COMSW 1003-1

Introduction to Computer Programming in C

Lecture 1

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Course Information - Goals

“A general introduction to computer science concepts, algorithmic problem-solving capabilities, and programming skills in C”

University bulletin

- Learn how to program, in C
- Understand basic Computer Science problems
- Learn about basic data structures
- Start to think as a computer scientist
- **Use all of the above to solve real world problems**

Course Information - Instructor

- Michele Merler
  - Email: mmerler@cs.columbia.edu or mm3233@columbia.edu
  - Office : 624 CEPSR
  - Office Hours: Friday 12pm-2pm

- 4th year PhD Student in CS Department

- Research Interests:
  - Image & Video Processing
  - Multimedia
  - Computer Vision

Course Information- TA

- TDB
  - Email: TDB@columbia.edu
  - Office : TA room
  - Office Hours: TDB

Course Information- Courseworks

We will be using Courseworks (https://courseworks.columbia.edu/) for:

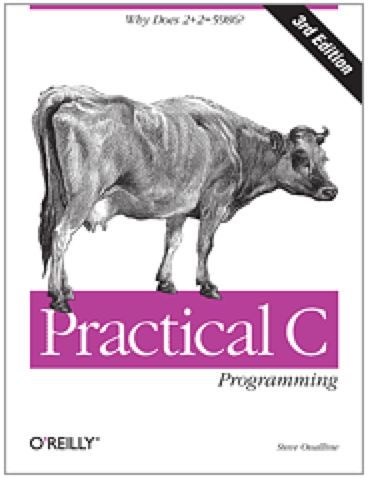
- Message board for discussions
- Submit Homeworks
- Grades

Check out the board before you send an email to the instructor or the TA, the answer you are looking for could already be there!

Course Information Requirements and Books

Requirements

- Basic computer skills
- CUNIX account



Textbooks

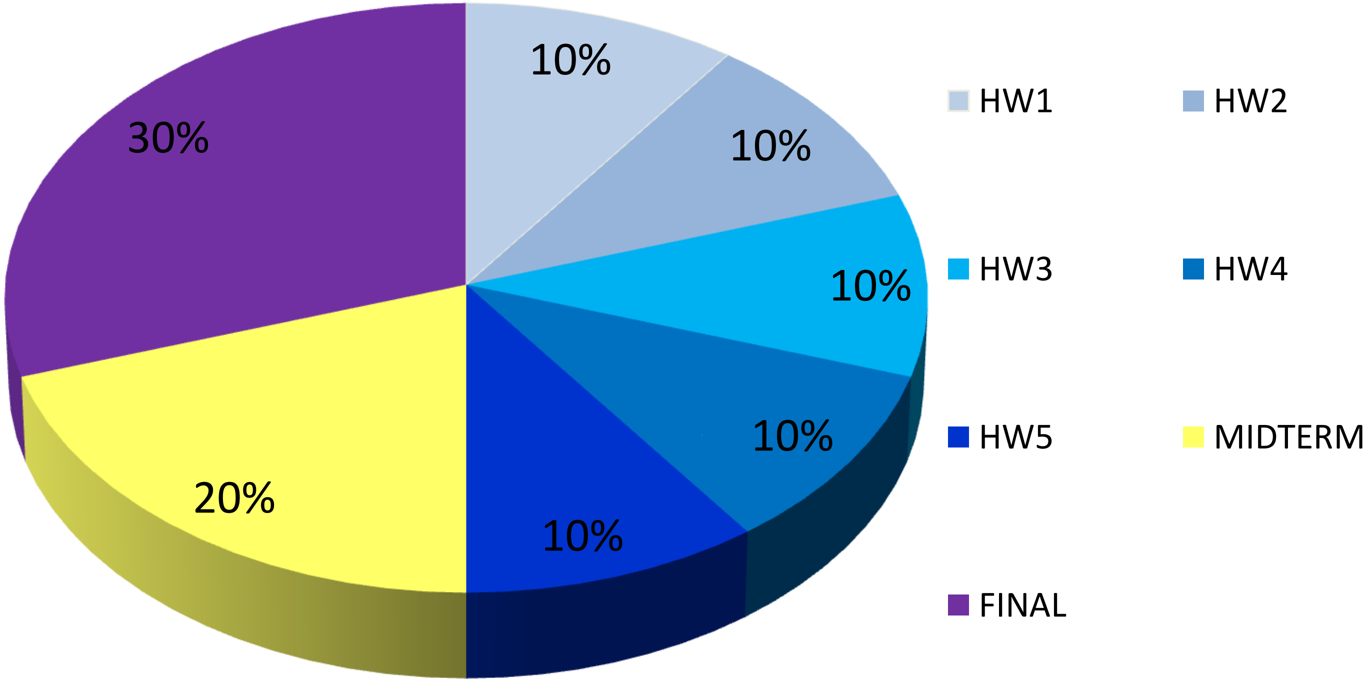
- **The C Programming Language (2nd Edition)** by Brian Kernighan and Dennis Ritchie

  http://www1.cs.columbia.edu/~mmerler/coms1003-1/C Programming Language.rar

- **Practical C Programming (3rd Edition)** by Steve Oualline

Course Information - Grading

- 5 Homeworks (10%, 10%, 10% , 10% , 10%)
- Midterm Exam (20%)
- Final Exam (30%)



Course Information
Academic Honesty

It’s quite simple:

- Do not copy from others
- Do not let others copy from you

Do your homework **individually**

Please read through the department’s policies on academic honesty
http://www.cs.columbia.edu/education/honesty/

Course Information - Syllabus

Go to class webpage

http://www1.cs.columbia.edu/~mmerler/coms1003-1_files/Syllabus.html

What is Computer Science?

Computer science (sometimes abbreviated **CS**) is the study of the theoretical foundations of **information** and **computation**, and of practical techniques for their implementation and application in computer systems

Wikipedia

"Computer science and engineering is the systematic study of algorithmic processes-their theory, analysis, design, efficiency, implementation, and application-that describe and transform information"

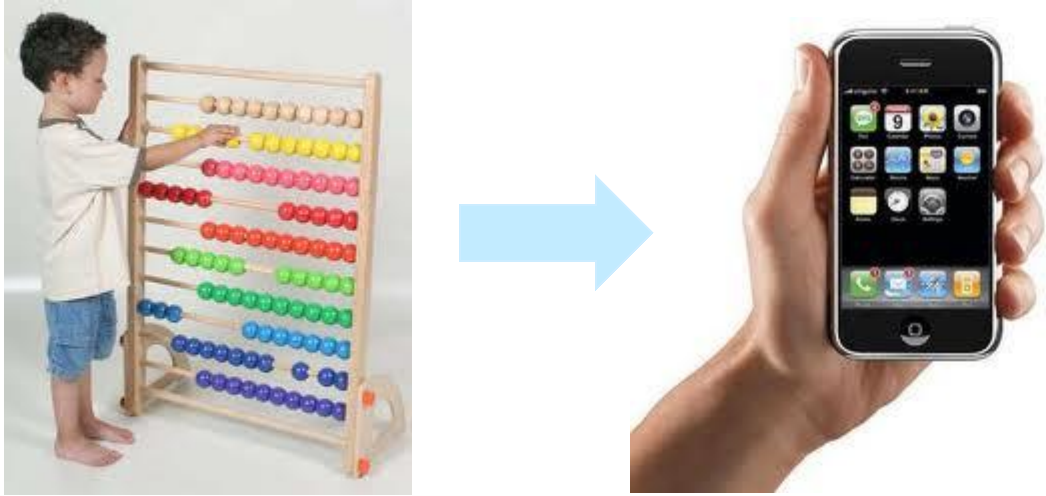
Comer, D. E.; Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., and Young, P. R. (Jan. 1989). "Computing as a discipline". *Communications of the ACM* **32** (1): 9.

"Computer science is the study of information structures"

Wegner, P. (October 13–15, 1976). "Research paradigms in computer science". *Proceedings of the 2nd international Conference on Software Engineering*. San Francisco, California, United States

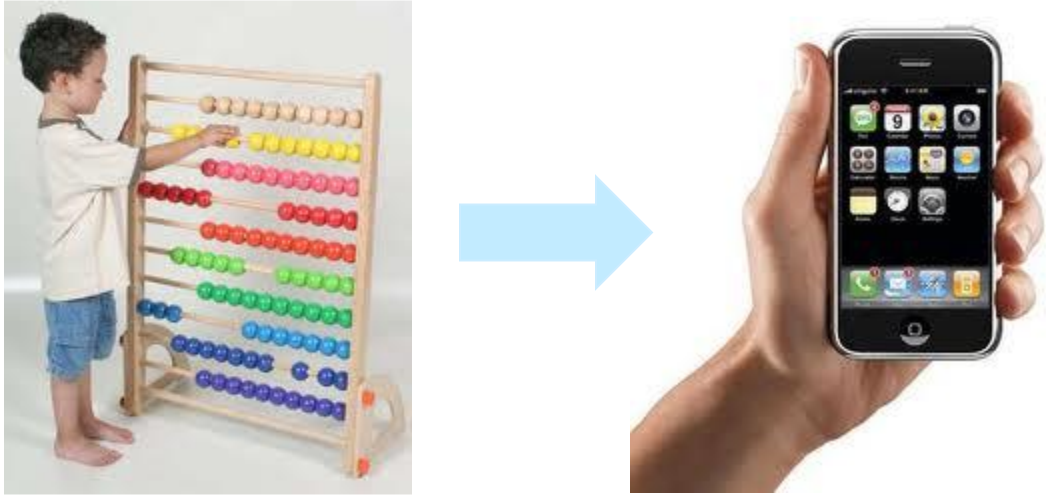
"*Computer Science is the study of all aspects of computer systems, from the theoretical foundations to the very practical aspects of managing large software projects.*"

Massey University



What is Computer Science?

Computer Science is the discipline that studies how to make computers perform tasks that are too complex or boring for humans



Computer Science Areas

Computational science


Numerical analysis

Computational physics

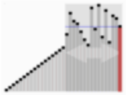

Computational chemistry


Bioinformatics

Algorithms and data structures

$O(n^2)$

Analysis of algorithms


Algorithms

Data structures

Artificial Intelligence


Machine Learning


Computer vision

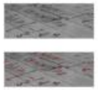
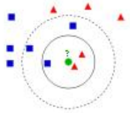
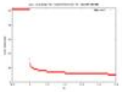


Image Processing

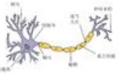

Pattern Recognition


Cognitive Science


Data Mining


Evolutionary Computation


Information Retrieval


Knowledge Representation

abcdefghijk
lmnopqrstuvwxyz

ABCDEFGHIJKLMN
OPQRSTUVWXYZ

Natural Language Processing

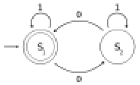

Robotics

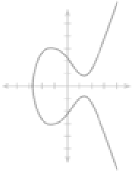

Human–computer interaction

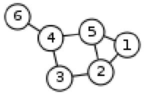
Theoretical computer science

$P \to Q$

Mathematical logic


Automata theory


Number theory



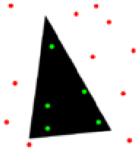
Graph theory

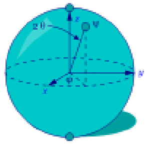
$\Gamma \vdash x : Int$

Type theory

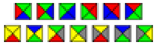
$$\begin{tikzcd} X \arrow[r, "f"] \arrow[dr, "g \circ f"'] & Y \arrow[d, "g"] \\ & Z \end{tikzcd}$$

Category theory


Computational geometry


Quantum computing theory

Theory of computation


Computability theory

P = NP ?

Computational complexity theory

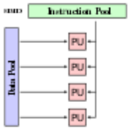
GNITIRW-TERCES

Cryptography

Computer architecture

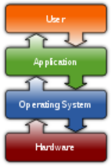

Digital logic


Microarchitecture



Multiprocessing

Software Engineering



Operating systems


Computer networks


Databases


Computer security


Ubiquitous computing

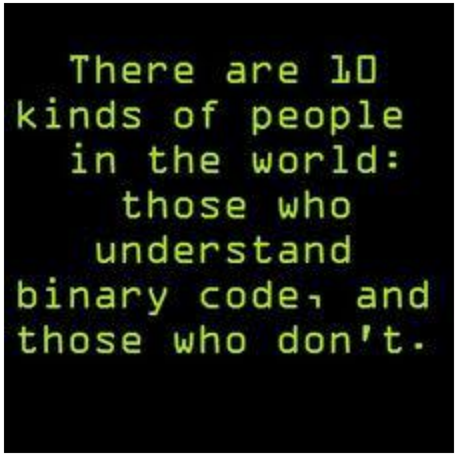

Systems architecture


Compiler design


Programming languages

Why programming?

- We need a way to tell computers what to do
- It would be nice to communicate with computers in English, but…
  - English can be ambiguous!
  - Computers only understand binary!
- Solution: programming languages



There are 10
kinds of people
in the world:
those who
understand
binary code, and
those who don't.

What is a Program?

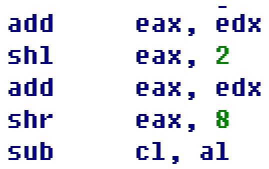
- A **Program** is a sequence of instructions and computations
- We'll be designing programs in this course.
- These programs will be based on **algorithms**
- An **Algorithm** is a step-by-step problem-solving procedure

Example

- Add 3 large numbers
    - 453 + 782 + 17,892
- Hard to do all at once
    - Solution: “divide and impera”!
    - (453 + 782) + 17,892 =
    - 1,235 + 17,892 = 19,127
- Algorithms help us divide and organize complex problems into sub-problems which are easier to solve (bottom-up approach)

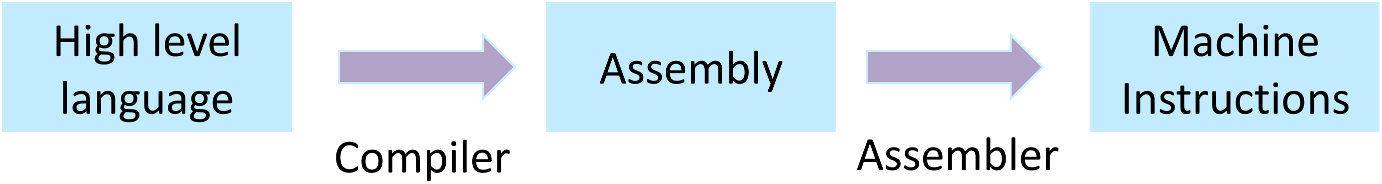
Programming

- Back in the day, programmers wrote in **Assembly**, a language where each word stands for a single instruction



```
add     eax, edx
shl     eax, 2
add     eax, edx
shr     eax, 8
sub     cl, al
```

- But then they had to **hand translate** each instruction into binary!!!
- Solution: the **assembler**, a computer program to do the translation
- From then, programmers could worry only about writing assembly code
- Then they started to devise higher level languages (FORTRAN, COBOL, PASCAL, C, C++, JAVA, Perl, Python, etc.), which get translated into Assembly by **compilers** (we will use **GCC**, a C compiler for Unix)



High level language → Compiler → Assembly → Assembler → Machine Instructions

What is C?

- Programming language developed by Dennis Ritchie in 1972 at AT&T Bell labs

- Why is it named “C”?
  Well… the B programming language already existed !

- C is still the most used programming language for Operating Systems

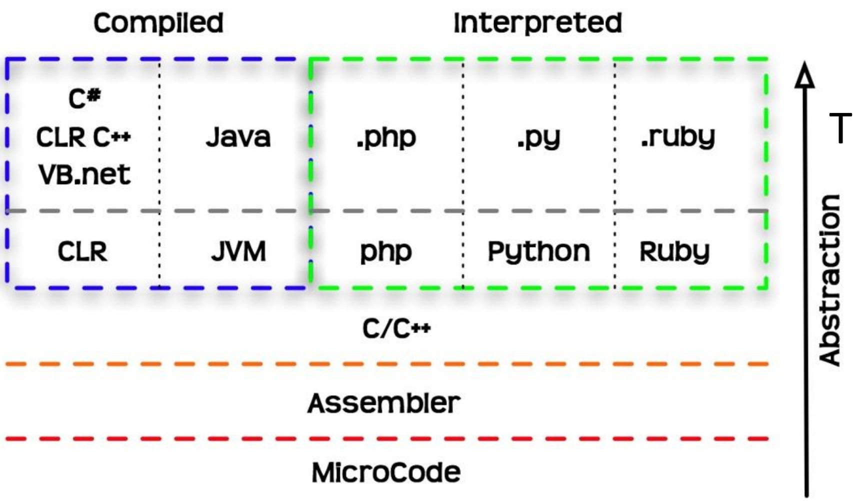
- Popular because:
  - Flexible
  - C compiler was widely available

- Basis for other popular programming languages: C++, C#

What is C?

- Among the “high level” programming languages, C is one with the lowest level of abstraction
- Close to English, but more precise!
- Easy to compile into Assembly => Fast
- Rich set of standard function = we don’t have to implement everything from scratch!

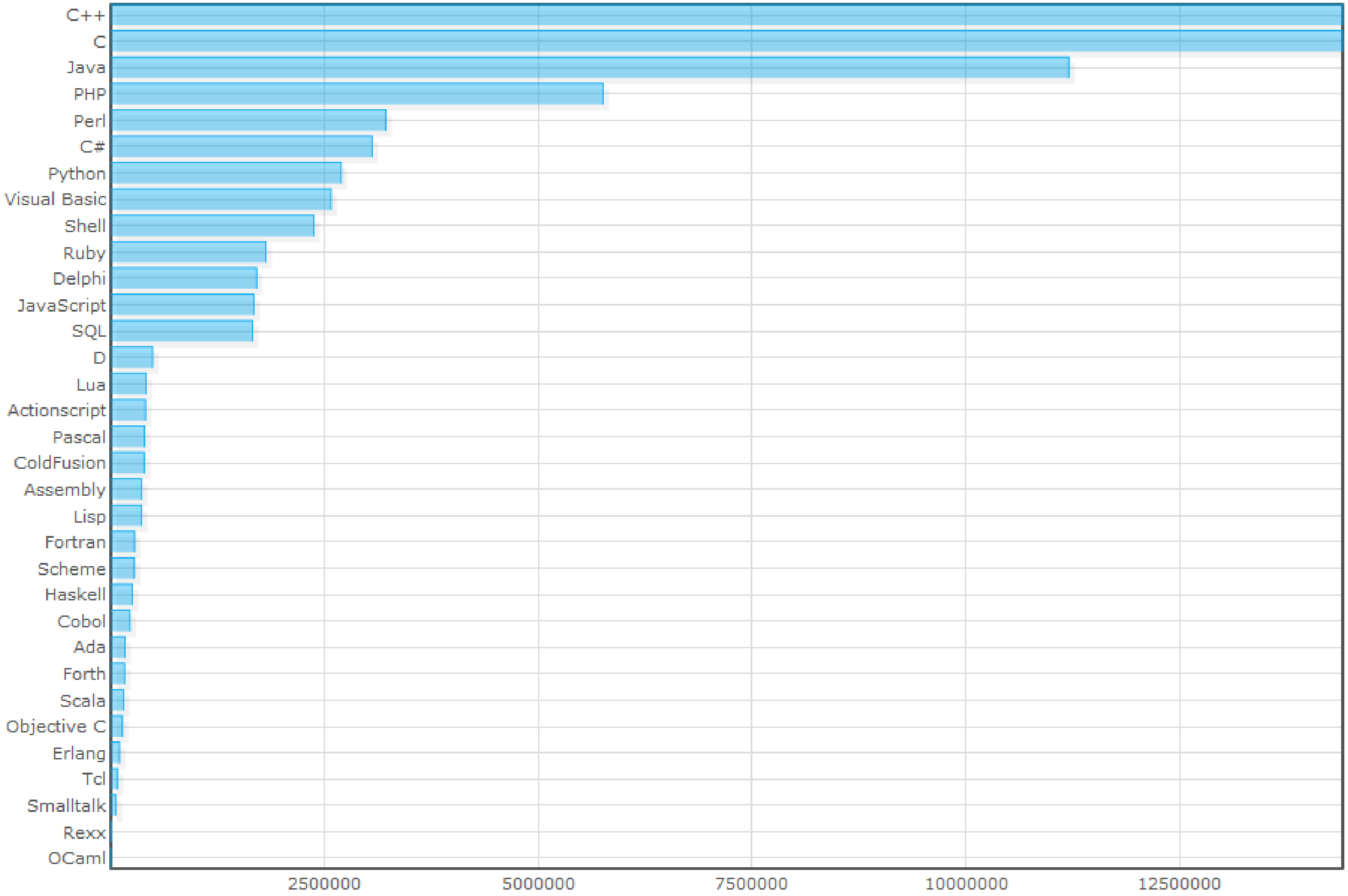
Translated before run time



Translated at run time

Why C? Interesting Facts …

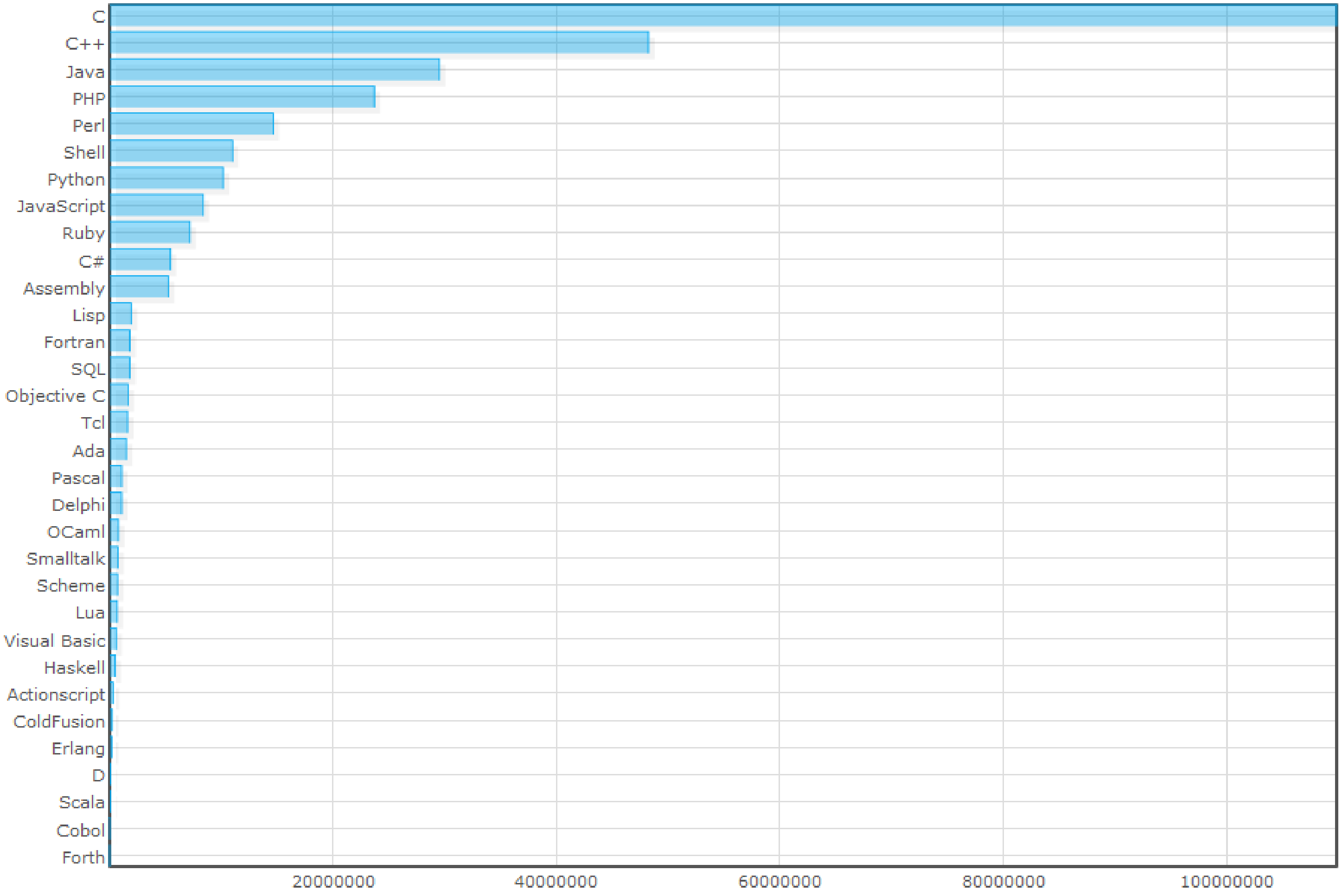
Approximation of **popularity** of language using Yahoo API http://www.langpop.com/



