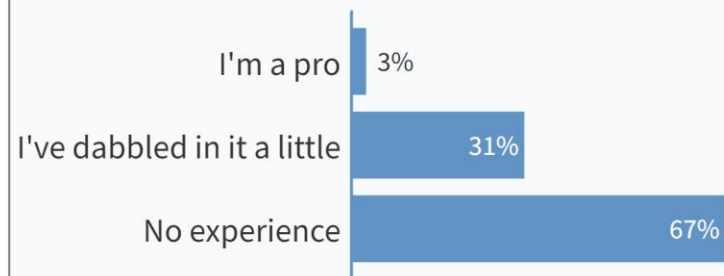
How much experience do you have working with the Command Line?



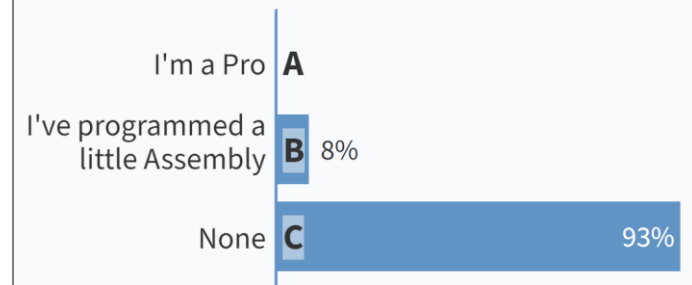
What Operating System will you be using?



How much experience do you have with the C or C++ language (not C#)?



How much experience do you have with Assembly Language?



Feedback from Lab 1 Assignment

- Great Job!
- Additional how-to info on Canvas:
 - VSCode Remote Development Config Video
 - Command Line Video (YouTube)
- If you had issues, or couldn't complete the assignment, please come see us !!
- Any questions?
- Going forward:
 - We'll ask for source code, no more binary
 - We'll ask for all the files to be placed in a .zip file
 - We'll ask for a Makefile

Today's Agenda

- Make
- Variables
- Looping, for
- Control Flow: if, else
- Functions



Make

Brief overview of Makefiles

- A Makefile is full of quick scripts, usually meant for the preparation of your program. For our purposes, we want to write down all the instructions we use to compile our C programs. Make will only compile those files that have changed.
- Upon running "make", the first group of instructions in the file will be executed.
- You can also add multiple groups of instructions for different purposes, and invoke them with "make [name]"
- A Makefile is made up of rules, each consisting of:
rule(target): prerequisites
 recipe (action)

Makefiles, continued

Here is an example Makefile, saved as Makefile, no extension:

```
M Makefile
1  build: hello.c
2      gcc -Wall -g hello.c -o hello -lm
3
4  run:
5      ./hello
```

Note: second line (recipe line) must start with a tab

`make` will run everything inside ‘build’, and `make run` will run everything inside ‘run’

Manual: <https://www.gnu.org/software/make/manual/>

Ref: https://ocw.mit.edu/courses/civil-and-environmental-engineering/1-124j-foundations-of-software-engineering-fall-2000/lecture-notes/makefile_primer/

Makefile on Canvas to use as a template

lab02 > **M** Makefile

```
1  FILE = file_name
2
3  build: $(FILE).c
4      # the next line is only needed if compiling outside Vulcan
5      #gcc -Wall -g $(FILE).c -o $(FILE) -lm -fno-pie -no-pie
6      # use this command to compile on Vulcan (enabled by default)
7      gcc -Wall -g $(FILE).c -o $(FILE) -lm
8
9  .PHONY: db
10
11 db:
12     gdb -tui $(FILE)
13
14 run:
15     ./$(FILE)
```

- `#`
 - a comment
- `FILE =`
 - Sets a variable for later use
 - Just replace `file_name` with the name of your file (no extension)
- `$(FILE)`
 - Is replaced with variable `FILE`

Must be saved as Makefile, capital M, no extension

More powerful Makefile (not required for this class)

10

```
2  CC = gcc
3  CFLAGS = -Wall -g # can also add -g to debug
4  DEPS = queue.h job_info.h Makefile
5  OBJS = sched.o queue.o
6  EXECS = sched
7
8  all: $(EXECS)
9
10 %.o: %.c $(DEPS)
11     $(CC) $(CFLAGS) -c -o $@ $<
12
13 sched: $(OBJS)
14     $(CC) $(CFLAGS) -o $@ $^ -lpthread
15
16
17 clean:
18     rm -i *.o $(EXECS)
```