Slide credit: Priyank Singh

Why C? Interesting Facts …

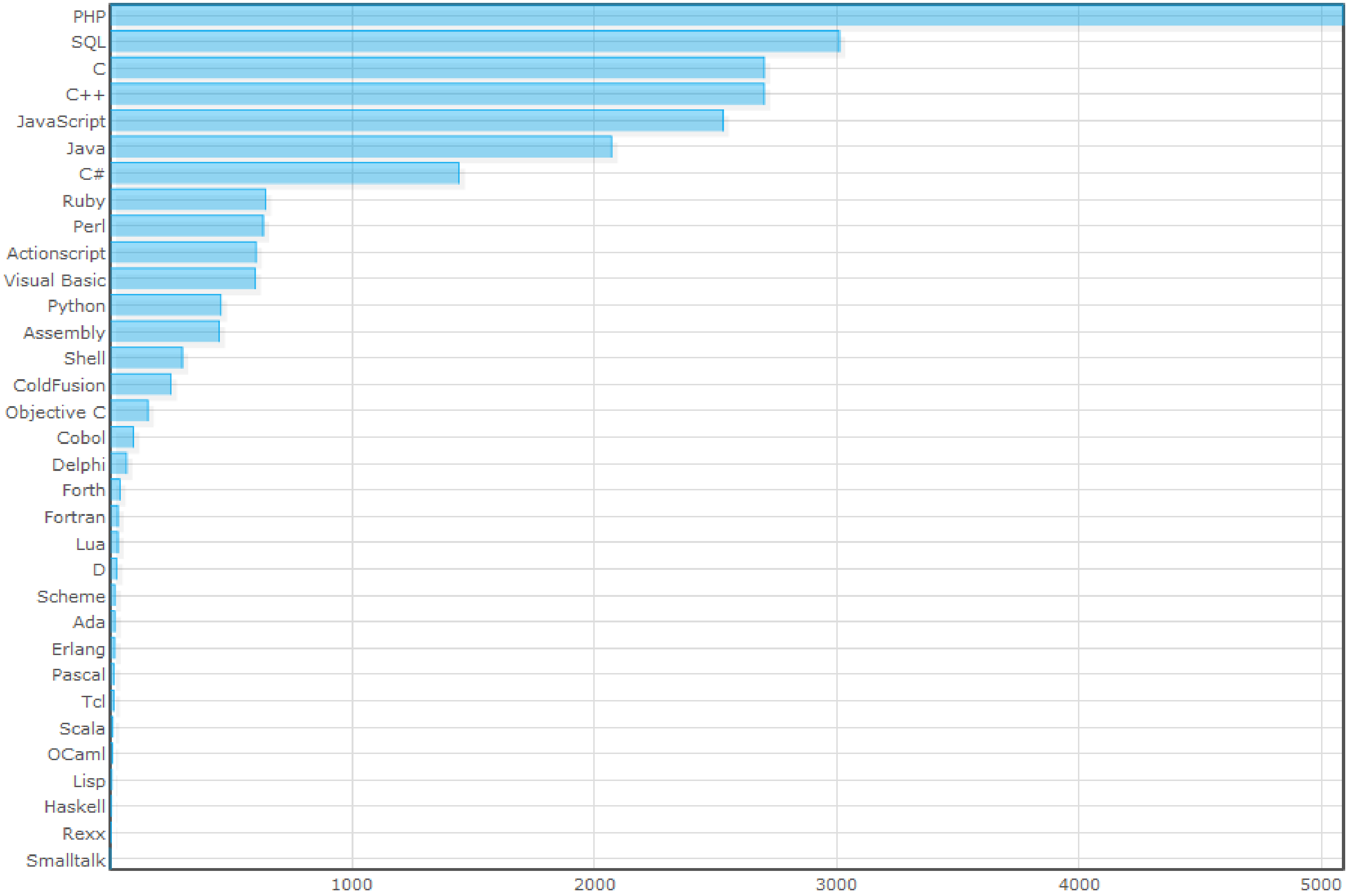
Available language **code** available using Google code search http://www.langpop.com/



Slide credit: Priyank Singh

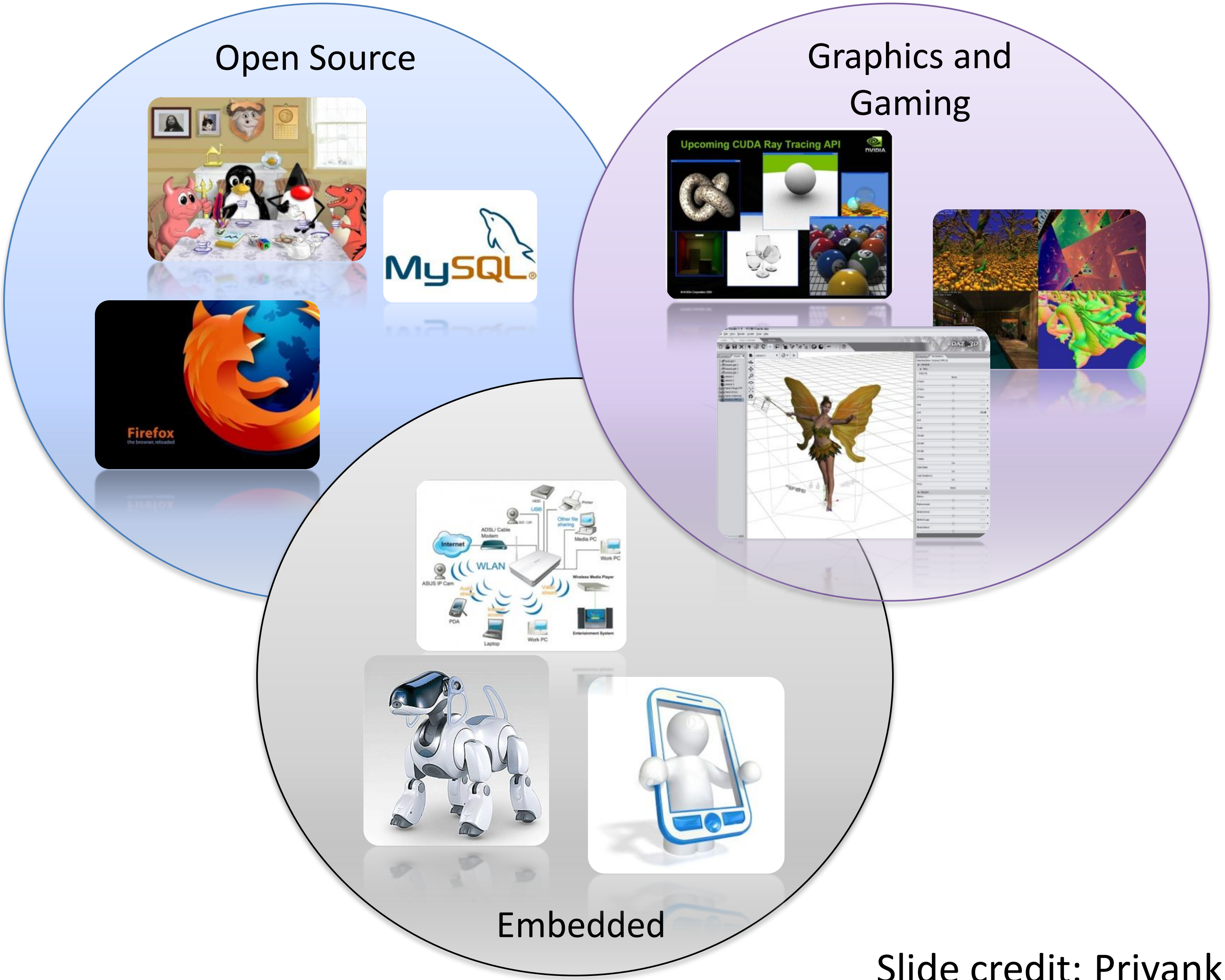
Why C? Interesting Facts …

Jobs posting on craiglist.org, from website http://www.langpop.com/



Slide credit: Priyank Singh

C/C++ Industry



Open Source

Graphics and Gaming

Embedded

Slide credit: Priyank Singh

Example of C program

Hello world!

Announcements

- Homework 0 is out! Due at the beginning of next class

- Bring your laptop to class

COMSW 1003-1

Introduction to Computer Programming in C

Lecture 2

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Announcements

- Exercise1 is out

- We have a TA!

  Gaurav Agarwal
  - MS student in CS department
  - **Email**: ga2310@columbia.edu
  - **Office Hours**: Tuesday 11am-12pm in Mudd 122A (TA room)

What is a Program?

- A **Program** is a sequence of instructions and computations
- We'll be designing programs in this course.
- These programs will be based on **algorithms**
- An **Algorithm** is a step-by-step problem-solving procedure

Example

- Add 3 large numbers
    - 453 + 782 + 17,892
- Hard to do all at once
    - Solution: “divide and impera”!
    - (453 + 782) + 17,892 =
    - 1,235 + 17,892 = 19,127
- Algorithms help us divide and organize complex problems into sub-problems which are easier to solve (bottom-up approach)

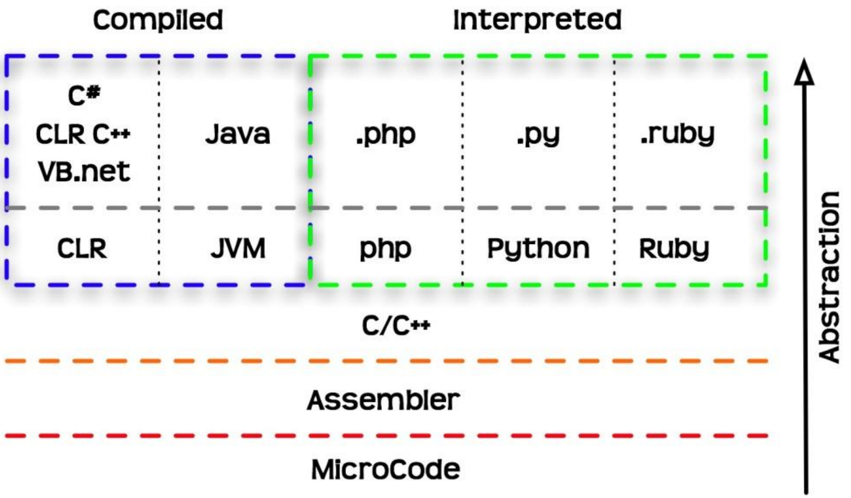
What is C?

- Programming language developed by Dennis Ritchie in 1972 at AT&T Bell labs

- Why is it named “C”?
  Well… the B programming language already existed !

- C is one of the high level programming language with the lowest level of abstraction

- Low to be close to assembly and machine language ⟶ fast!
- High to be programmable by humans without (too many) headaches


Compiled

Interpreted

C#
CLR C++
VB.net

Java

.php

.py

.ruby

CLR

JVM

php

Python

Ruby

C/C++

Assembler

MicroCode

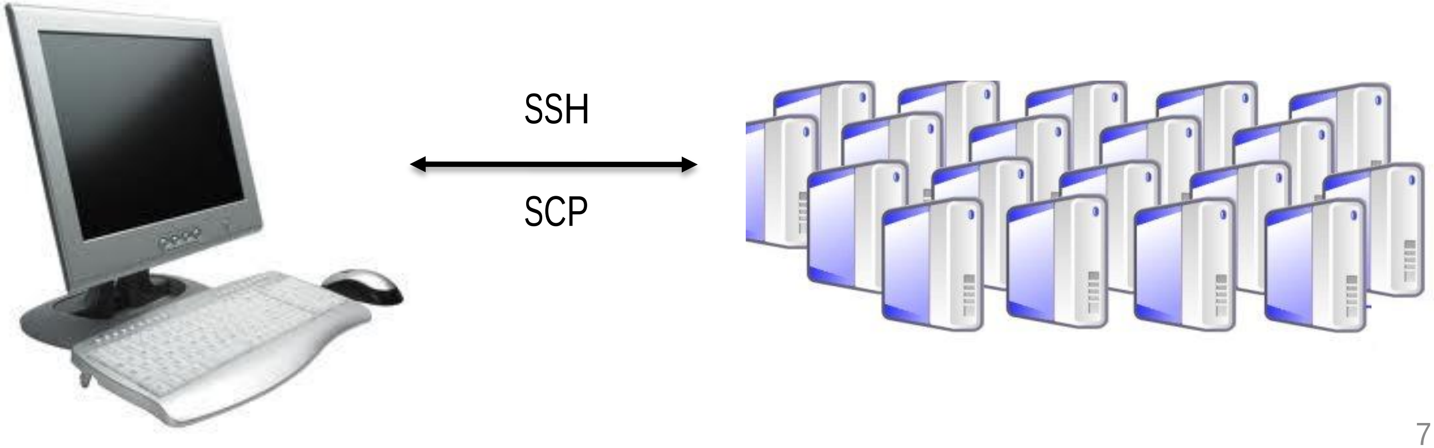
Abstraction

CUNIX

- CUNIX refers to the Columbia Unix environment
- For you: place where you develop your programs!

Accessing CUNIX remotely

- **Secure Shell** or **SSH** is a **network protocol** that allows data to be exchanged using a **secure** channel between two networked devices
- The **SCP** protocol is a **network protocol** that supports **file transfers**



Code Developing Tools – Linux and Mac

- Open terminal

- SSH to cunix.cc.columbia.edu

```
ssh yourUNI@cunix.cc.columbia.edu
```

- Data transfer: scp or get/put
  - Copying file to host:

```
scp SourceFile user@host:directory/TargetFile
```

  - Copying file from host:

```
scp user@host:/directory/SourceFile TargetFile
```

For MAC: use FUGU (graphical data transfer tool)
http://www.columbia.edu/acis/software/fugu/
http://download.cnet.com/Fugu/3000-2155_4-26526.html

Code Developing Tools – Linux and Mac

To use windowing environment:

Mac users need only start **X11** (found in the Utilities folder) and log in to the X11 terminal like this:

```
ssh -X username@cunix.cc.columbia.edu
```

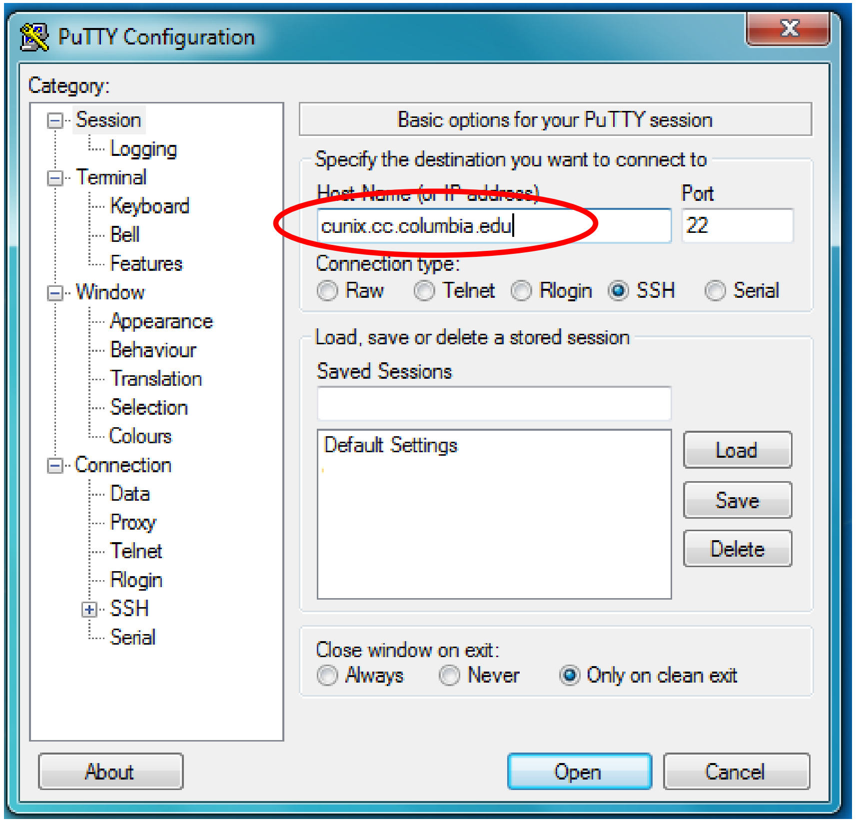
- Linux users: see X-Windows section in CUNIX tutorial

Code Developing Tools - Windows

- Xming and Putty to SSH and visualization
  - http://sourceforge.net/projects/xming/
  - http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html
- WinSCP for data transfer
  - http://winscp.net/eng/download.php#download2
- Notepad++ for editing (can be used in combination with WinSCP)
  - http://notepad-plus-plus.org/

Code Developing Tools - Windows

- Launch Xming
- Open a session in putty with Host Name
  - cunix.cc.columbia.edu



PuTTY Configuration

Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): cunix.cc.columbia.edu

Port: 22

Connection type:
○ Raw ○ Telnet ○ Rlogin ● SSH ○ Serial

Load, save or delete a stored session

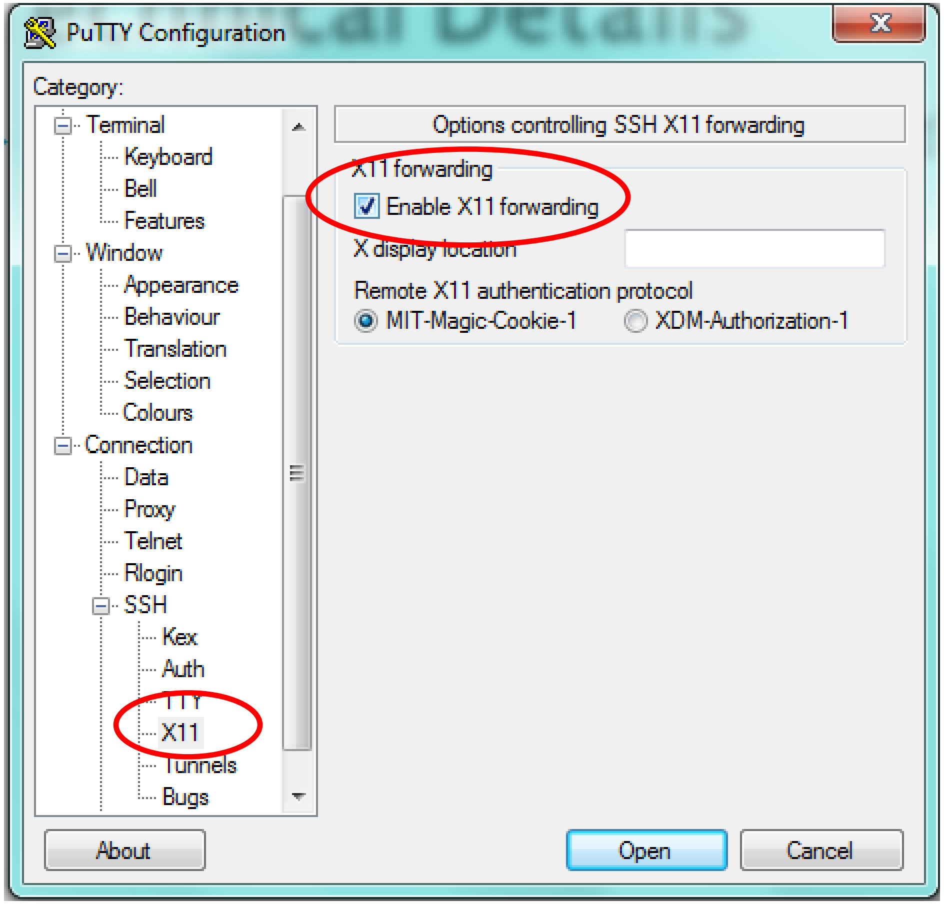
Saved Sessions

Default Settings

Close window on exit:
○ Always ○ Never ● Only on clean exit

Code Developing Tools - Windows

- Make sure the X11 option of the SSH category is enabled



Options controlling SSH X11 forwarding

X11 forwarding

☑ Enable X11 forwarding

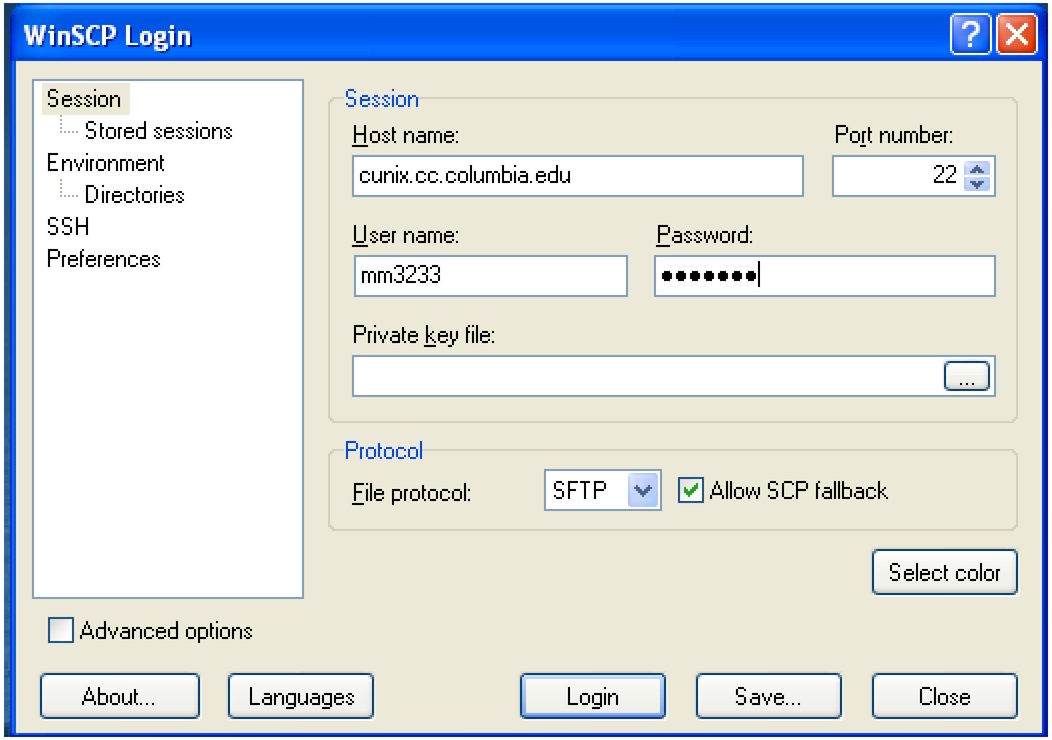
X display location

Remote X11 authentication protocol

● MIT-Magic-Cookie-1 ○ XDM-Authorization-1

Code Developing Tools - Windows

- Use WinScp to transfer files



Session

Host name: cunix.cc.columbia.edu

Port number: 22

User name: mm3233

Password: ●●●●●●●●

Private key file:

Protocol

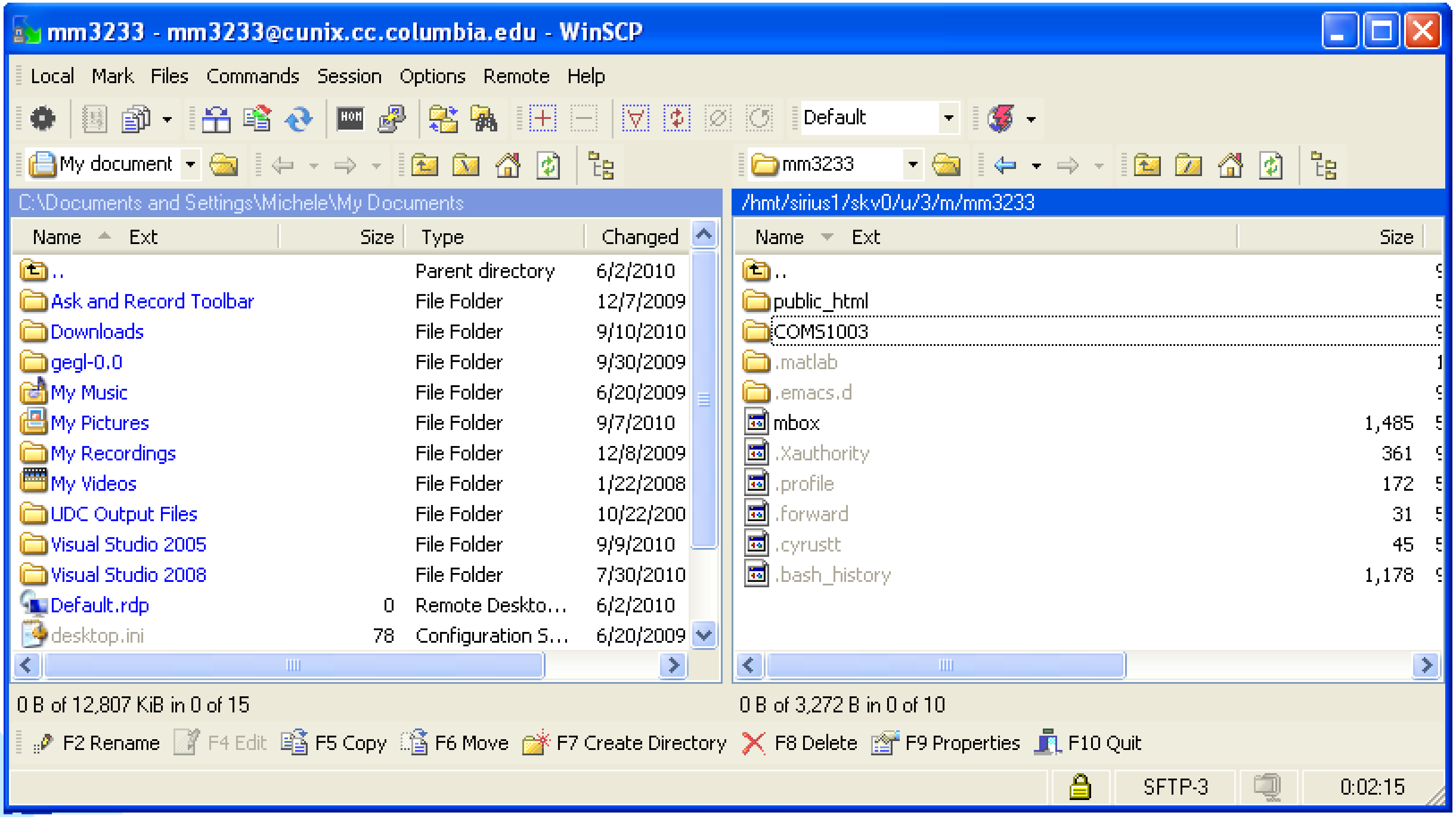
File protocol: SFTP

☑ Allow SCP fallback

☐ Advanced options

Code Developing Tools - Windows

- Use WinScp to transfer files



C:\Documents and Settings\Michele\My Documents

Name	Ext	Size	Type	Changed
..			Parent directory	6/2/2010
Ask and Record Toolbar			File Folder	12/7/2009
Downloads			File Folder	9/10/2010
gegl-0.0			File Folder	9/30/2009
My Music			File Folder	6/20/2009
My Pictures			File Folder	9/7/2010
My Recordings			File Folder	12/8/2009
My Videos			File Folder	1/22/2008
UDC Output Files			File Folder	10/22/200
Visual Studio 2005			File Folder	9/9/2010
Visual Studio 2008			File Folder	7/30/2010
Default.rdp		0	Remote Deskto…	6/2/2010
desktop.ini		78	Configuration S…	6/20/2009

0 B of 12,807 KiB in 0 of 15

/hmt/sirius1/skv0/u/3/m/mm3233

Name	Ext	Size
..		
public_html		
COMS1003		
.matlab		
.emacs.d		
mbox		1,485
.Xauthority		361
.profile		172
.forward		31
.cyrustt		45
.bash_history		1,178

0 B of 3,272 B in 0 of 10

Code Developing Environment

CUNIX Tutorial

Compiling your C code

- GCC : **GNU Compiler Collection**
- When you invoke GCC, it normally does preprocessing, compilation, assembly and linking
  - Basic Command
    - `gcc myProgram.c`
    - `./a.out`

Run compiled program (executable)

  - More advanced options
    - `gcc –Wall –o myProgram myProgram.c`
    - `./myProgram`

Compiling your C code

- GCC : **GNU Compiler Collection**
- When you invoke GCC, it normally does preprocessing, compilation, assembly and linking
  - Basic Command
    - gcc myProgram.c
    - ./a.out

Run compiled program (executable)

Display all types of warnings, not only errors

Specify name of the executable

- gcc –Wall –o myProgram myProgram.c
- ./myProgram

Run compiled program (executable)

Assignment

- Read PCP Ch 1
- Read PCP Ch 2, pages 11 to 15, 33

COMSW 1003-1

Introduction to Computer Programming in C

Lecture 3

Spring 2011

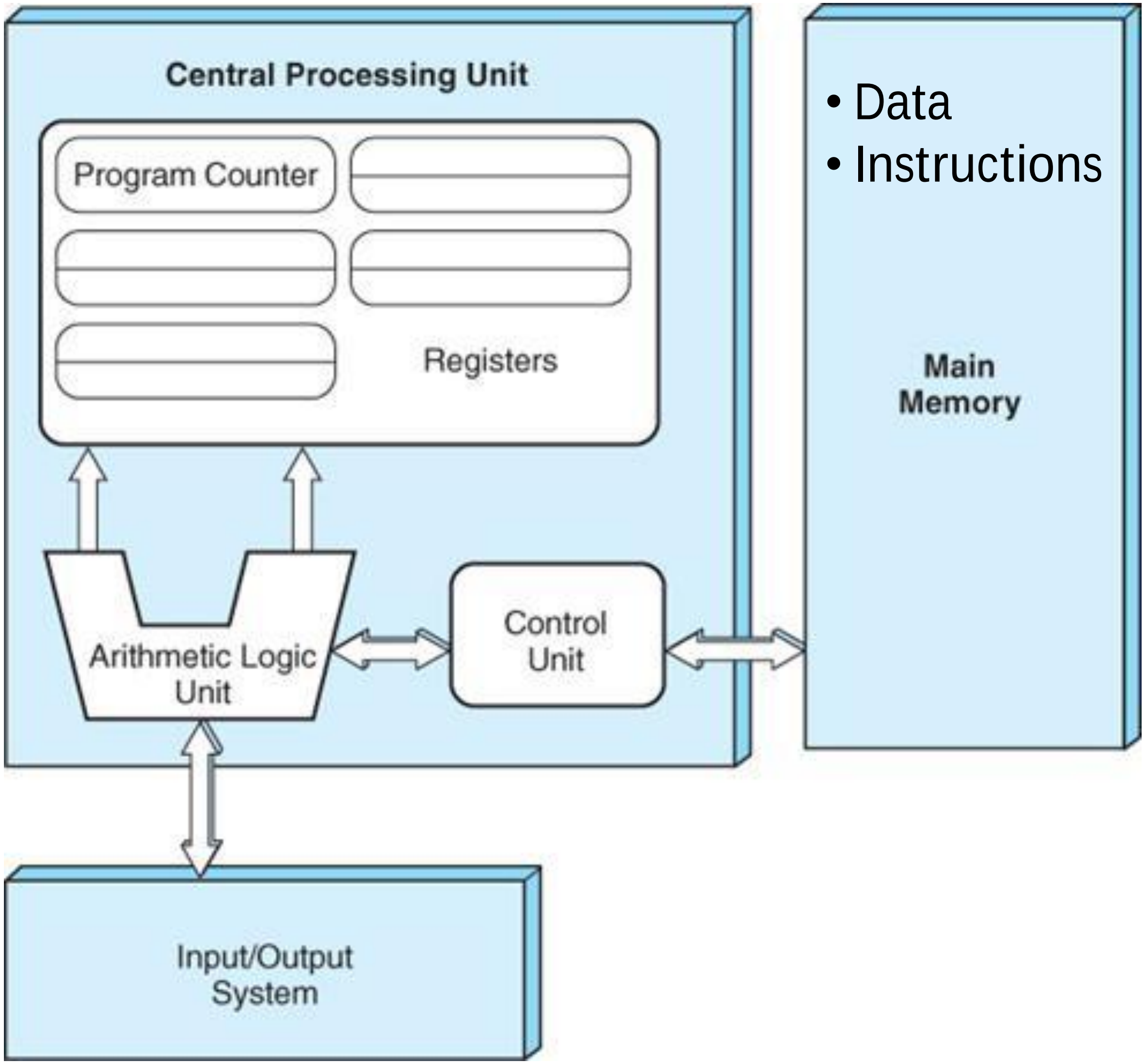
Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Today

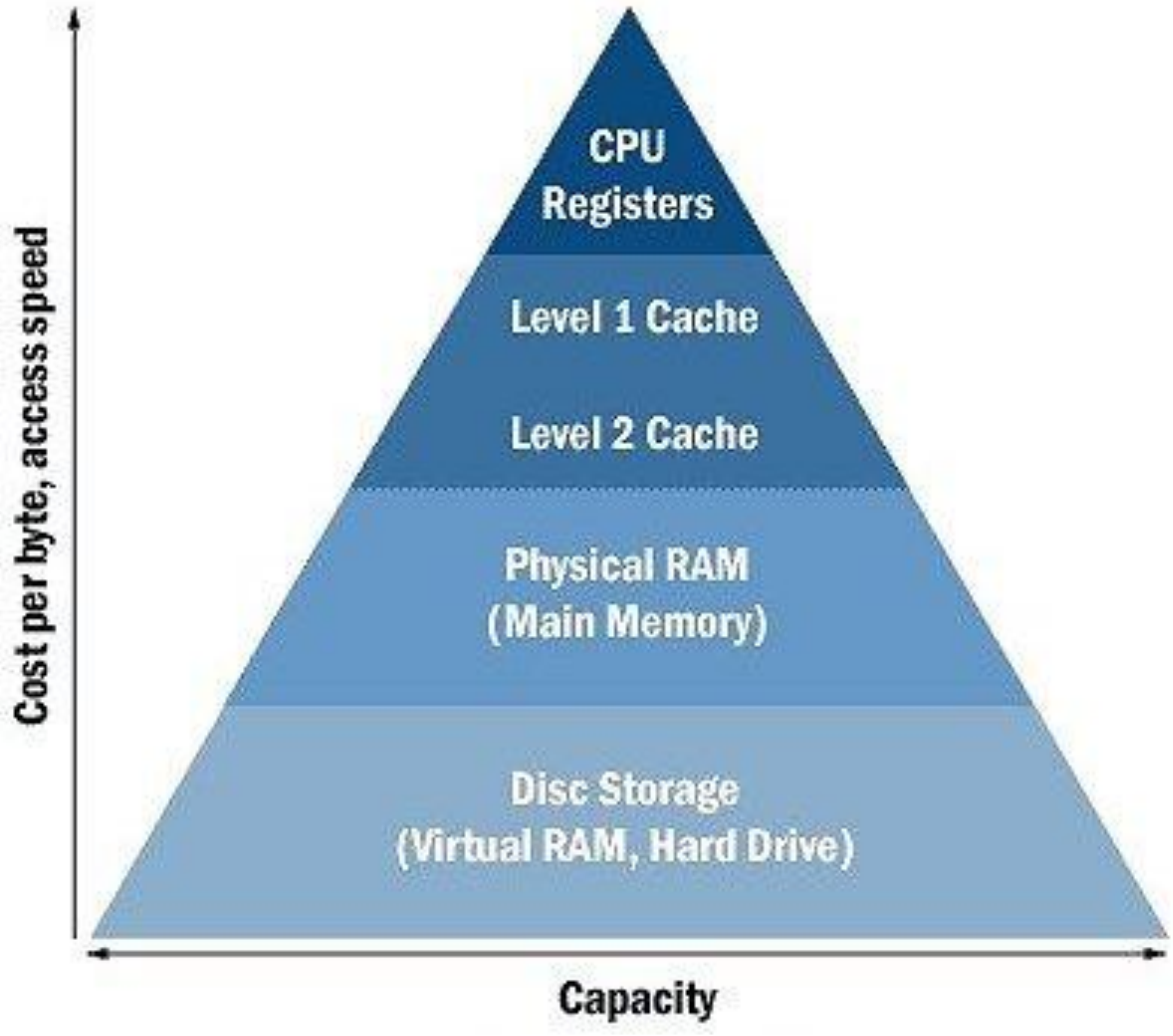
- Computer Architecture (Brief Overview)
- “Hello World” in detail
- C Syntax
- Variables and Types
- Operators
- printf (if there is time)

Von Neumann Architecture

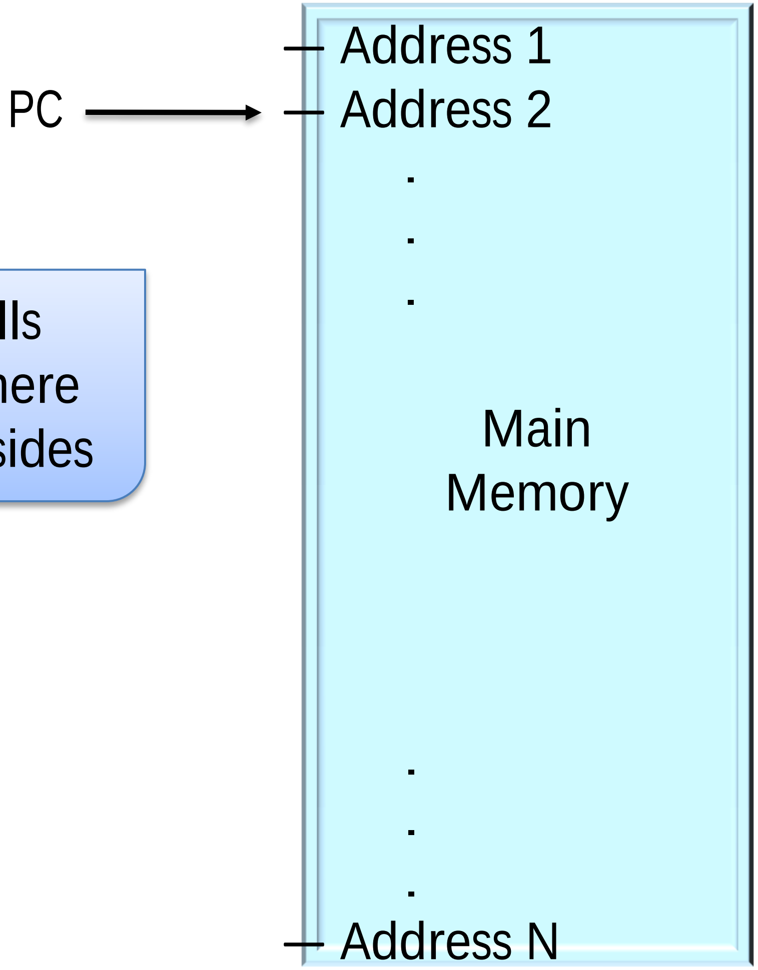


- Data
- Instructions

Computer Memory Architecture



Von Neumann Architecture

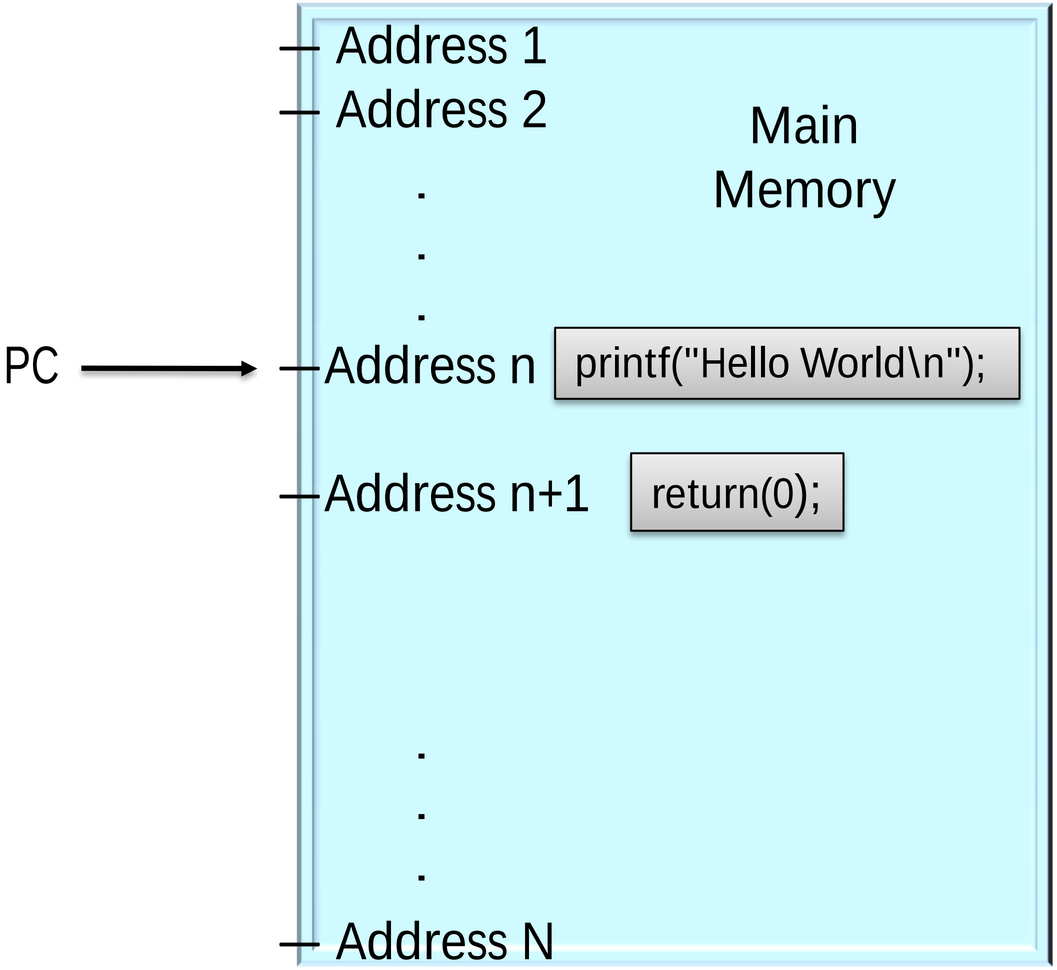


The Program Counter (PC) **points** (= tells the CPU) to the address in memory where the next instruction to be executed resides

Von Neumann Architecture

Hello World

```
#include <stdio.h>

int main(){

  printf("Hello World\n");

  return(0);
}
```



PC

Address 1

Address 2

Main
Memory

.
.
.

Address n printf("Hello World\n");

Address n+1 return(0);

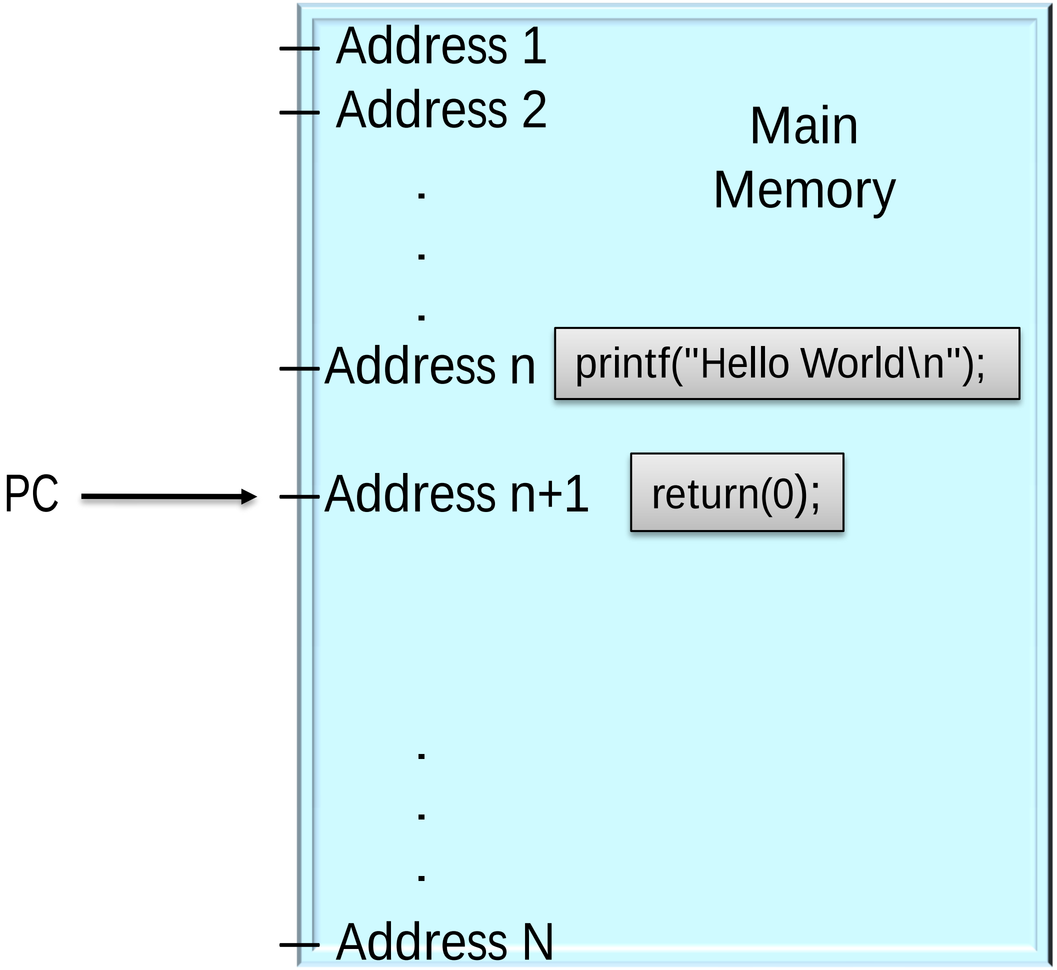
.
.
.

Address N

Von Neumann Architecture

Hello World

```
#include <stdio.h>

int main(){

  printf("Hello World\n");

  return(0);
}
```



PC

Address 1

Address 2

Main Memory

.

.

.