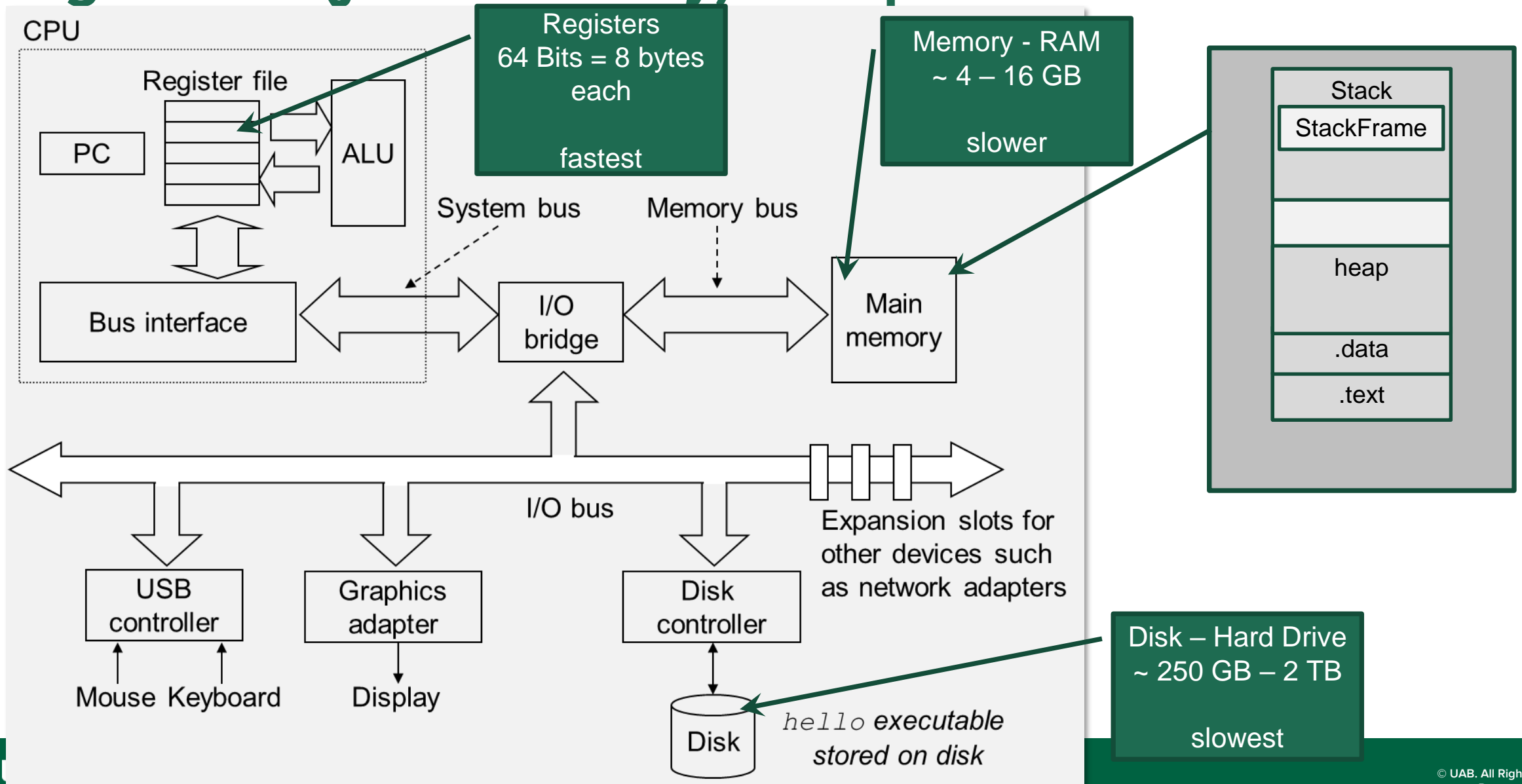
- Syntax:
 - <RULE: DEPENDENCY LINE>
 <tab><ACTION LINE>
 - Dependency line is: <target files:>
 [source files]
- % is pattern matching
 - See:
 <https://www.gnu.org/software/make/manual/make.html#Pattern-Rules>
- Automatic Variables
 - \$@ file name of the rule target
 - \$< for first prerequisite source file name
 - \$^ for all prerequisites separated by spaces
 - See:
 <https://www.gnu.org/software/make/manual/make.html#Automatic-Variables>