Address n printf("Hello World\n");

Address n+1 return(0);

.

.

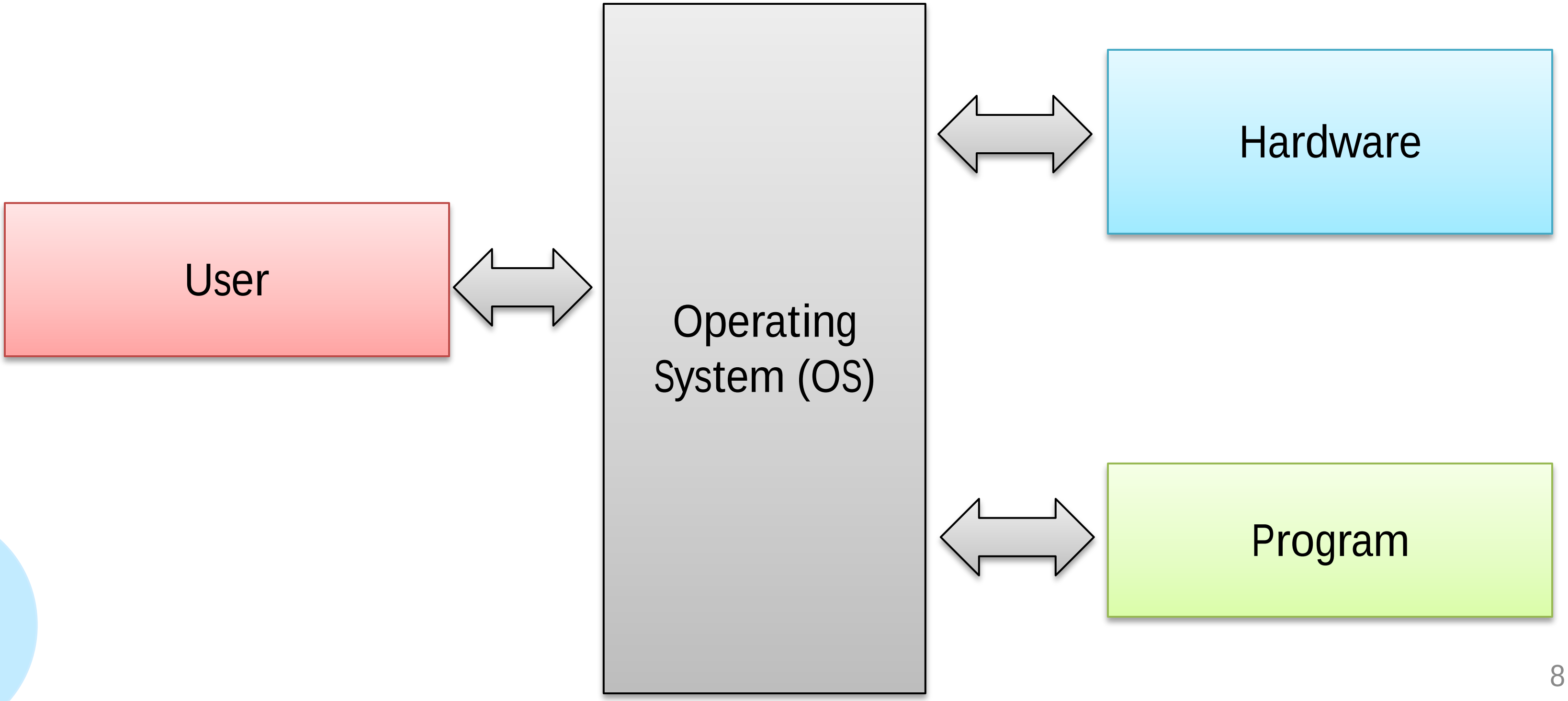
.

Address N

The Operating System

- Manages the hardware
- Allocates resources to programs
- Accommodates user requests
- First program to be executed when computer starts (loaded from ROM)

- Windows
- Unix
- Mac OS
- Android
- Linux
- Solaris
- Chrome OS



Hello World

Global Definitions

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

External Header (standard C library containing functions for Input/Output)

```
int main(){
```

Function definition:

- It’s called *main*
- It does not take any input ( )
- It returns an integer

Body of function

```
    printf("Hello World\n");

    return(0);

}
```

Single statements

C Syntax

- Statements
  - one line commands
  - always end with ;
  - can be grouped between { }
  - spaces are not considered

- Comments

// single line comment

/* multiple lines comments
*/

Hello World + Comments

```
/*
 *  My first C program
 */

#include <stdio.h>


int main(){


   printf(”Hello World\n”);

   return(0);   // return 0 to the OS = OK


}
```

Variables and types

- **Variables** are placeholders for values

```
int x = 2;
x = x + 3; // x value is 5 now
```

- In C, variables are divided into **types**, according to how they are **represented in memory** (always represented in binary)
  - **int**
  - **float**
  - **double**
  - **char**

Variables Declaration

- Before we can use a variable, we must **declare** (= create) it
- When we declare a variable, we specify its **type** and its **name**

```
int x;
float y = 3.2;
```

- Most of the time, the compiler also **allocates memory** for the variable when it’s declared. In that case **declaration = definition**
- There exist special cases in which a variable is declared but not defined, and the computer allocates memory for it only at run time (will see with functions and external variables)

int

- No fractional part or decimal point (ex. +3, -100)
- Represented with 4 bytes (32 bits) in UNIX
- Sign
  - **unsigned** : represents only positive values, all bites for value

    Range: from 0 to 2^32
  - **signed** (default) : 1 bit for sign + 31 for actual value

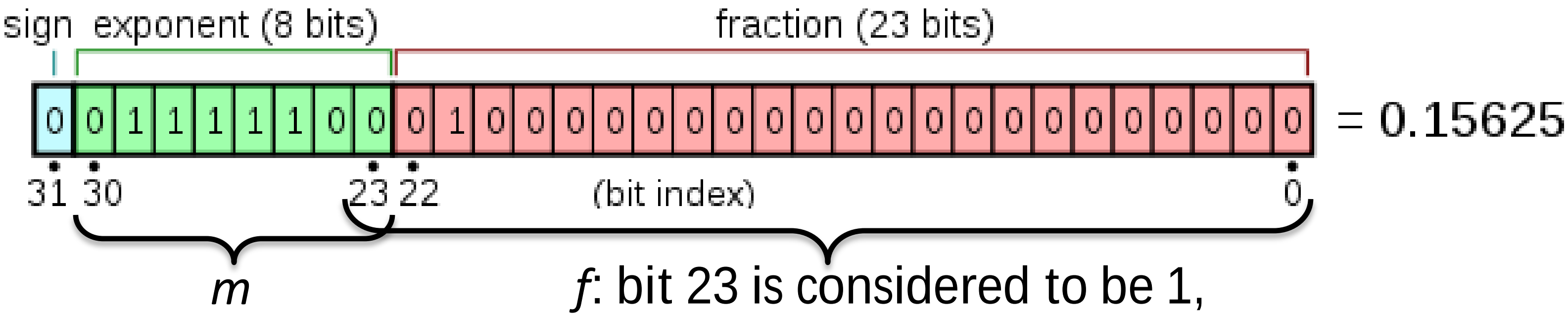
    Range: from -2^31 to 2^31
- Size
  - **short** int : at least 16 bits
  - **long** int : at least 32 bits
  - **long long** int : at least 64 bits
  - size(short) ≤ size(int) ≤ size(long)

```
int x = -12;
unsigned int x  = 5;
short (int) x = 2;
```

float

- Single precision floating point value
- Fractional numbers with decimal point
- Represented with 4 bytes (32 bits)
- Range: -10^(38) to 10^(38)
- Exponential notation : $-\ 0.\underset{f}{278} * 10\text{^}\underset{m}{3}$

```
float x = 11.5;
```

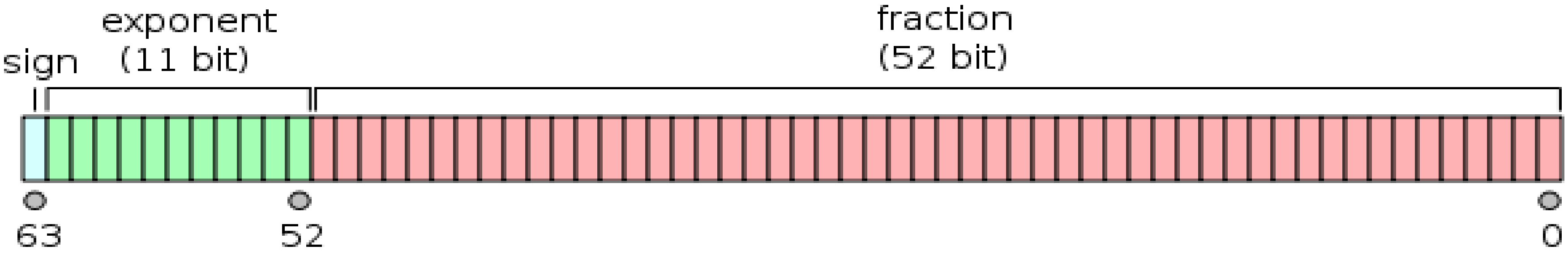


f: bit 23 is considered to be 1, unless *m* is all zeros

$$n_{10} = (-1)^s \cdot (f \cdot 2^{-23}) \cdot 2^{m-127}$$

double

- Double precision floating point
- Represented with 8 bytes (64 bits)



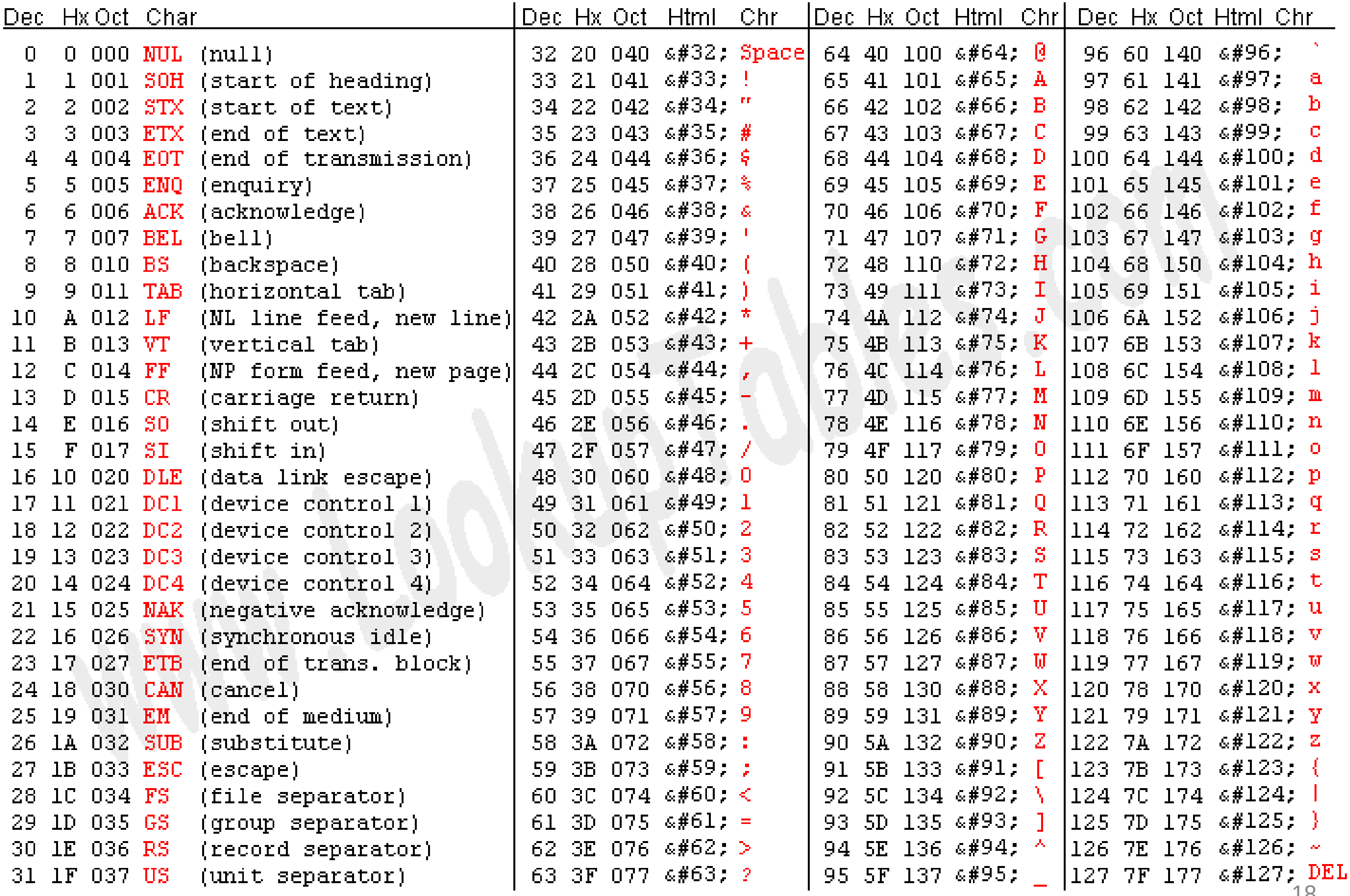
```
double x = 121.45;
```

char

- Character
- Single byte representation
- 0 to 255 values expressed in the ASCII table

```
char c = ‘w’ ;
```

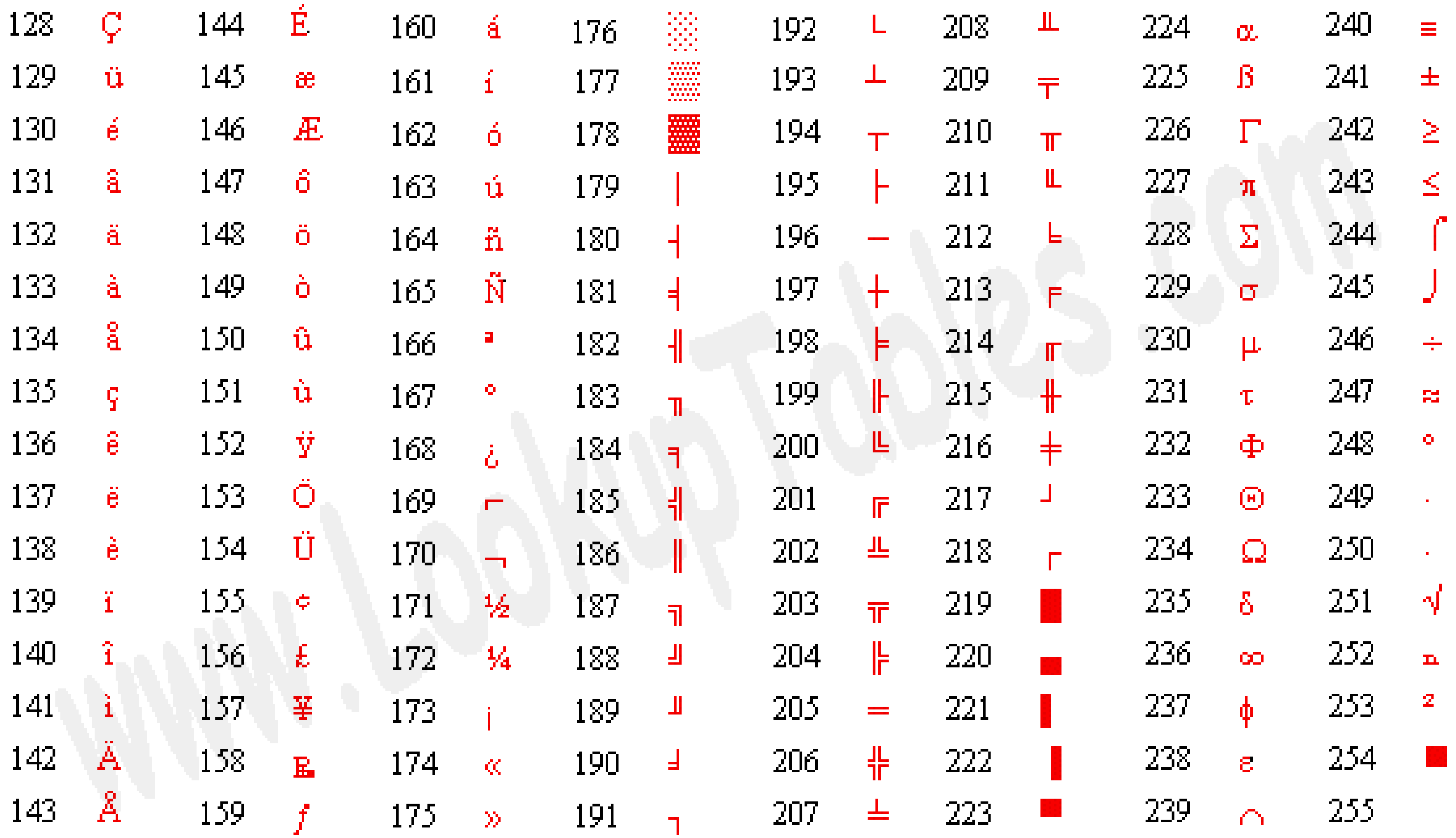
ASCII Table



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Extended ASCII Table



128	Ç	144	É	160	á	176	░	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	ⁿ
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Source: www.LookupTables.com

Casting

- Casting is a method to correctly use variables of different types together
- It allows to treat a variable of one type as if it were of another type in a specific context
- When it makes sense, the compiler does it for us automatically

- Implicit (automatic)

```
int x = 1;
float y = 2.3;
x = x + y;
```

x= 3 compiler automatically casted (=converted) y to be an integer just for this instruction

- Explicit (non-automatic)

```
char c = ‘A’ ;
int x = (int) c;
```

Explicit casting from char to int. The value of x here is 65

Operators

- Assignment =
- Arithmetic * / % + -
- Increment ++ - - += -=
- Relational < <= > >= == !=
- Logical && | | !
- Bitwise & | ~ ^ << >>
- Comma ,

Operators – Assignment

```
int x = 3;

x = 7;



int x, y = 5;

x = y = 7;

float y  = 2.3, z = 3, q = 700;



int i,j,k;

k = (i=2, j=3);

printf( “i = %d, j = %d, k = %d\n” ,i,j,k);
```

The comma operator allows us to perform multiple assignments/declarations

Operators - Arithmetic

`*  /  %  +  -`

- Arithmetic operators have a **precedence**

```
int x;

x = 3 + 5 * 2 - 4 / 2;
```

- We can use parentheses () to impose our precedence order

```
int x;

x = (3 + 5) * (2 – 4) / 2;
```

- % returns the module (or the remainder of the division)

```
int x;

x = 5 % 3;  // x = 2
```

- We have to be careful with integer vs. float division : remember automatic casting!

```
int x = 3;
float y;

y = x / 2; // y = 1.00
```

```
float y;

y = 1 / 2;  // y = 0.00
```

Operators - Arithmetic

*   /   %   +   -

- Arithmetic operators have a **precedence**

```
int x;
x = 3 + 5 * 2 - 4 / 2;
```

- We can use parentheses () to impose our precedence order

```
int x;
x = (3 + 5) * (2 – 4) / 2;
```

- % returns the module (or the remainder of the division)

```
int x;
x = 5 % 3;  // x = 2
```

- We have to be careful with integer vs. float division : remember automatic casting!

```
int x = 3;
float y;

y = x / 2; // y = 1.00
```

Possible fixes:

```
1)float x  = 3;
2)y = (float) x /2;
```

Then `y = 1.50`

```
float y;
y = 1 / 2;  // y = 0.00
```

Possible fix: `y = 1.0/2;`
Then `y = 0.50`

Operators - Increment

++   - -   +=   -=

```
int x = 3, y, z;
```

`x++;` → x is incremented at the end of statement

`++x;` → x is incremented at the beginning of statement

```
y = ++x + 3; // x = x + 1; y = x + 3;

z = x++ + 3; // z = x + 3; x = x + 1;

x -= 2;      // x = x - 2;
```

Operators - Relational

<   <=   >   >=   ==   !=

- Return 0 if statement is false, 1 if statement is true

```
int x = 3, y = 2, z, k, t;

z = x > y;        // z = 1

k = x <= y;       // k = 0

t = x != y;       // t = 1
```

Operators - Logical

`&&   ||   !`

- A variable with value 0 is false, a variable with value !=0 is true

```c
int x = 3, y = 0, z, k, t, q = -3;
```

`z = x && y;     // z = 0;` *x is true but y is false*

`k = x || y;     // k = 1;` *x is true*

`t = !q;         // t = 0;` *q is true*

Review: Operators - Bitwise

- Work on the binary representation of data
- Remember: computers store and see data in binary format!

```
int x, y, z , t, q, s, v;
```

`x = 3;`		00000000000000000000000000000011
`y = 16;`		00000000000000000000000000010000
`z = x << 1;`	equivalent to $z = x \cdot 2^1$	00000000000000000000000000000110
`t = y >> 3;`	equivalent to $t = y \cdot 2^{-3}$	00000000000000000000000000000010
`q = x & y;`		00000000000000000000000000000000
`s = x | y;`		00000000000000000000000000010011
`v = x ^ y;`		00000000000000000000000000010011

XOR

printf

- `printf` is a function used to print to standard output (command line)

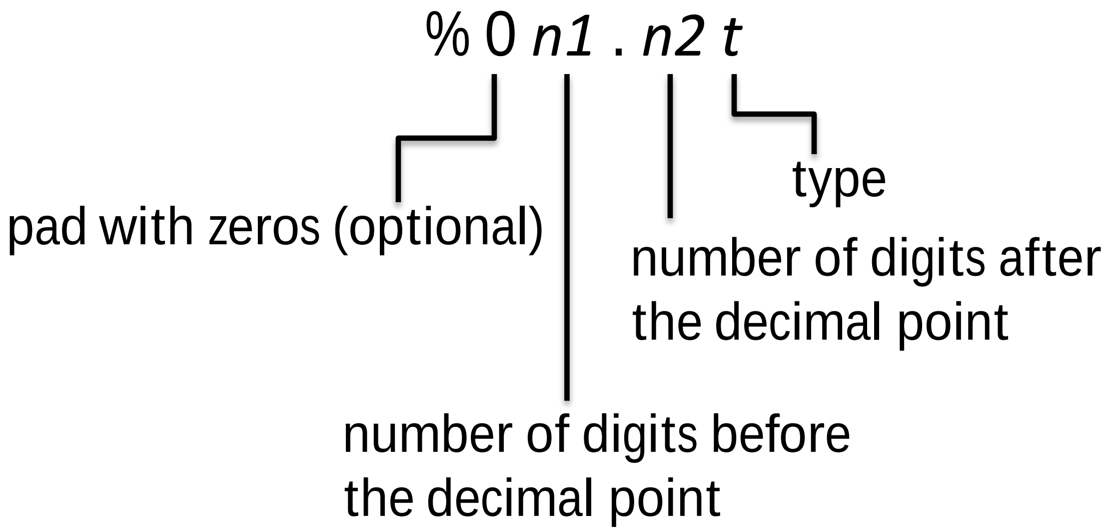
- Syntax:

  `printf(“format1 format2 …”, variable1, variable2,…);`

- Format characters:

– %d or %i	integer
– %f	float
– %lf	double
– %c	char
– %u	unsigned
– %s	string

Format



% 0 *n1* . *n2* *t*

pad with zeros (optional)

number of digits before the decimal point

number of digits after the decimal point

type

printf

```
#include <stdio.h>

int main() {

   int a,b;
   float c,d;
   a = 15;
   b = a / 2;

   printf(”%d\n”,b);
   printf(”%3d\n”,b);
   printf(”%03d\n”,b);

   c = 15.3;
   d = c / 3;
   printf(”%3.2f\n”,d);

   return(0);

}
```

Output:

```
7
  7
007



5.10
```

printf

<u>Escape sequences</u>

\n	newline
\t	tab
\v	vertical tab
\f	new page
\b	backspace
\r	carriage return

Assignment

- Read PCP Chapter 3 and 4

COMSW 1003-1

Introduction to Computer Programming in C

Lecture 4

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Announcements

- HW 1 is due on Monday, February 14th at the beginning of class, no exceptions
- Read so far: PCP Chapters 1 to 4
- Reading for next Wednesday: PCP Chapter 5

Review – Access CUNIX

http://www1.cs.columbia.edu/~bert/courses/1003/cunix.html

1) Enable windowing environment
   - X11, Xming, X-Server

2) Launch SSH session (login with UNI and password)
   - Terminal, Putty

3) Launch Emacs

```
$ emacs &
```

4) Open/create a file, than save it with .c extension

5) Compile source code into executable with gcc

Review - Compiling your C code

- GCC : **GNU Compiler Collection**
- When you invoke GCC, it normally does preprocessing, compilation, assembly and linking
  - Basic Command
    - gcc myProgram.c
    - ./a.out

Run compiled program (executable)

  - More advanced options
    - gcc –Wall –o myProgram myProgram.c
    - ./myProgram

Review - Compiling your C code

- GCC : **GNU Compiler Collection**
- When you invoke GCC, it normally does preprocessing, compilation, assembly and linking
  - Basic Command
    - gcc myProgram.c
    - ./a.out

Run compiled program (executable)

Display all types of warnings, not only errors

Specify name of the executable

    - gcc –Wall –o myProgram myProgram.c
    - ./myProgram

Run compiled program (executable)

Review: C Syntax

- Statements
  - one line commands
  - always end with ;
  - can be grouped between { }

- Comments

/ / single line comment

/ * multiple lines comments
* /

Review : Variables and types

- **Variables** are placeholders for values

```
int x = 2;
x = x + 3; // x value is 5 now
```

- In C, variables are divided into **types**, according to how they are **represented in memory** (always represented in binary)

– int	**4 bytes, signed/unsigned**
– float	**4 bytes, decimal part + exponent**
– double	**8 bytes**
– char	**1 byte, ASCII Table**

Review : Casting

- Casting is a method to correctly use variables of different types together
- It allows to treat a variable of one type as if it were of another type in a specific context
- When it makes sense, the compiler does it for us automatically

- Implicit (automatic)

```
int x =1;
float y = 2.3;
x = x + y;
```

x= 3 compiler automatically casted (=converted) y to be an integer just for this instruction

- Explicit (non-automatic)

```
char c = 'A' ;
int x = (int) c;
```

Explicit casting from char to int. The value of x here is 65

Today

- Operators
- printf()
- Binary logic

Operators

Assignment	=
Arithmetic	* / % + -
Increment	++ - - += -=
Relational	< <= > >= == !=
Logical	&& || !
Bitwise	& | ~ ^ << >>
Comma	,

Operators – Assignment and Comma

```
int x = 3;

x = 7;



int x, y = 5;

x = y = 7;

float y  = 2.3, z = 3, q = 700;



int i,j,k;

k = (i=2, j=3);

printf( “i = %d, j = %d, k = %d\n” ,i,j,k);
```

The comma operator allows us to perform multiple assignments/declarations

Operators - Arithmetic

* / % + -

- Arithmetic operators have a **precedence**

```
int x;
x = 3 + 5 * 2 - 4 / 2;
```

- We can use parentheses () to impose our precedence order

```
int x;
x = (3 + 5) * (2 – 4) / 2;
```

- % returns the module (or the remainder of the division)

```
int x;
x = 5 % 3;  // x = 2
```

- We have to be careful with integer vs. float division : remember automatic casting!

```
int x = 3;
float y;

y = x / 2; // y = 1.00
```

```
float y;
y = 1 / 2;  // y = 0.00
```

Operators - Arithmetic

*   /   %   +   -

- Arithmetic operators have a **precedence**

```
int x;

x = 3 + 5 * 2 - 4 / 2;
```

- We can use parentheses () to impose our precedence order

```
int x;

x = (3 + 5) * (2 – 4) / 2;
```

- % returns the module (or the remainder of the division)

```
int x;

x = 5 % 3;  // x = 2
```

- We have to be careful with integer vs. float division : remember automatic casting!

```
int x = 3;
float y;

y = x / 2; // y = 1.00
```

Possible fixes:
1)`float x  = 3;`
2)`y = (float) x /2;`
Then `y = 1.50`

```
float y;

y = 1 / 2;  // y = 0.00
```

Possible fix: `y = 1.0/2;`
Then `y = 0.50`

Operators – Increment/Decrement

`++` `--` `+=` `-=`

```
int x = 3, y, z;
```

`x++;` → x is incremented at the end of statement

`++x;` → x is incremented at the beginning of statement

```
y = ++x + 3; // x = x + 1; y = x + 3;

z = x++ + 3; // z = x + 3; x = x + 1;

x -= 2;      // x = x - 2;
```

Operators - Relational

`<   <=   >   >=   ==   !=`

- Return 0 if statement is false, 1 if statement is true

```
int x = 3, y = 2, z, k, t;

z = x > y;        // z = 1

k = x <= y;       // k = 0

t = x != y;       // t = 1
```

Operators - Logical

`&&   ||   !`

- A variable with value 0 is false, a variable with value !=0 is true

```
int x = 3, y = 0, z, k, t, q = -3;
```

`z = x && y;   // z = 0;`   *x is true but y is false*

`k = x || y;   // k = 1;`   *x is true*

`t = !q;       // t = 0;`   *q is true*

Operators - Bitwise

- Work on the binary representation of data
- Remember: computers store and see data in binary format!

`int x, y, z , t, q, s, v;`

`x = 3;` 00000000000000000000000000000011

`y = 16;` 00000000000000000000000000010000

`z = x << 1;` equivalent to $z = x \cdot 2^1$ 00000000000000000000000000000110

`t = y >> 3;` equivalent to $t = y \cdot 2^{-3}$ 00000000000000000000000000000010

`q = x & y;` 00000000000000000000000000000000

`s = x | y;` 00000000000000000000000000010011

`v = x ^ y;` 00000000000000000000000000010011

↓

XOR

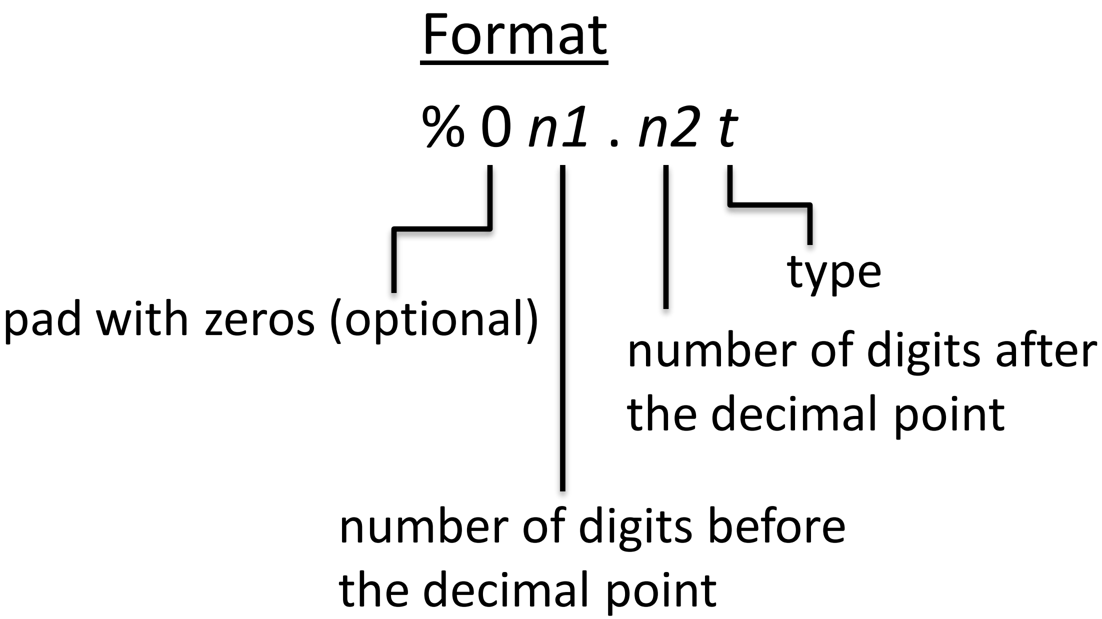
printf

- `printf` is a function used to print to standard output (command line)

- Syntax:
  `printf(“format1 format2 …”, variable1, variable2,…);`

- Format characters:

%d or %i	integer
%f	float
%lf	double
%c	char
%u	unsigned
%s	string



Format

% 0 *n1* . *n2* *t*

pad with zeros (optional)

number of digits before the decimal point

number of digits after the decimal point

type

printf

```
#include <stdio.h>

int main() {

   int a,b;
   float c,d;
   a = 15;
   b = a / 2;

   printf(”%d\n”,b);
   printf(”%3d\n”,b);
   printf(”%03d\n”,b);

   c = 15.3;
   d = c / 3;
   printf(”%3.2f\n”,d);

   return(0);

}
```

Output:

```
7
  7
007



5.10
```

printf

Escape sequences

\n	newline
\t	tab
\v	vertical tab
\f	new page
\b	backspace
\r	carriage return

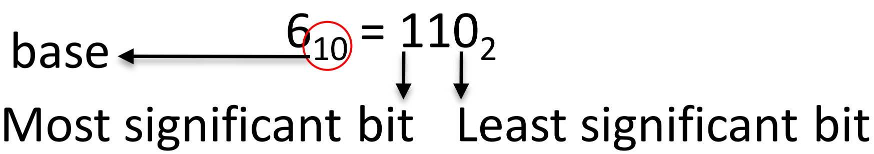
Binary Logic

- 1 = true, 0 = false
- Decimal to binary conversion

$$6_{10} = 110_2$$

Binary Logic

- 1 = true, 0 = false
- Decimal to binary conversion

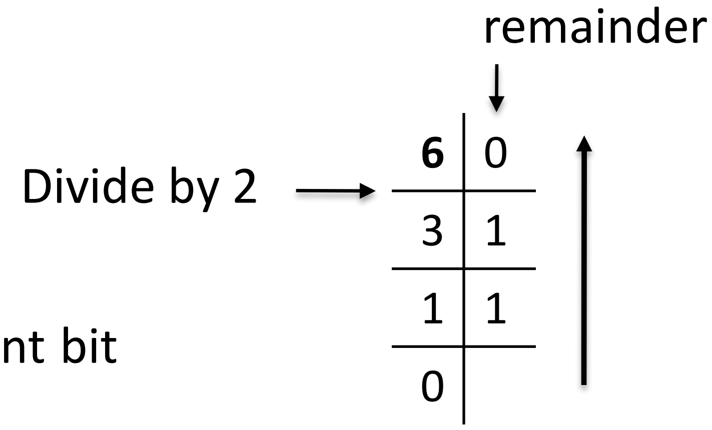


$6_{10} = 110_2$

base

Most significant bit

Least significant bit



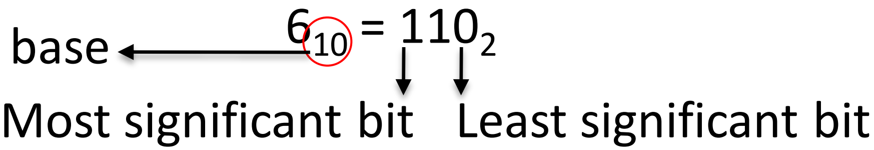
remainder

Divide by 2

6	0
3	1
1	1
0	

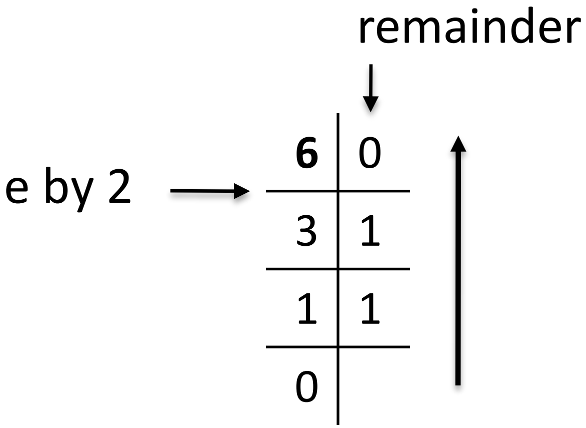
Binary Logic

- 1 = true, 0 = false
- Decimal to binary conversion



$6_{10} = 110_2$

Divide by 2



6	0
3	1
1	1
0	

- Binary to decimal conversion

$11001_2 = 1x2^0 + 0x2^1 + 0x2^2 + 1x2^3 + 1x2^4 = 25$

Binary Logic

- 1 = true, 0 = false

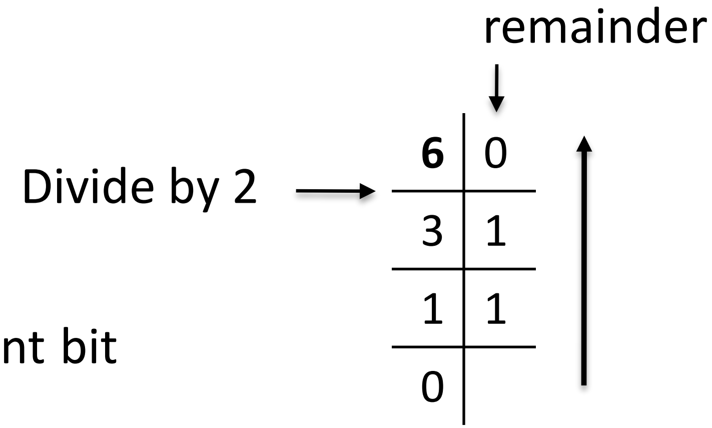
- Decimal to binary conversion

$6_{10} = 110_2$

base

Most significant bit

Least significant bit



Divide by 2

remainder

6	0
3	1
1	1
0	

- Binary to decimal conversion

$11001_2 = 1x2^0 + 0x2^1 + 0x2^2 + 1x2^3 + 1x2^4 = 25$

- AND

v = x & y

x	y	v
0	0	0
0	1	0
1	0	0
1	1	1

- OR

v = x | y

x	y	v
0	0	0
0	1	1
1	0	1
1	1	1

- NOT

v = !x

x	v
0	1
1	0

- EXOR

v = x ^ y

x	y	v
0	0	0
0	1	1
1	0	1
1	1	0

Homework 1 review

HOW TO COMPRESS/UNCOMPRESS folders in UNIX

- Compress folder ~/COMS1003/HW1 to HW1.tar.gz

```
tar -zcvf HW1.tar.gz  ~/COMS1003/HW1
```

- Uncompress HW1.tar.gz to folder ~/COMS1003/HW1new

```
tar -zxvf HW1.tar.gz  -C  ~/COMS1003/HW1new
```

(note: ~/COMS1003/HW1new must exist already)

COMSW 1003-1

Introduction to Computer Programming in C

Lecture 5

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Announcements

- Exercise 1 solution out
- Exercise 2 out

- Read PCP Ch 6

Today

- Review of operators and printf()
- Binary Logic
- Arrays
- Strings

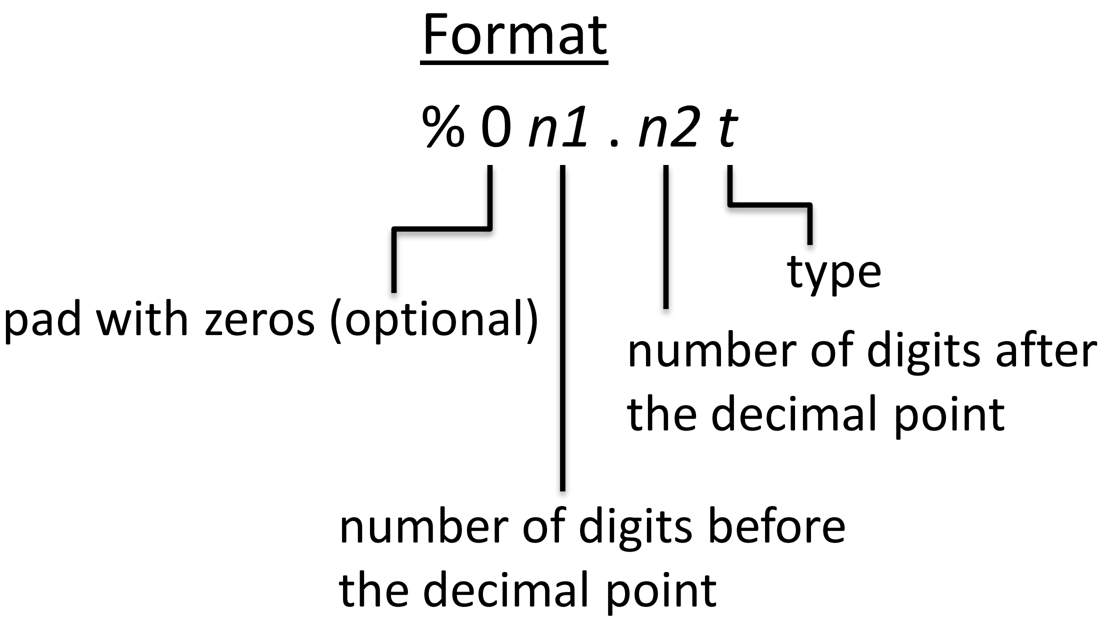
Review : printf

- `printf` is a function used to print to standard output (command line)
- Syntax:

  `printf(“format1 format2 …”, variable1, variable2,…);`

- Format characters:

%d or %i	integer
%f	float
%lf	double
%c	char
%u	unsigned
%s	string



Format

% 0 *n1* . *n2* *t*

pad with zeros (optional)

type

number of digits after the decimal point

number of digits before the decimal point

Review : printf

printfExample.c

```
#include <stdio.h>

int main() {

  int a,b;
  float c,d;
  a = 15;
  b = a / 2;

  printf(”%d\n”,b);
  printf(”%3d\n”,b);
  printf(”%03d\n”,b);

  c = 15.3;
  d = c / 3;
  printf(”%3.2f\n”,d);

  return(0);

}
```

Output:

```
7
  7
007


5.10
```

Review : printf

Escape sequences

\n	newline
\t	tab
\v	vertical tab
\f	new page
\b	backspace
\r	carriage return

Binary Logic

- In binary logic, variables can have only 2 values:
  - True (commonly associated with 1)
  - False (commonly associated with 0)

- Binary Operations are defined through TRUTH TABLES

AND

v = x & y

x	y	v
0	0	0
0	1	0
1	0	0
1	1	1

NOT

v = !x

x	v
0	1
1	0

OR

v = x | y

x	y	v
0	0	0
0	1	1
1	0	1
1	1	1

EXOR

v = x ^ y

x	y	v
0	0	0
0	1	1
1	0	1
1	1	0

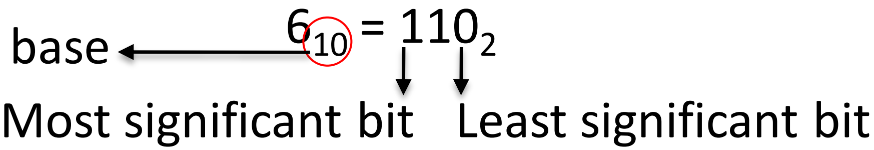
Binary Logic

- 1 = true, 0 = false
- Decimal to binary conversion

$6_{10} = 110_2$

Binary Logic

- 1 = true, 0 = false
- Decimal to binary conversion

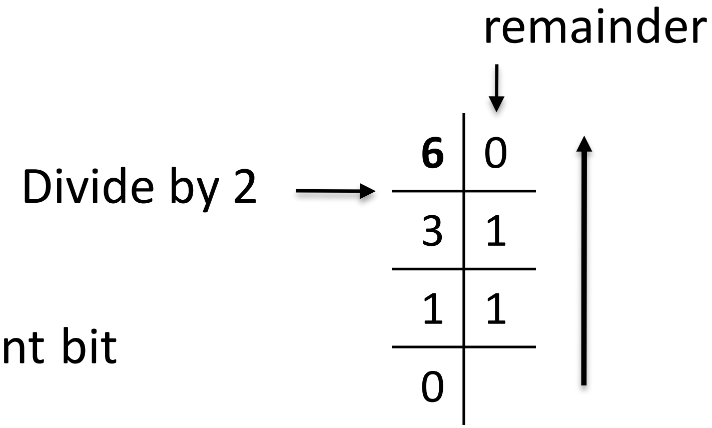


$6_{10} = 110_2$

base

Most significant bit

Least significant bit



remainder

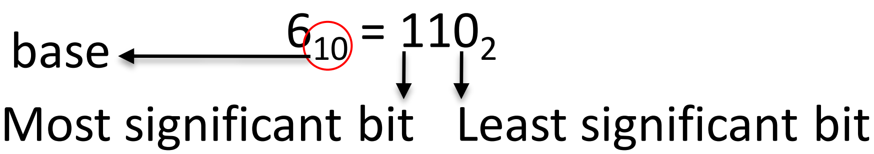
Divide by 2

6	0
3	1
1	1
0	

Binary Logic

- 1 = true, 0 = false

- Decimal to binary conversion



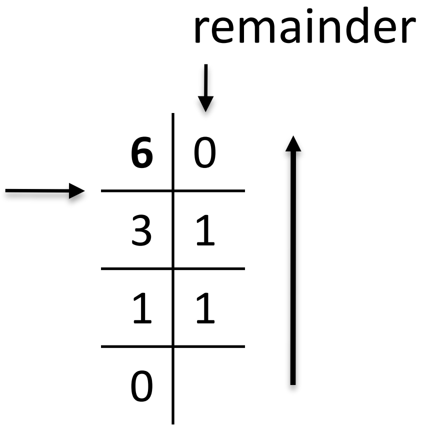
$6_{10} = 110_2$

base

Most significant bit

Least significant bit

Divide by 2



remainder

6	0
3	1
1	1
0	

- Binary to decimal conversion

$11001_2 = 1x2^0 + 0x2^1 + 0x2^2 + 1x2^3 + 1x2^4 = 25$

Binary Logic

- 1 = true, 0 = false

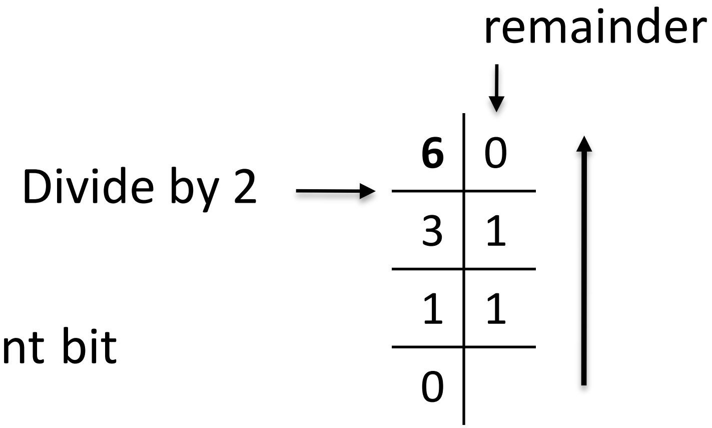
- Decimal to binary conversion

$6_{10} = 110_2$

base

Most significant bit

Least significant bit



6	0
3	1
1	1
0	

- Binary to decimal conversion

$11001_2 = 1x2^0 + 0x2^1 + 0x2^2 + 1x2^3 + 1x2^4 = 25$

- AND
  v = x & y

x	y	v
0	0	0
0	1	0
1	0	0
1	1	1

- OR
  v = x | y

x	y	v
0	0	0
0	1	1
1	0	1
1	1	1

- NOT
  v = !x

x	v
0	1
1	0

- EXOR
  v = x ^ y

x	y	v
0	0	0
0	1	1
1	0	1
1	1	0

Review: Operators

- Assignment =
- Arithmetic * / % + -
- Increment ++ - - += -=
- Relational < <= > >= == !=
- Logical && | | !
- Bitwise & | ~ ^ << >>
- Comma ,

Operators - Bitwise

- Work on the binary representation of data
- Remember: computers store and see data in binary format!