It's all about the memory

And our mental model of a Computer

High Level (just for today) Computer Architecture

12





C

The Basics

The ANSI Standard

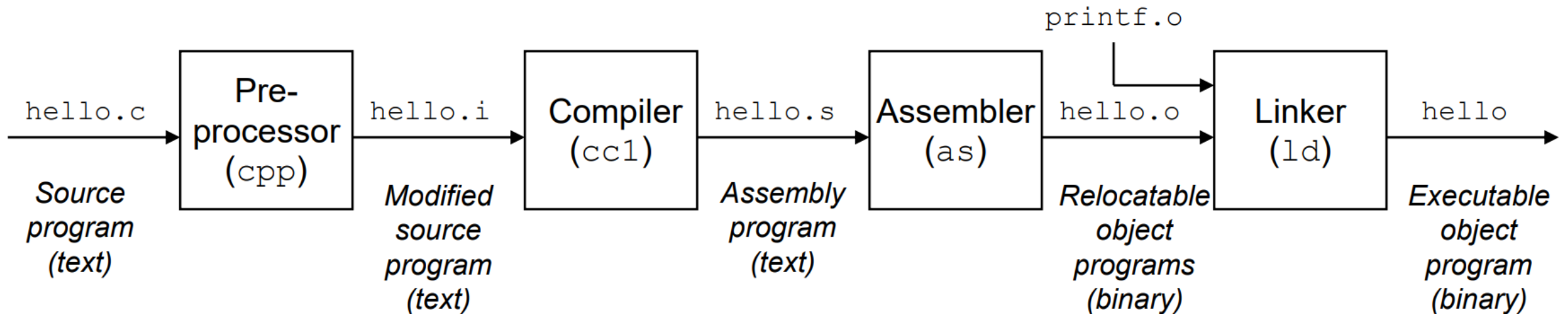
- The rapid expansion of the C language
 - Many companies developed their own C compilers
- In 1983, the American National Standard Institute (ANSI) began the development of the C standard that was completed and formally approved, in 1989, as ANSI C or Standard C
 - C89
 - C90
 - C95
 - C99
 - C11
 - C18

C (and Assembly) is a Compiled Language

When we type:

gcc hello.c -o hello

The code is compiled as follows:



Compilation System

Variables

C Primitive Types

TABLE 2.1 From: C from Theory to Practice

C Data Types

Data Type	Usual Size (bytes)	Range of Values (min-max)	Precision Digits
→ char	1	-128...127	
short int	2	-32.768...32.767	
→ int	4	-2.147.483.648...2.147.483.647	
long int	4	-2.147.483.648...2.147.483.647	
→ float	4	Lowest positive value: $1.17 \cdot 10^{-38}$ Highest positive value: $3.4 \cdot 10^{38}$	6
→ double	8	Lowest positive value: $2.2 \cdot 10^{-308}$ Highest positive value: $1.8 \cdot 10^{308}$	15
long double	8, 10, 12, 16		
unsigned char	1	0...255	
unsigned short int	2	0...65535	
unsigned int	4	0...4.294.967.295	
unsigned long int	4	0...4.294.967.295	

Variable Declaration and Storage Classes

18

Variable Declaration Syntax:
[Storage Class] <type> <name>;

Note: There is also a keyword: **const** (constant) Variable can be initialized, but not changed

Storage Class	Scope	Lifetime	Default Value
auto (default)	Same block	Until the block completes	Garbage Value
static	Same block	Until the program completes	0 (for int)
extern	Program	Until the program completes	0 (for int)
registers (fast, not guaranteed)	Same block	Until the block completes	Garbage Value

Scope: where variable can be used

Lifetime: how long the variable is in memory

Variable Names

- The name can contain letters, digits, and underscore characters _.
- The name must begin with either a letter or underscore
- C is case sensitive (distinguishes btw uppercase and lowercase)
- The following keywords cannot be used as variable names because they have special significance to the C compiler:

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

Arithmetic Conversions

```
char c;  
short s;  
int i;  
unsigned int u;  
float f;  
double d;  
long double ld;  
  
i = i+c; /* c is converted to int. */  
i = i+s; /* s is converted to int. */  
u = u+i; /* i is converted to unsigned int. */  
f = f+i; /* i is converted to float. */  
f = f+u; /* u is converted to float. */  
d = d+f; /* f is converted to double. */  
ld = ld+d; /* d is converted to long double. */
```

Arithmetic Operators

- + - / * %
- **int/int = cuts off the decimal part**

```
int a=7;
```

```
int b=5;
```

a/b will be equal to 1

also, be careful with the % operator

if ((n%2)==1) is dangerous**

if ((n%2)!=0) is safe

**** if n is odd and negative**

Comparisons

- `> >= < <= != ==`
- `if (a == 10)`

	Logical	Bitwise
NOT	!	\sim (one's complement)
AND	&&	&
OR		(inclusive OR) ^ (exclusive OR)
Leftwise bit shift		<<
Rightwise bit shift		>>