`int x, y, z , t, q, s, v;`

`x = 3;` 00000000000000000000000000000011
`y = 16;` 00000000000000000000000000010000

`z = x << 1;` equivalent to $z = x \cdot 2^1$ 00000000000000000000000000000110

`t = y >> 3;` equivalent to $t = y \cdot 2^{-3}$ 00000000000000000000000000000010

`q = x & y;` 00000000000000000000000000000000

`s = x | y;` 00000000000000000000000000010011

`v = x ^ y;` 00000000000000000000000000010011

↓ XOR

Operators - Arithmetic

`*   /   %   +   -`

- Arithmetic operators have a **precedence**

```
int x;

x = 3 + 5 * 2 - 4 / 2;
```

- We can use parentheses () to impose our precedence order

```
int x;

x = (3 + 5) * (2 – 4) / 2;
```

- % returns the module (or the remainder of the division)

```
int x;

x = 5 % 3;  // x = 2
```

- We have to be careful with integer vs. float division : remember automatic casting!

```
int x = 3;
float y;

y = x / 2; // y = 1.00
```

Possible fixes:

```
1)float x  = 3;
2)y = (float) x /2;
```

Then `y = 1.50`

```
float y;

y = 1 / 2;  // y = 0.00
```

Possible fix: `y = 1.0/2;`
Then `y = 0.50`

Operators – Increment/Decrement

`++    --    +=    -=`

```
int x = 3, y, z;
```

`x++;` → x is incremented at the end of statement

`++x;` → x is incremented at the beginning of statement

```
y = ++x + 3;  // x = x + 1; y = x + 3;

z = x++ + 3;  // z = x + 3; x = x + 1;

x -= 2;       // x = x - 2;
```

Operators - Relational

< <= > >= == !=

- Return 0 if statement is false, 1 if statement is true

```
int x = 3, y = 2, z, k, t;

z = x > y;       // z = 1

k = x <= y;      // k = 0

t = x != y;      // t = 1
```

Operators - Logical

`&&    ||    !`

- A variable with value 0 is false, a variable with value !=0 is true

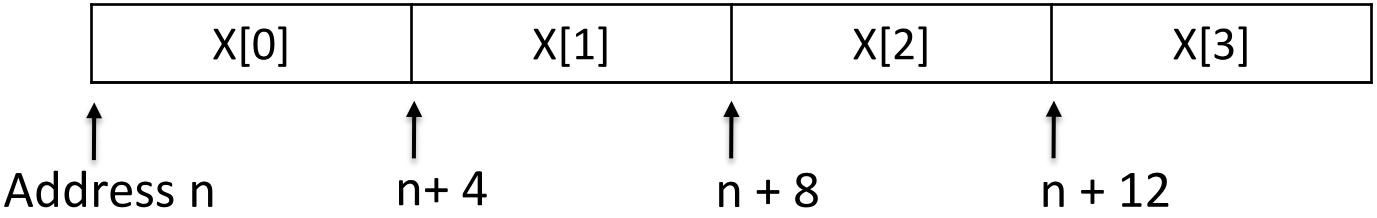
```
int x = 3, y = 0, z, k, t, q = -3;
```

`z = x && y;`	`// z = 0;`	*x is true but y is false*
`k = x || y;`	`// k = 1;`	*x is true*
`t = !q;`	`// t = 0;`	*q is true*

Arrays

- “A set of consecutive memory locations used to store data” [PCP, Ch 5]

```
int X[4];  // a vector containing 4 integers
```



X[0] X[1] X[2] X[3]

Address n n+ 4 n + 8 n + 12

- Indexing starts at 0 !

```
X[0] = 3;
X[2] = 7;
```

- Be careful not to access uninitialized elements!

```
int c = X[7];
```

gcc will not complain about this, but the value of x is going to be random!

Arrays

- Multidimensional arrays

```
int arr[4][3];  // a matrix containing 4x3 = 12 integers
```

arr[0][0]	arr[0][1]	arr[0][2]
arr[1][0]	arr[1][1]	arr[1][2]
arr[2][0]	arr[2][1]	arr[2][2]
arr[3][0]	arr[3][1]	arr[3][2]

- Indexing starts at 0 !

```
arr[0][0] = 1;
arr[3][1] = 7;
```

- Initialize arrays

```
int X[4] = { 3, 6, 7, 89};
int Y[2][4] = { {19, 2, 6, 99}, {55, 5, 555, 0} };
int Arr[] = { 3, 6, 77};
```

This automatically allocates memory for an array of 3 integers

Strings

- Strings are arrays of `char`
- ‘\0’ is a special character that indicates the end of a string

```
char s[6] = {‘H’, ’e’, ‘l’, ‘l’, ‘o’, ‘\0’};
```

We need 6 characters because there is ‘\0’

‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’

```
char s[10] = “Hello”;
```

‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’				

```
char s[6];
s[0] = ‘H’;
s[1] = ‘e’;
s[2] = ‘l’;
s[3] = ‘l’;
s[4] = ‘o’;
s[5] = ‘\0’;
```

- Difference between string and char

```
char c = ‘a’ ;
```

‘a’

```
char s[2] = “a” ;
```

‘a’	‘\0’

Strings functions

String specific functions are included in the library string.h

```
#include <string.h>

char s[6];
s = “Hello”;
```

Illegal ! String assignment can be done only at declaration!

- strcpy() : copy a string to another

```
strcpy( string1 , string2 );
```

Copy string2 to string1

```
char s[6];
strcpy(s, “Hello”);
```

String functions

String specific functions are included in the library string.h

- strcmp() : compare two strings

```
strcmp( string1 , string2 );
```

Returns :
0 if string1 and string2 are the same
value != 0 otherwise

```
char s1[] = “Hi”;
char s2[] = “Him”;
char s3[3];
strcpy( s3, s1 );
int x = strcmp( s1, s2 );   // x != 0
int y = strcmp( s1, s3 );   // y = 0
```

Strings functions

String specific functions are included in the library string.h

- strcat() : concatenate two strings

```
strcat( string1 , string2);
```

Concatenate *string2* at the end of *string1*

```
char s1[] = “Hello ”;
char s2[] = “World!”;
strcat(s1, s2);
```

‘H’	‘e’	‘l’	‘l’	‘o’	‘ ‘	‘W’	‘o’	‘r’	‘l’	‘d’	‘\0’

- strlen() : returns the length of a string (does not count ‘\0’)

```
strlen( string );
```

```
char s1[] = “Hello”;
int x = strlen(s1);     // x = 5
```

Reading Strings

Use functions from library stdio.h

- fgets() : get string from standard input (command line)

```
fgets( name , sizeof(name), stdin);
```

Reads a maximum of sizeof(*name*) characters of a string from stdin and saves them into string *name*

```
char s1[100];
fgets( s1, sizeof(s1), stdin);
```

NOTE: fgets() reads the newline character ‘\n’, so we should substitute it with ‘\0’;

```
name[strlen(name)-1] = ‘\0’;
```

‘H’	‘e’	‘l’	‘l’	‘o’	‘\n’

‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’

- sizeof() : returns the size (number of bytes occupied in memory) of a variable (for strings it counts the number of elements, including ‘\0’)

Reading numbers – Option 1

- First, read a string
- Then, convert string to number
- sscanf() : get string from standard input (command line)

```
sscanf( string, "format", &var1, …, &varN);
```

```
char s1[100];
int x, y;
printf("Please enter two numbers separated by a space\n")
fgets( s1, sizeof(s1), stdin);
```

User enters: 3 18

```
sscanf( s1, "%d %d", &x, &y );

// x = 3; y = 18;
```

Reading numbers – Option 2

- Read directly the number
- scanf() : get string from standard input (command line) and automatically convert into a number

```
scanf( “format”, &var1, …, &varN);
```

```
int x, y;
printf(“Please enter two numbers separated by a space\n”)
```

User enters: 3 18

```
scanf( “%d %d”, &x, &y );

// x = 3; y = 18;
```

Strings functions - recap

sumNums.c

```
char s1[] = “Hello”;   char s2[] = “He”;   int x;   char c;
```

- strcmp(s1, s2)

```
x = strcmp(s1, s2) // x != 0
```

- strcpy(s1, s2)

```
strcpy( s2, s1 ); // s2 = “Hello”
```

- strcat(s1, s2)

```
strcat( s2, s1 ); //s2 = “HelloHello”
```

- strlen(s)

```
x = strlen(s1);   // x  = 5;
```

- sizeof(s)

```
x = sizeof(s1);   // x = 6;
```

- fgets(s, sizeof(s1), stdin)

```
fgets( s1, sizeof(s1), stdin);
```

User enters “7R”

- sscanf(s, “%d”, &var)

```
sscanf( s1, “%d%c”, &x, &c);
// x = 7; c = ‘R’;
```

Read PCP Ch 6

Homework 1 review

HOW TO COMPRESS/UNCOMPRESS folders in UNIX

- Compress folder ~/COMS1003/HW1 to HW1.tar.gz

```
tar -zcvf HW1.tar.gz  ~/COMS1003/HW1
```

- Uncompress HW1.tar.gz to folder ~/COMS1003/HW1new

```
tar -zxvf HW1.tar.gz  -C  ~/COMS1003/HW1new
```

(note: ~/COMS1003/HW1new must exist already)

COMsW 1003-1

Introduction to Computer Programming in C

Lecture 6

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Announcements

Homework 1 is due next Monday

Exercise 2 is out

Today

- Strings
- Control Flow
- Loops (if time permits)

Review - arrays

- Multidimensional arrays

```
int X[4][3];  // a matrix containing 4x3 = 12 integers
```

X[0][0]	X[0][1]	X[0][2]
X[1][0]	X[1][1]	X[1][2]
X[2][0]	X[2][1]	X[2][2]
X[3][0]	X[3][1]	X[3][2]

- Indexing starts at 0 !

```
X[0][0] = 1;
X[3][1] = 7;
```

- Initialize says

```
int arr[4] = { 3, 6, 7, 89};
int arr2[2][4] = { {19, 2, 6, 99}, {55, 5, 555, 0} };
int arr[] = { 3, 6, 77};
```

This automatically allocates memory for an array of 3 integers

Strings

- Strings are arrays of `char`
- ‘\0’ is a special character that indicates the end of a string

```
char s[6] = {‘H’, ’e’, ‘l’, ‘l’, ‘o’, ‘\0’};
```

We need 6 characters because there is ‘\0’

‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’

```
char s[10] = “Hello”;
```

‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’				

```
char s[6];
s[0] = ‘H’;
s[1] = ‘e’;
s[2] = ‘l’;
s[3] = ‘l’;
s[4] = ‘o’;
s[5] = ‘\0’;
```

- Difference between string and char

```
char c =  ‘a’ ;
```

‘a’

```
char s[2] =  “a” ;
```

‘a’	‘\0’

Strings functions

String specific functions are included in the library string.h

```
#include <string.h>

char s[6];
s = "Hello";
```

Illegal ! String assignment can be done only at declaration!

- strcpy() : copy a string to another

```
strcpy( string1 , string2 );
```

Copy string2 to string1

```
char s[6];
strcpy(s, "Hello");
```

String functions

String specific functions are included in the library string.h

- strcmp() : compare two strings

```
strcmp( string1 , string2 );
```

Returns :
0 if string1 and string2 are the same
value != 0 otherwise

```
char s1[] = “Hi”;
char s2[] = “Him”;
char s3[3];
strcpy( s3, s1 );
int x = strcmp( s1, s2 );    // x != 0
int y = strcmp( s1, s3 );    // y = 0
```

Strings functions

String specific functions are included in the library string.h

- strcat() : concatenate two strings

```
strcat( string1 , string2);
```

Concatenate *string2* at the end of *string1*

```
char s1[] = “Hello ”;
char s2[] = “World!”;
strcat(s1, s2);
```

‘H’	‘e’	‘l’	‘l’	‘o’	‘ ‘	‘W’	‘o’	‘r’	‘l’	‘d’	‘\0’

- strlen() : returns the length of a string (does not count ‘\0’)

```
strlen( string );
```

```
char s1[] = “Hello”;
int x = strlen(s1);     // x = 5
```

Reading Strings

Use functions from library stdio.h

- fgets() : get string from standard input (command line)

```
fgets( name , sizeof(name), stdin);
```

Reads a maximum of sizeof(*name*) characters of a string from stdin and saves them into string *name*

```
char s1[100];
fgets( s1, sizeof(s1), stdin);
```

NOTE: fgets() reads the newline character ‘\n’, so we should substitute it with ‘\0’;

```
s1[strlen(s1)-1] = '\0';
```

‘H’	‘e’	‘l’	‘l’	‘o’	‘\n’

‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’

- sizeof() : returns the size (number of bytes occupied in memory) of a variable (for strings it counts the number of elements, including ‘\0’)

Reading numbers – Option 1

- First, read a string
- Then, convert string to number
- sscanf() : get string from standard input (command line)

```
sscanf( string, “format”, &var1, …, &varN);
```

```
char s1[100];
int x, y;
printf(“Please enter two numbers separated by a space\n”)
fgets( s1, sizeof(s1), stdin);
```

User enters: 3 18

```
sscanf( s1, “%d %d”, &x, &y );

// x = 3; y = 18;
```

Reading numbers – Option 2

- Read directly the number
- scanf() : get string from standard input (command line) and automatically convert into a number

```
scanf( “format”, &var1, …, &varN);
```

```
int x, y;
printf(“Please enter two numbers separated by a space\n”)
```

User enters: 3 18

```
scanf( “%d %d”, &x, &y );

// x = 3; y = 18;
```

sumNums.c

Strings functions - recap

```
char s1[] = “Hello”;    char s2[] = “He”;    int x;    char c;
```

- strcmp(s1, s2)

```
x = strcmp(s1, s2) // x != 0
```

- strcpy(s1, s2)

```
strcpy( s2, s1 ); // s2 = “Hello”
```

- strcat(s1, s2)

```
strcat( s2, s1 ); //s2 = “HelloHello”
```

- strlen(s)

```
x = strlen(s1);   // x  = 5;
```

- sizeof(s)

```
x = sizeof(s1);   // x = 6;
```

- fgets(s, sizeof(s1), stdin)

```
fgets( s1, sizeof(s1), stdin);
```

User enters “7R”

- sscanf(s, “%d”, &var)

```
sscanf( s1, “%d%c”, &x, &c);
// x = 7; c = ‘R’;
```

Example – sumNums.c

Control Flow

- So far we have seen **linear programs**, statements are executed in the order in which they are written
- What if we want to skip some instructions, or execute them only under certain conditions?
- Solution: **control flow**

Control flow – General syntax

```
keyword ( condition ) {

      body statement 1;
              .
              .
              .
      body statement n;

}
```

The body is executed only if the *condition* is true!

If the body of the control flow has only one statement, we can **optionally** not use the { }

```
keyword ( condition )

      body statement 1;
```

Control flow – if

- To execute a particular body of statements only **if** a particular *condition* is satisfied

```
if ( condition ) {

      body statement 1;
            .
            .
            .
      body statement n;

}
```

Example

```
int x = 3, y;

if ( x > 2 ) {

     x++;
     y = x;

}

printf(“y = %d\n”,y);
```

Control flow - else

- To execute a particular body of statements only **if** a particular *condition* is **not** satisfied

```
if ( condition ) {

      body statement 1;
             .
             .
             .
      body statement n;

}
else {

      body statement 1;
             .
             .
             .
      body statement m;

}
```

Example

```
int x = 3, y;

if ( x > 2 ) {

     x++;
     y = x;

}
else {

     y = 2 * x;

}

printf("y = %d\n",y);
```

Control Flow – if/else example

```
int x = 3, y = 1;

if( x > 2 )
     if( x == 4)
            y = x;
else
     y = 2 * x;

printf(“y = %d\n”,y);
```

Control Flow – if/else example

```
int x = 3, y = 1;

if( x > 2 )
     if( x == 4)
           y = x;
else
     y = 2 * x;

printf(“y = %d\n”,y);
```

`else` refers always to the last `if` that was not already closed by another `else`

Control Flow – if/else example

```
int x = 3, y = 1;

if( x > 2 ) {

    if( x == 4) {
        y = x;
    }
    else {
        y = 2 * x;
    }
}

printf("y = %d\n",y);
```

This is why we need brackets and indentation!

Control Flow – if/else example

```
int x = 3, y = 1;

if( x > 2 ) {

    if( x == 4) {
        y = x;
    }
}
else {
    y = 2 * x;
}

printf("y = %d\n",y);
```

Using brackets we can change the `if` to which the `else` refers

Control flow - Switch

Equivalent to a series of if/else statements

```
switch ( variable ) {

       case val1:
               statement 1;
                  .
                  .
                  .
               break;

       case val2:
               statement 1;
                  .
                  .
                  .
             /* fall through */
          .
          .
          .
       default:
               statement 1;
                  .
                  .
                  .
               break;
}
```

```
int i,j;

switch( i ) {

 case 1:
     j  = i + 1;
     break;

 case 10:
     j = i - 1;

 default:
     j = 1;

}
```

Control flow - Switch

Equivalent to a series of if/else statements

```
switch ( variable ) {

      case val1:
              statement 1;
              ⋮
              break;

      case val2:
              statement 1;
              ⋮
          /* fall through */
        ⋮
      default:
              statement 1;
              ⋮
              break;
}
```

These values are CONSTANT

If variable has value different from all other cases

```
int i,j;

switch( i ) {

  case 1:
       j  = i + 1;
       break;

  case 10:
       j = i - 1;

  default:
       j = 1;

}
```

i	j
1	2
10	1
Any other number	1

Control flow - Switch

Equivalent to a series of if/else statements

```
switch ( variable ) {

      case val1:
            statement 1;
                .
                .
                .
            break;

      case val2:
            statement 1;
                .
                .
                .
          /* fall through */
      .
      .
      .
      default:
            statement 1;
                .
                .
                .
            break;
}
```

These values are CONSTANT

If variable has value different from all other cases

```
int i,j;

switch( i ) {

 case 1:
     j  = i + 1;
     break;

 case 10:
     j = i - 1;

 default:
     j = 1;

}
```

After last `case` I can avoid using `break`

Switch

Equivalent to a series of if/else statements

```
switch ( variable ) {

        case val1:
                statement 1;
                  .
                  .
                  .
                break;

        case val2:
                statement 1;
                  .
                  .
                  .
             /* fall through */
           .
           .
           .
        default:
                statement 1;
                  .
                  .
                  .
                break;
}
```

```
int i,j;

switch( i ) {

 case 1:
       j  = i + 1;
       break;

 case 10:
       j = i - 1;

 default:
       j = 1;

}
```

variable can only be `char` or `int` !

~~float i = 2;~~

~~switch(i) {~~

Control Flow - Loops

- What if we want to perform the same operation multiple times?
- Example: we want to initialize all elements in a 100 dimensional array of integers to the value 7

```
int arr[100];

arr[0] = 7;
arr[1] = 7;
arr[2] = 7;
arr[3] = 7;
    .
    .
    .
arr[99] = 7;
```

This is crazy!

Loops - while

- To execute a particular body of statements only **until** a particular *condition* is satisfied

```
while ( condition ) {

     body statement 1;
           .
           .
           .
     body statement n;

}
```

Example

```
int i = 0;
int arr[100];

while( i < 100 ) {

   arr[i] = 7;
   i++;

}
```

Loops – do/while

- **First** execute body statements, **then** check if *condition* is satisfied

```
do {

   body statement 1;
             .
             .
             .
   body statement n;

} while ( condition );
```

<u>Example</u>

```
int i = 10,
int j = 0;

while( i < 10 )
{

   j++;
   i++;

}
```

<u>Example</u>

```
int i = 10;
int j = 0;

do
{

   j++;
   i++;

} while( i < 10 );
```

j = ?

Loops – do/while

- **First** execute body of statements, **then** check if *condition* is satisfied

```
do {

   body statement 1;
             .
             .
             .
   body statement n;

} while ( condition );
```

Example

```
int i = 10,
int j = 0;

while( i < 10 )
{

   j++;
   i++;

}
```

j = 0

Example

```
int i = 10;
int j = 0;

do
{

   j++;
   i++;

} while( i < 10 );
```

j = 1

Loops - break

- To interrupt a loop once a certain condition different from the one in the loop declaration

```
while( condition1 ){

    body statement 1;
          .
          .
          .
    if( condition2 )
      break;
          .
          .
          .
    body statement n;

}
```

When **break** is reached, the statements after it are ignored and the program exits the loop

Example

```
int i = 0;
char s[10] = “hi”;

while( i < 10 )
{

   if(s[i]==‘\0’)
       break;

   printf(“%c”,s[i]);

   i++;

}
```

Loops - continue

- To ignore the following instructions in a loop

```
while( condition1 ){

    body statement 1;
              .
              .
              .
    if( condition2 )
       continue;
              .
              .
              .
    body statement n;

}
```

When **continue** is reached, the statements after it are ignored, and the loop continues

Example

```
int i = 0, sum = 0;
int s[3] = {7, 5, 9};

while( i < 3 )
{

    if(s[i] < 6)
         continue;

    sum += s[i];
}
```

break vs. continue

```
int  x = 0, y = 0;

while( x < 10) {

     x++;

     if(x == 3) {

        continue;
    }

    y++;
}
```

```
int  x = 0, y = 0;

while( x < 10) {

    x++;

    if(x == 3) {

         break;
    }

    y++;
}
```

y = ?

break vs. continue

```c
int  x = 0, y = 0;

while( x < 10) {

     x++;

     if(x == 3) {

        continue;
    }

    y++;
}
```

y = 9

```c
int  x = 0, y = 0;

while( x < 10) {

    x++;

    if(x == 3) {

         break;
    }

    y++;
}
```

y = 2

Loops - for

```
for (initial state ; condition ; state change ) {

     body statement 1;
               .
               .
               .
     body statement n;

}
```

Example

```
int i;
int arr[100];

for( i = 0; i < 100 ; i++ ) {
   arr[i] = 7;
}
```

```
int i = 0;
int arr[100];

while( i < 100 ) {
   arr[i] = 7;
   i++;
}
```

Homework 1 review

HOW TO COMPRESS/UNCOMPRESS folders in UNIX

- Compress folder ~/COMS1003/HW1 to HW1.tar.gz

```
tar -zcvf HW1.tar.gz  ~/COMS1003/HW1
```

- Uncompress HW1.tar.gz to folder ~/COMS1003/HW1new

```
tar -zxvf HW1.tar.gz  -C  ~/COMS1003/HW1new
```

(note: ~/COMS1003/HW1new must exist already)

COMsW 1003-1

Introduction to Computer Programming in C

Lecture 7

Spring2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Today

- Loops (from Lec6)
- Scope of variables
- Functions

Scope of Variables

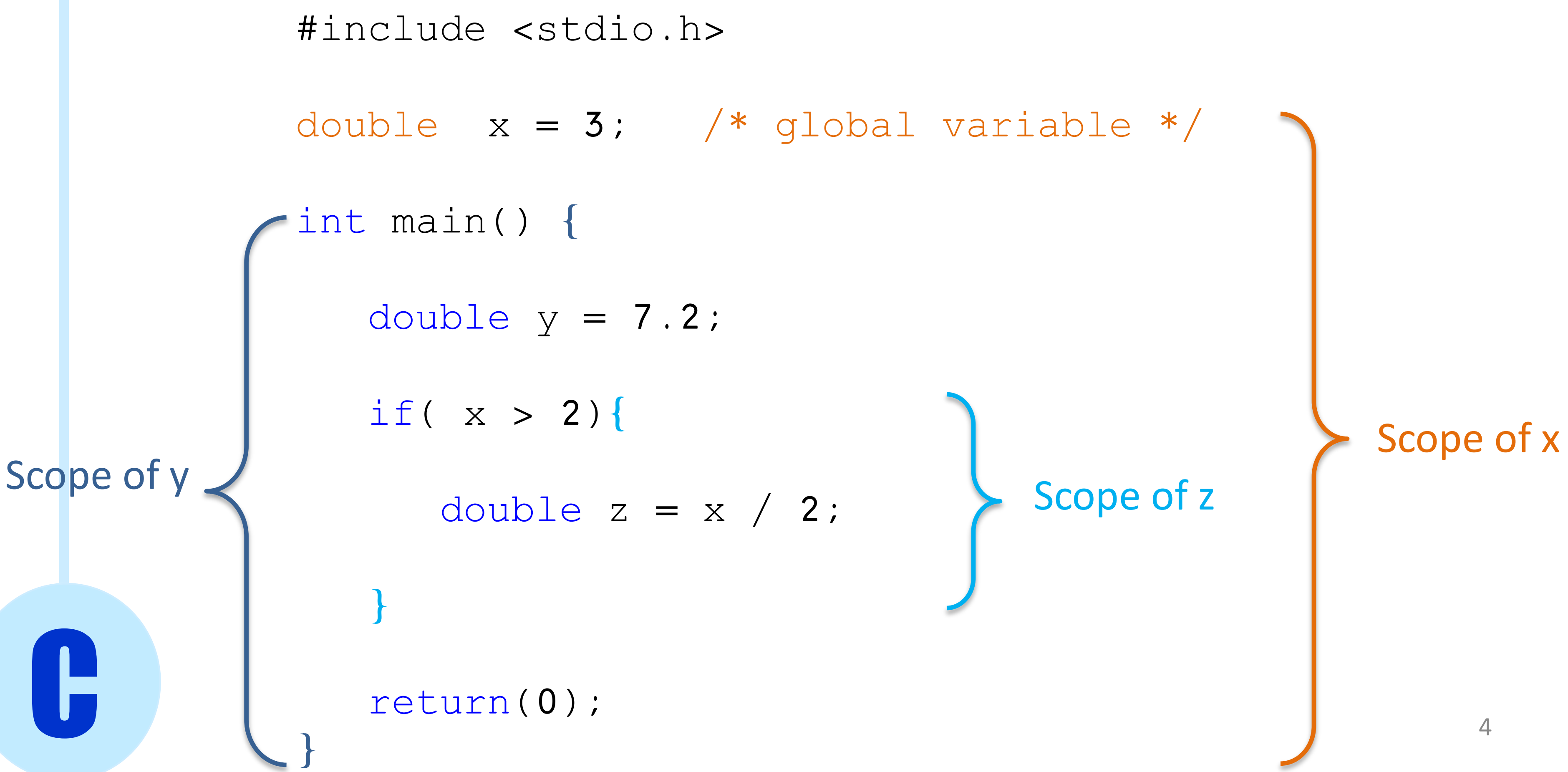
- **Scope** is the portion of program in which a variable is valid

- Depends on where the variable is **declared**

- Variables can be
  - **Global** : valid everywhere
  - **Local** : valid in a specific portion of the program included in { }

Scope of Variables

- **Scope** is the portion of program in which a variable is valid
- Depends on where the variable is **declared**
- Variables can be
  - **Global** : valid everywhere
  - **Local** : valid in a specific portion of the program included in { }



```
#include <stdio.h>

double  x = 3;   /* global variable */

int main() {

   double y = 7.2;

   if( x > 2){

      double z = x / 2;

   }

   return(0);
}
```

Scope of x

Scope of y

Scope of z

Scope of variables

```
#include <stdio.h>

double  z = 1;

int main() {
   printf(”z1 = %lf\n”, z);          // z1 = 1.0000000
   double z = 7;
   if( z > 2){
      double z =  0.5;
      printf(”z2 = %lf\n”, z);      // z2 = 0.5000000
   }
   printf(”z3 = %lf\n”, z);         // z3 = 7.0000000
   {
     double z = 11;
     printf(”z4 = %lf\n”,z);        // z4 = 11.0000000
   }
   printf(”z5 = %lf\n”,z);           // z5 = 7.0000000
   return(0);
}
```

Scope of variables

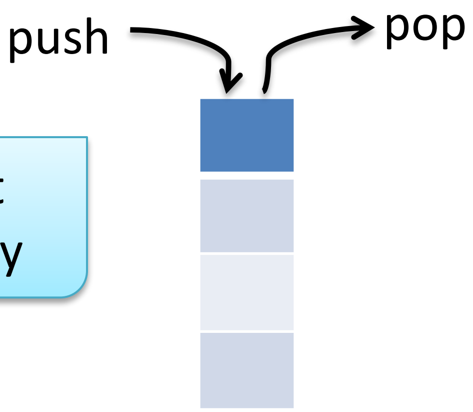
```
#include <stdio.h>

double  z = 1;

int main() {
   printf(”z1 = %lf\n”, z);           // z1 = 1.0000000
   double z = 7;
   if( z > 2){
      double z =  0.5;
      printf(”z2 = %lf\n”, z);     // z2 = 0.5000000
   }
   printf(”z3 = %lf\n”, z);          // z3 = 7.0000000
   {
     double z = 11;
     printf(”z4 = %lf\n”,z);         // z4 = 11.0000000
   }
   printf(”z5 = %lf\n”,z);           // z5 = 7.0000000
   return(0);
}
```

Class of Variables

- A variable can be either
  - **Temporary** : allocated in stack at beginning of block (if too many local variables allocated, stack overflow)
  - **Permanent** : allocated before the program starts

- **Global** variables are always **permanent**

- **Local** variables are **temporary** unless they are declared **static**



Stack: First In Last Out (FILO) type of memory

Variables – Scope and Class

Declared	**Scope**	**Class**	**initialized**
Outside all blocks	Global	Permanent	Once
Static outside all blocks	Global	Permanent	Once
Inside a block	Local	Temporary	Each time block is entered
Static inside a block	Local	Permanent	Once

From PCP Ch 9

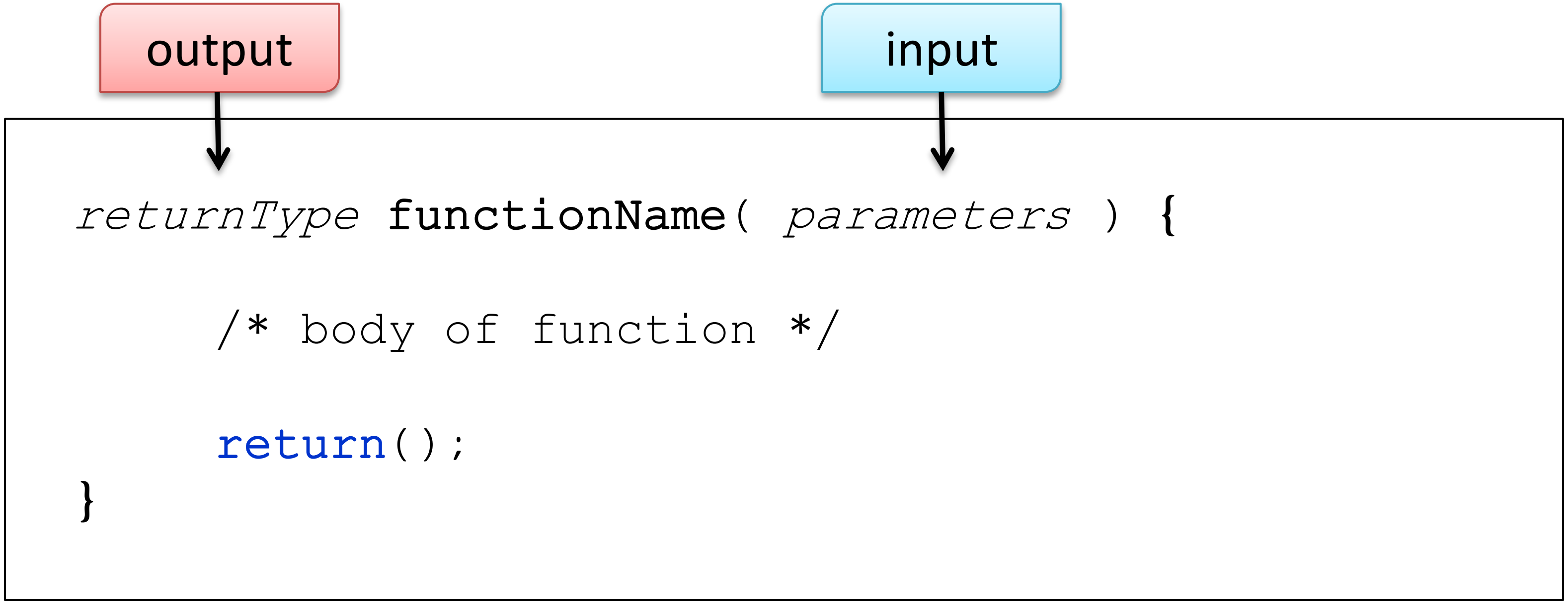
```
#include <stdio.h>

int z = 0;
static int b;

int main() {

  int g =  0;

  while( z < 3){

      int y =  0;
      static int x = 0;

      y++;
      x++;
      z++;

      printf("x = %d, y = %d, z = %d\n", x, y, z);

  }
  return(0);
}
```

```
x = 1, y = 1, z = 1
x = 2, y = 1, z = 2
x = 3, y = 1, z = 3
```

y is initialized every time

Functions

- Functions allow to write and reuse pieces of code that accomplish a task
- Help keeping large codes ordered



output

input

```
returnType functionName( parameters ) {

     /* body of function */

     return();
}
```

Functions - Example

The function *sumTwoNumbers* takes two numbers as input and returns their sum.

```
double sumTwoNumbers( double n1, double n2 ) {

      double s;

      s = n1 + n2;

      return(s);
}
```

Functions - Example

The function *sumTwoNumbers* takes two numbers as input and returns their sum.

```
double sumTwoNumbers( double n1, double n2 ) {

    double s;

    s = n1 + n2;

    return(s);            // return s;
}
```

Returned type must be consistent!

These two notations are equivalent

sumNumbers.c

Functions – Example

```
#include <stdio.h>
double sumTwoNumbers( double n1, double n2 ){
      double s;
      n1++;
      s = n1 + n2;
      return(s);
}
int main() {
      double x, y, z;
       x = 2;
       y = 2;
       z = sumTwoNumbers(x, y);
       printf(“%f + %f = %f\n”, x, y, z);
       return(0);
}
```

Function Declaration must happen BEFORE its use in the main() function

Scope of n1 and n2

is scope of function!

2 + 2 = 5 !

Functions - void

printArrow.c

- If a function does not take any input
- If a function does not return any value

```
/* function to print an arrow to command line */
void printArrow(void){

       /* function body */

       return;
}

/* function to print multiple arrows to command line */
void printMultipleArrows(int nTimes){

       int i;

       for(i = 0; i < nTimes; i++){

           printArrow();
        }

       return;
}


int main() {

     int x = 3;

     printMultipleArrows(x);

     return(0);

}
```

Functions - void

printArrow.c

- If a function does not take any input
- If a function does not return any value

```c
/* function to print an arrow to command line */
void printArrow(void){

       /* function body */

       return;
}

/* function to print multiple arrows to command line */
void printMultipleArrows(int nTimes){

       int i;

       for(i = 0; i < nTimes; i++){

           printArrow();
        }

       return;
}

int main() {

      int x = 3;

      printMultipleArrows(x);

      return(0);

}
```

Function does not return anything

Function invoked without passing any parameter ()

Functions - void

printArrow.c

- If a function does not take any input
- If a function does not return any value

```
/* function to print an arrow to comman
void printArrow(void){

        /* function body */

        return;
}

/* function to print multiple arrows to command line */
void printMultipleArrows(int nTimes){

        int i;

        for(i = 0; i < nTimes; i++){

            printArrow();
         }

        return;
}


int main() {

      int x = 3;

      printMultipleArrows(x);

      return(0);

}
```

Return can be viewed as equivalent of break for functions

Function is declared before being used

Functions – Passing Arrays

length.c

```
/* function to compute the length of a string*/
int length( char s[] ){

     int size = 0;

     while(s[size] != '\0'){
          size++;
     }

     return size;
}

/* function to copy a string*/
char[] copyString( char s[] ){

      char s2[100];

      strcpy(s2, s);

      return s2;
}
```

Functions – Passing Arrays

length.c

```
/* function to compute the length of a string*/
 int length( char s[] ){

        int size = 0;

        while(s[size] != '\0'){
              size++;
        }

        return size;
 }

 /* function to copy a string*/
  char[] copyString( char s[] ){

         char s2[100];

         strcpy(s2, s);

         return s2;
  }
```

Functions – exit()

`exit()` is used to exit (=terminate) the program

Different from return, which simply exits the function

Exit() is defined inside the library stdlib.h

```
#include <stdlib.h>

 int length( char s[] ){

        int size = 0;

        while(s[size] != ‘\0’){

               if(s[size] == ‘m’)
                      exit(-1);

               size++;
        }

        return size;
}
```

COMsW 1003-1

Introduction to Computer Programming in C

Lecture 8

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Announcements

Homework 1 correction out this afternoon

Homework 2 is out

- Due Monday, February 28th
- Start early (especially Exercise 2)!

Today

- Functions
- Recursion
- Debugging (if time)

Infinite Loops

- Loops where the condition is always TRUE

- Will stop only with:
  - `break`
  - modification of the condition variables

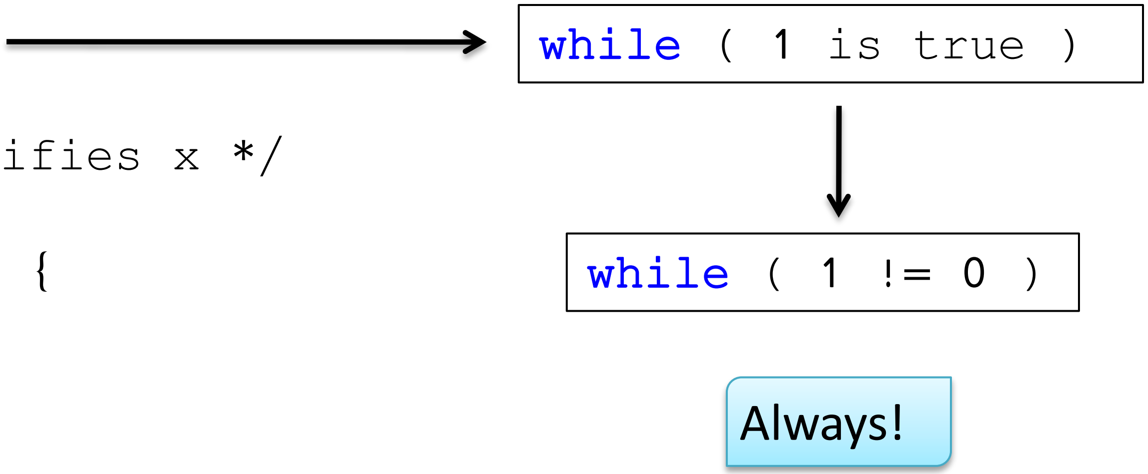
```
while ( 1 ){

   /* body modifies x */

   if( x!= 0 ) {
         break;
   }

}
```

Infinite Loops

- Loops where the condition is always TRUE

- Will stop only with:
  - `break`
  - modification of the condition variables

```
while ( 1 ){

   /* body modifies x */

   if( x!= 0 ) {
         break;
   }

}
```



`while ( 1 is true )`

`while ( 1 != 0 )`

Always!

Operators - Logical

- A variable with value 0 is false, a variable with value !=0 is true

`int x = 3, y = 0, z, k, t, q = -3;`

`z = x && y;    // z = 0;` *x is true but y is false*

`k = x || y;    // k = 1;` *x is true*

`t = !q;        // t = 0;` *q is true*

Infinite Loops

- Loops where the condition is always TRUE

- Will stop only with:
  - `break`
  - modification of the condition variables

```
int cond = 7;

while ( cond ){

   /* body */
   if( x[3][5] != 7 ){
      cond = 0;
   }

}
```

`while ( cond is true )`

`while ( cond != 0 )`

Until we set `cond` to 0!

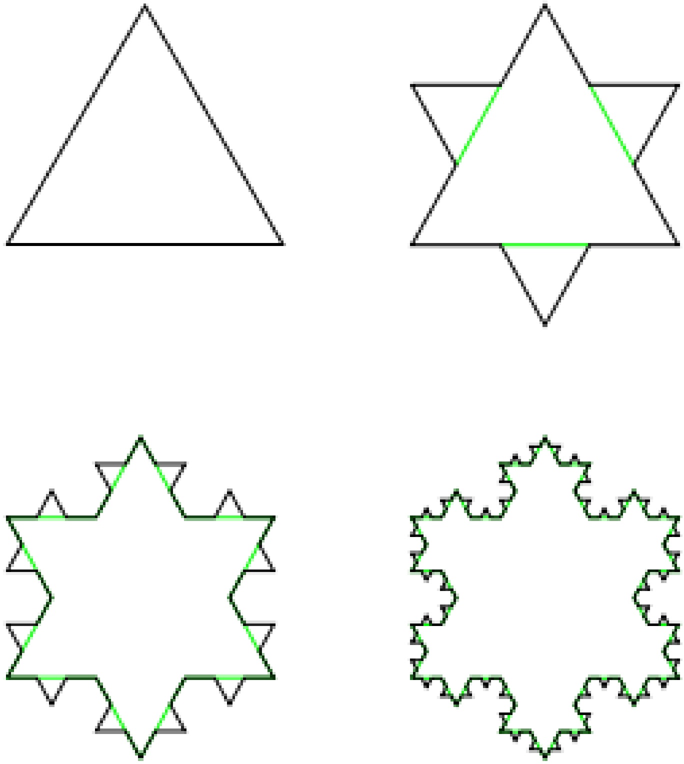
calculator.c

Functions Example

- Simple calculator
- Program that computes one basic arithmetic operation between 2 numbers

Functions - Recursion

- What if a function calls itself? Recursion



Functions - Recursion

- A recursive function must have two properties:
  - **Ending point** (i.e. a terminating condition)
  - **Simplify the problem** (every call is to a simpler input)

Example: Fibonacci sequence

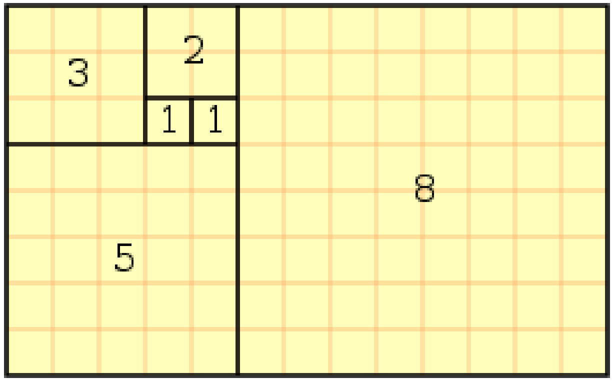
In mathematics, famous numbers following the sequence

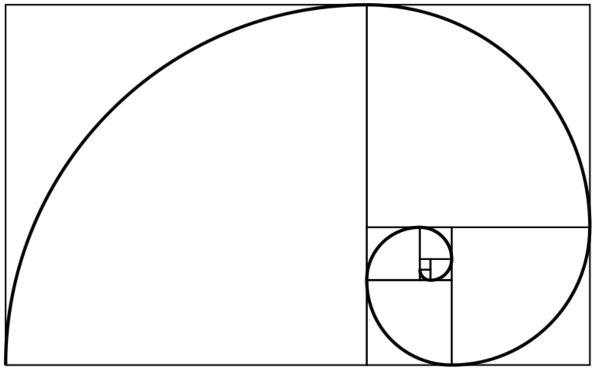
0 1 1 2 3 5 8 13 21 34 55 89 …

Given $F_0 = 0$, $F_1 = 1$ can be computed with recurrence $\boxed{F_n = F_{n-1} + F_{n-2}}$

Code to compute the first 100 Fibonacci numbers:

```
int i = 0;
int fib[100];

fib[0] = 0;
fib[1] = 1;

for( i = 2; i < 100 ; i++ ) {
    fib[i] = fib[i-1] + fib[i-2];
}
```





Functions - Recursion

recursiveFib.c

- What if a function calls itself? Recursion
- What is the value of the number at position num in the Fibonacci sequence?

```
/* Fibonacci value of a given position in the sequence */
int fib ( int num ) {

  switch(num) {
   case 0:
       return(0);

   case 1:
       return(1);

   default: /* Including recursive calls */
      return(fib(num - 1) + fib(num - 2));

  }
}
```

Why are there no `breaks` ?