Operator Precedence

Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

```

2 static int x = 5;
3 extern int z;
4 int y = 5;
5 void myFunc(){
6     int i = 5;
7     printf("inside funct, i is %d (%p)\n", i, &i);
8     return;
9 }
10 void incrementAuto(){
11     auto int i = 0; // scope block, lifetime block
12     i++;
13     printf("inside incrementAuto, i is %d (%p)\n", i, &i);
14     return;
15 }
16 void incrementStatic(){
17     static int i = 0; // scope block, lifetime program
18     i++;
19     printf("inside incrementStatic, i is %d (%p) .data\n", i, &i);
20     return;
21 }
22 int main(){
23     int i = 10;
24     printf("initial value of i is %d (%p)\n", i, &i);
25     /* this block is just for demo, we wouldn't do this ir1 */
26     {
27         int i = 15;
28         printf("inside block, i is %d (%p)\n", i, &i);
29     }
30     printf("outside block, i is %d (%p)\n", i, &i);
31     myFunc();
32     incrementAuto();
33     incrementAuto();
34     incrementStatic();
35     incrementStatic();
36     printf("static x is %d (%p) .data\n", x, &x); // x declare global, so it's in .data
37     extern int y;
38     //int y = 5;
39     printf("extern y is %d (%p) .data\n", y, &y); // y declare global, so it's in .data
40     int z = 10;

```

Vars.c

24

```

initial value of i is 10 (0x7ffff4b4f250)
inside block, i is 15 (0x7ffff4b4f254)
outside block, i is 10 (0x7ffff4b4f250)
inside funct, i is 5 (0x7ffff4b4f234)
inside incrementAuto, i is 1 (0x7ffff4b4f234)
inside incrementAuto, i is 1 (0x7ffff4b4f234)
inside incrementStatic, i is 1 (0x7ff96ba0101c) .data
inside incrementStatic, i is 2 (0x7ff96ba0101c) .data
static x is 5 (0x7ff96ba01010) .data
extern y is 5 (0x7ff96ba01014) .data
extern z is 10 (0x7ffff4b4f254) stack

```

<https://youtu.be/hxh8cORcerM>

printf

printf

printf Syntax: % conversion specifier
% [flags] [min field width] [precision] [length] <conversion type>

Flag	Description
'	(apostrophe) format integer with thousands grouping characters
-	left-justify the output in the field
+	always display sign of a signed conversion
(space)	prefix by a space if no sign is generated
#	convert using alternative form (include 0x prefix for hexadecimal format, for example)
0	prefix with leading zeros instead of padding with spaces

Figure 5.7 The flags component of a conversion specification

Precision:
e.g. .3 is three digits after the decimal
Default is 6 digits, if precision is 0 no decimal appears (f)

Length modifier	Description
hh	signed or unsigned char
h	signed or unsigned short
l	signed or unsigned long or wide character
ll	signed or unsigned long long
j	intmax_t or uintmax_t
z	size_t
t	ptrdiff_t
L	long double

Figure 5.8 The length modifier component of a conversion specification

Conversion Type	Description
d	signed decimal
f	double floating-point number
c	character
s	string
p	pointer
x, X	unsigned hexadecimal
%	% character
o	unsigned octal
u	unsigned decimal
a, A	double floating-point number in hexadecimal exponential format
e, E	double floating-point number in exponential format

Use “\” as escape character. e.g. “\n” newline, “\t” tab, “\\” is backslash

For more info, see man page: <https://linux.die.net/man/3/printf>
Also: http://www.pixelbeat.org/programming/gcc/format_specs.html

Looping – for loop

```
int i;
```

```
// NOTE: on Vulcan, must declare the variable outside for loop
```

```
for(i = 0; i < 5; i++) {  
    // do stuff  
}
```

Control Flow – if-else

```
if(expression1){  
    // do stuff if expression1 evaluates to true  
} else if (expression2) {  
    // do other stuff if expression2 evaluates to true  
  
} else {  
    // do stuff if all other expressions are false  
}
```

Functions

Exercise – Jumping Jack Person

30

- Write a C program to generate the image (ASCII) of a person doing 10 jumping jacks
 - Each individual image should be a function
- To clear the screen:
 - `system("clear");`
 - Don't forget to `#include<stdlib.h>`
- To pause, you can use:
 - `sleep([seconds])`
 - `usleep([milliseconds])`
 - Don't forget to `#include<unistd.h>`
 - Note: if you `printf` before you sleep you need to clear the buffer with a newline `"\n"` or `fflush(stdout);`