Functions - Recursion

recursiveFib.c

- What if a function calls itself? Recursion
- What is the value of the number at position num in the Fibonacci sequence?

```
/* Fibonacci value of a given position in the sequence */
int fib ( int num ) {

  switch(num) {
   case 0:
        return(0);

   case 1:
        return(1);

   default: /* Including recursive calls */
       return(fib(num - 1) + fib(num - 2));

  }
}
```

Ending Points

Simplify problem

Debugging

Debugging

- Debugging consists basically in finding and correcting **run-time errors** in your program

- Multiple ways of doing it
  - Manual runs (for small programs)
  - Insert `printf()` in key lines

- There also exist INTERACTIVE debugging tools

- We will now see a basic one for UNIX: **gdb**

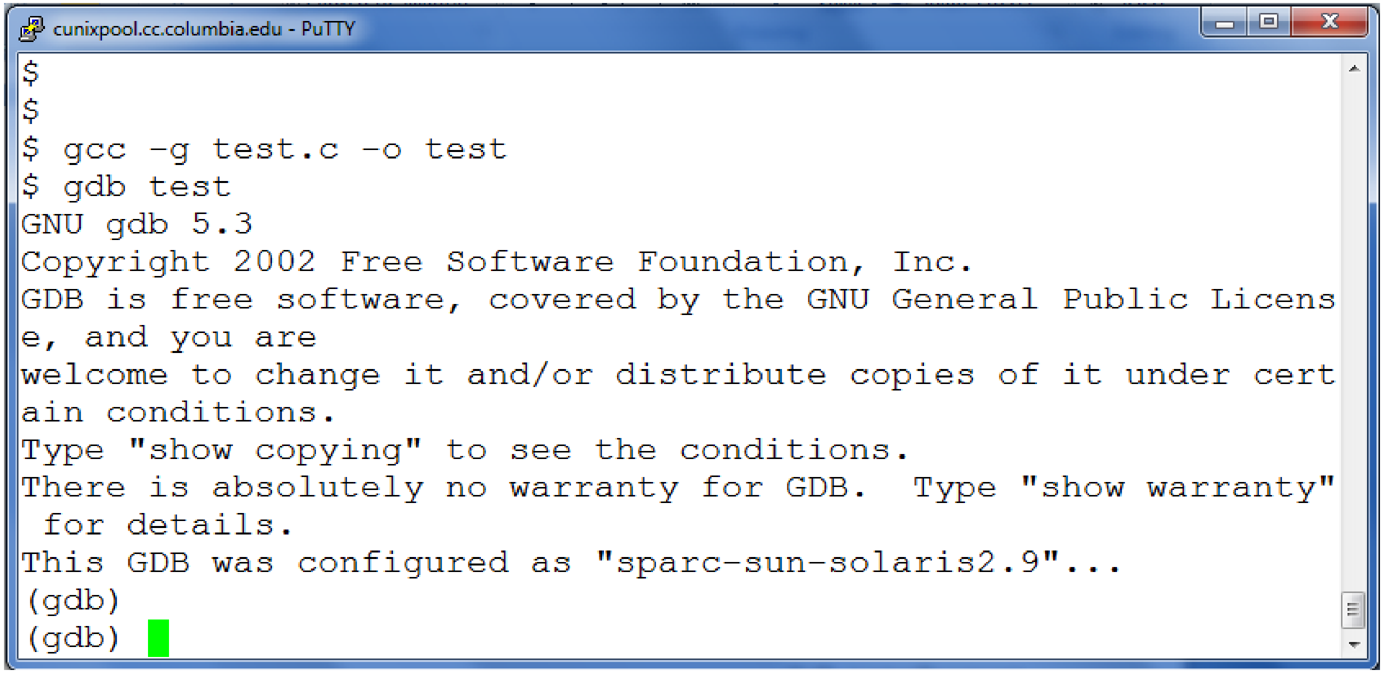
gdb

1. In order to use gdb on a program, we must use the **–g** option when compiling it

```
gcc  -g program.c   -Wall -o nameOfExecutable
```

2. Then, we can use the gdb command to start the interactive debugging environment

```
gdb nameOfExecutable
```



cunixpool.cc.columbia.edu - PuTTY

```
$
$
1. $ gcc -g test.c -o test
2. $ gdb test
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public Licens
e, and you are
welcome to change it and/or distribute copies of it under cert
ain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"
 for details.
This GDB was configured as "sparc-sun-solaris2.9"...
(gdb)
(gdb) 
```

gdb commands

- **run** : run executable (program)currently watched.

```
(gdb) run
```

- **kill** : kill current execution of program

```
(gdb) kill
```

- **list** : show program source code

`(gdb) list 2,8` : shows lines 2 to 8 from source program

- **print** : print value of a variable or expression at the current point

```
(gdb) print buf
```

gdb commands

- **break** : insert breakpoint in program. Debugging run will stop at the breakpoint

```
(gdb) break  nameSource.c : lineNumber
(gdb) break  test.c: 12
```

- **next** : step to the next line (execute current line)

```
(gdb) next
```

- **continue** : continue with execution until next breakpoint or end of program

```
(gdb) continue
```

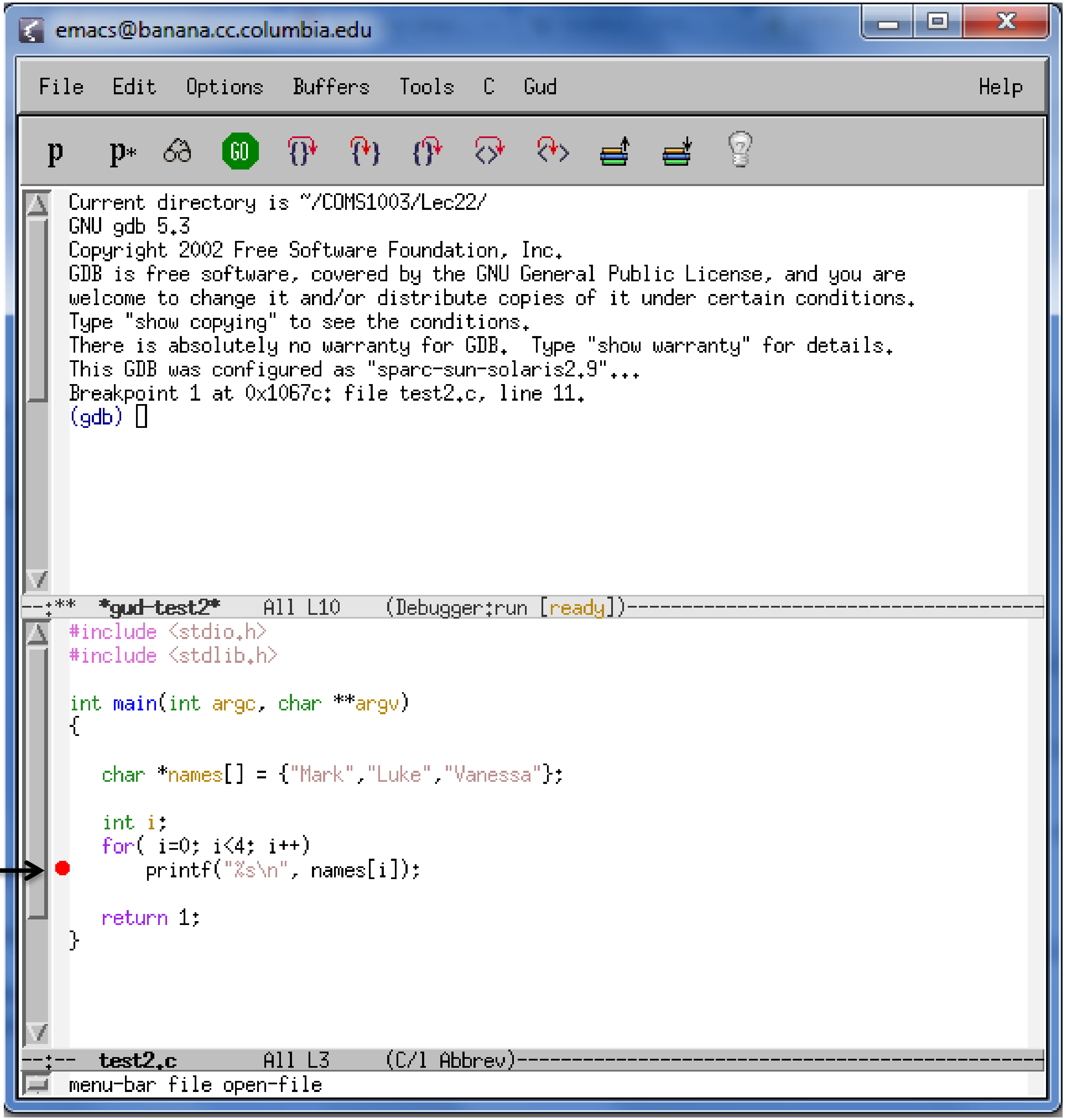
- **Quit** : exit gdb

```
(gdb) quit
```

Graphical GDB

- gdb can be run from Emacs

- Press `M-x` (in Windows `Esc-x`)
- Insert `gdb`
- Insert `executableName`
- Visual debugger



```
Current directory is ~/COMS1003/Lec22/
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.9"...
Breakpoint 1 at 0x1067c: file test2.c, line 11.
(gdb) 
--:**  *gud-test2*     All L10     (Debugger:run [ready])------------------------
```

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

int main(int argc, char **argv)
{

   char *names[] = {"Mark","Luke","Vanessa"};

   int i;
   for( i=0; i<4; i++)
       printf("%s\n", names[i]);

   return 1;
}
--:--  test2.c        All L3      (C/l Abbrev)----------------------------------
menu-bar file open-file
```

Can enable breakpoints with a click

COMsW 1003-1

Introduction to Computer Programming in C

Lecture 9

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Are Computers Smarter than Humans?

IBM's Watson on 'Jeopardy': Computer takes big lead over humans in Round 2

February 15, 2011 | 9:20 pm (0) (0) Comments (0)

On Tuesday night's "Jeopardy" episode, Watson, the IBM supercomputer, steamrollered to a commanding lead over his human competitors.

[Link](http://latimesblogs.latimes.com/technology/2011/02/ibms-watson-on-jeopardy-computer-takes-big-lead-over-humans-in-round-2.html)

http://latimesblogs.latimes.com/technology/2011/02/ibms-watson-on-jeopardy-computer-takes-big-lead-over-humans-in-round-2.html

Today

- Homework 1 Correction
- Debugging (from Lecture 8)
- C Preprocessor

Conditional Assignment

- Another way of embedding `if - else` in a single statement
- Uses the `?` `:` operators

```
variable = ( condition ) ? val1 : val2 ;
```

If condition is true, we assign val1 to variable

If condition is false, we assign val2 to variable

```
int x = 7, y;

y =  ( x > 5 ) ? x : 5;
```

y= 7

⟺

```
int x = 7, y;

if( x > 5 ) {
    y = x;
}
else{
    y = 5;
}
```

The comma operator

- In C statements can also be separated by , not only ;

```
int x = 2;
int y;

x++, y = x/3, y += 2;
```

```
int x = 2;
int y;

x++;
y = x/3;
y = y+2;
```

Be careful with declarations!

`int x = 2, char c = ‘m’;` ✗ Different types, NO

`int x = 2, y;` V Same type, OK

The comma operator

Special case, the `for` loop statement

Example: the palindrome word checking. Check if a word is the same when read right to left

```
int i, flag = 1;

char word[100] = “radar”;

for( i=0 , j=strlen(word)-1 ;  i < strlen(word)/2 ;  i++ , j-- ) {

    if( word[i] != word[j] ) {
            flag = 0;
            break;
    }
}
```

The comma operator

Special case, the `for` loop statement

Example: the palindrome word checking

Initial conditions

change conditions

```
for( i=0 , j=strlen(word)-1 ;  i < strlen(word)/2 ;  i++ , j-- ) {
    if( word[i] != word[j] ) {
        flag = 0;
        break;
    }
}
```

Advanced Types - Const

`const` defines a variable whose value cannot be changed

```
const double PI = 3.14;

double r = 5, circ;

circ = 2 * PI * r;

PI = 7;
```

Advanced Types - Const

`const` defines a variable whose value cannot be changed

```
const double PI = 3.14;

double r = 5, circ;

circ = 2 * PI * r;
```

~~PI = 7~~;

Once it’s initialized, a const variable cannot change value

C Preprocessor

C Preprocessor

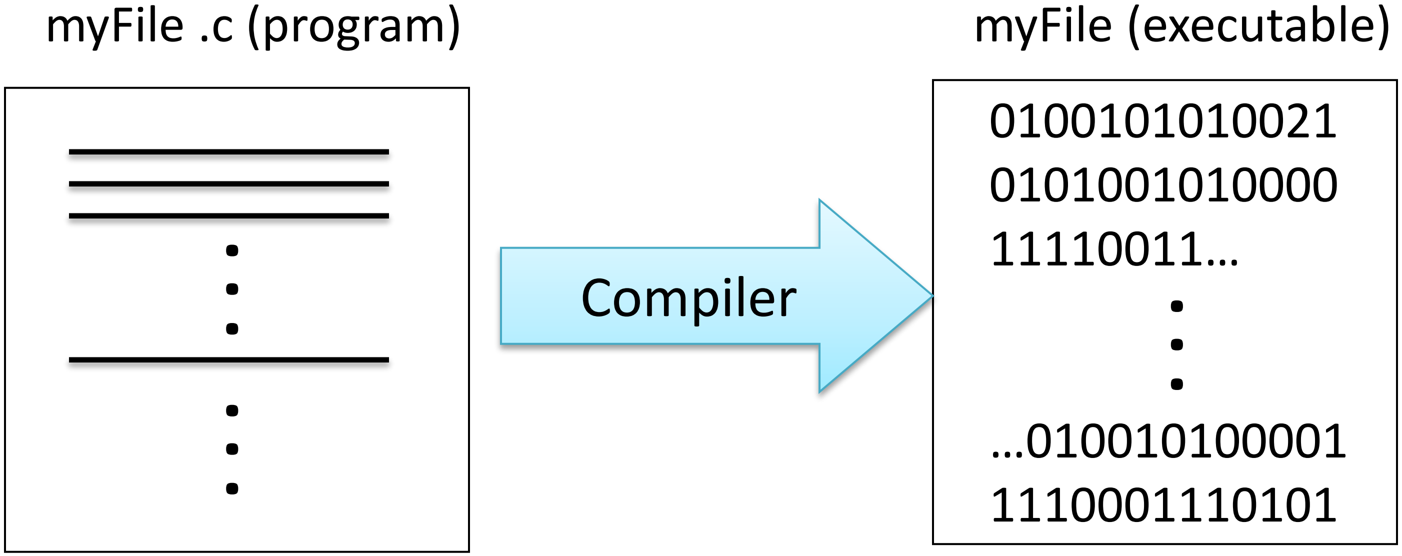
Preprocessor is a facility to handle

- **Header files**
- **Macros**

Independent from C itself, it’s basically a text editor that modifies your code before compiling

Preprocessor statements begin with **#** and do **not** end with ;

C Preprocessor



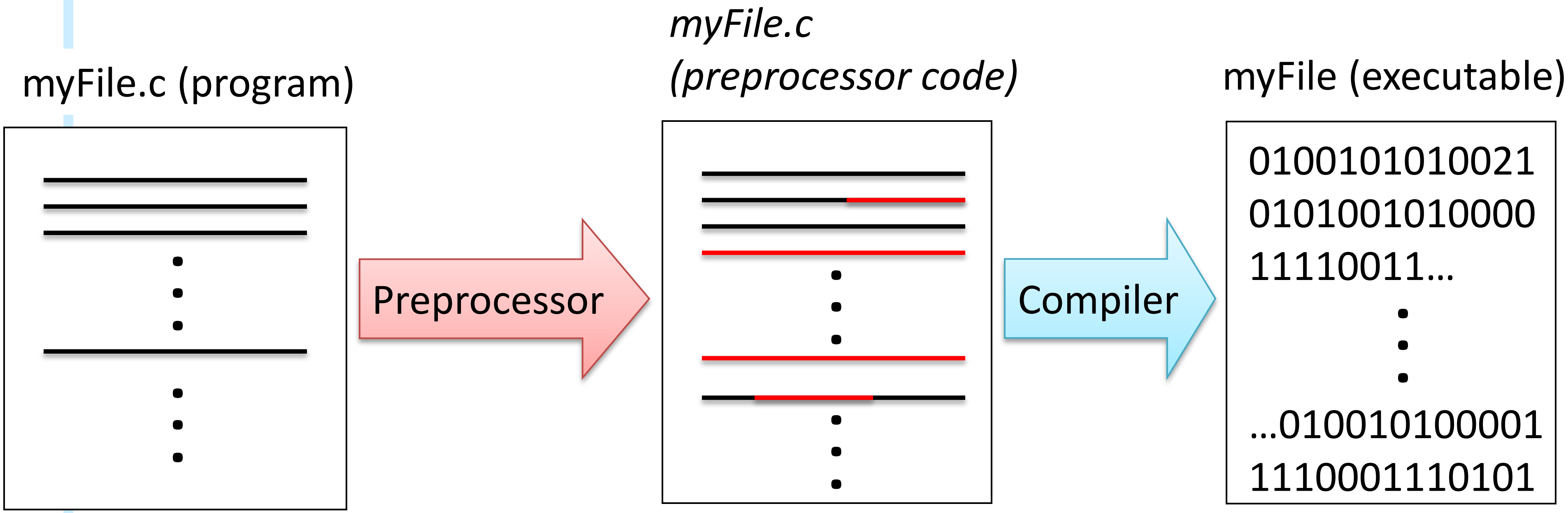
myFile .c (program)

Compiler

myFile (executable)

0100101010021
0101001010000
11110011…
⋮
…010010100001
1110001110101

C Preprocessor



myFile.c (program)

Preprocessor

myFile.c
(preprocessor code)

Compiler

myFile (executable)

0100101010021
0101001010000
11110011…
⋮
…010010100001
1110001110101

View Preprocessor Code

gcc has a special option that allows to run only the preprocessor

```
gcc -E myFile.c
```

We can send output to a file using the UNIX **>** operator

```
gcc -E myFile.c   >  outFile.txt
```

Saves gcc’s output to outFile.txt

Header files

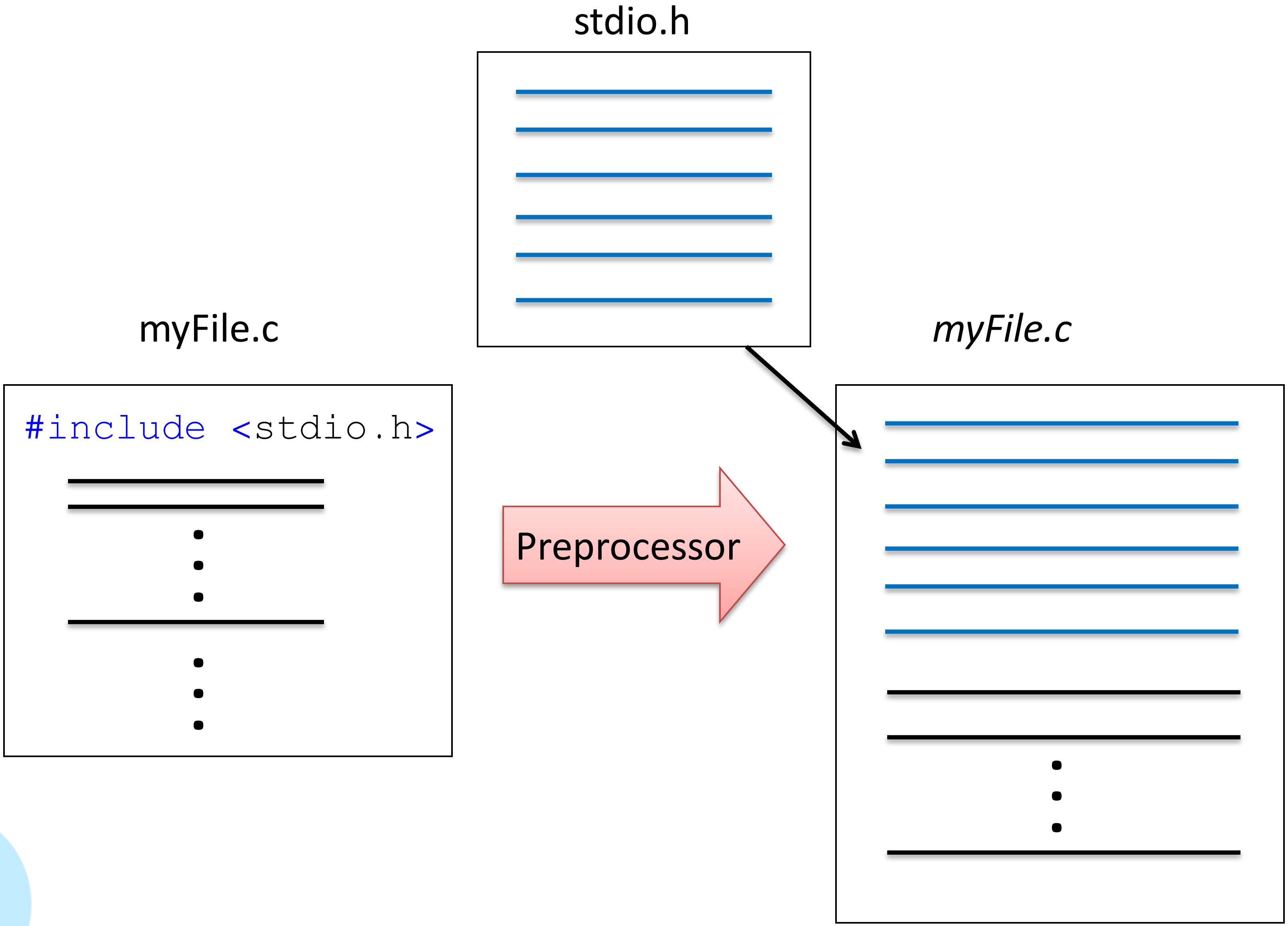
- Header files are fundamentally libraries
- Their extension is .h
- They contain function definitions, variables declarations, macros
- In order to use them, the preprocessor uses the following code

`#include <nameOfHeader.h>` → For standard C libraries

`#include “nameOfHeader.h”` → For user defined headers

- So far, we have used predefined C header files, but we can create our own! (more on this in upcoming Lectures)

Header files



stdio.h

myFile.c

`#include <stdio.h>`

Preprocessor

myFile.c

Macros

- A **macro** is a piece of code ***c*** which has been given a name ***n***

- Every time we use that ***n*** in our program, it gets replaced with ***c***

- The preprocessor allows you to declare them with `#define`

- Two types:
  - Object-like macros
  - Function-like macros

macros.c

Object like macros

- Constants, usually defined on top of programs

```
#define name text_to_substitute
```

```
#define SIZE 10

#define FOR_ALL for( i=0; i< SIZE; i++ )
```

macros.c

Object like macros

```
#define SIZE 10

/* main function */
int main(){

   int arr[SIZE];

   return(0);
}
```

From now on, every time we write SIZE inside our program it is going to be replaced by 10

Object like macros

macros.c

- Some compilers do not allow you to declare arrays with a variable as size

```
int size1 = 10;
int arr1[size1]; /* should always cause error */

const int size2=10;
int arr2[size2]; /* causes errors in many compilers */

#define SIZE 10
int arr3[SIZE]; /* OK in any C compiler */
```

macros.c

Function-like macros

- Macros that can take parameters like functions

```
#define SQR(x) ((x) * (x))

#define MAX(x,y) ((x) > (y) ? (x) : (y))
```

- Parameters MUST be included in parentheses in the macro name, without spaces
- It is a good habit to include parameters in parentheses also in the text to be substituted

condComp.c

Conditional Compilation

- Allows to use or not certain parts of a program based on definitions of macros

`#ifdef var`	if `var` is defined, consider the following code
`#ifndef var`	if `var` is not defined, consider the following code
`#else`	
`#endif`	close `if(n)def`
`#undef var`	undefine `var` (opposite of `#define`)

condComp.c

Conditional Compilation

```
#define DEBUG
        .
        .
        .
#ifdef DEBUG

printf(“The value of x is %d\n”, x);

#endif
```

If `DEBUG` was defined earlier in the program, then the statement `printf(…);` is considered, otherwise the preprocessor does not copy it to the file to be compiled

COMSW 1003-1

Introduction to Computer Programming in C

Lecture 10

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Announcements

Change in Office Hours this week

1 hour Wednesday, Feb 23rd, 12pm-1pm

1 hour Saturday, Feb 26th, 11am-12pm

Today

- Preprocessor (from Lecture 9)
- Advanced C Types

students.c

Advanced Types - Struct

- Arrays group variables of the **same** type
- Structs group variables of **different** types

Struct definition

```
 struct structName {

    fieldType fieldNameval1;
    fieldType fieldNameval2;
            ⋮
    fieldType fieldNamevalN;
};
```

Once we define the struct, we can use `structName` as if were a type, to create variables!

Advanced Types - Struct

Example: we want to build a database with the name, age and grade of the students in the class

Student 1
Name:
Age:
Grade:

Student 2
Name:
Age:
Grade:

. . .

Student N
Name:
Age:
Grade:

```
struct student {

    char name[100];
    int  age;
    double grade;
};

struct student st1;
```

st1 is a variable of type struct!

Advanced Types - Struct

In order to access struct fields, we need to use the . operator

```
struct student {
    char name[100];
    int  age;
    double grade;
};

struct student st1, st2;

st1.age = 3;
st2.age = st1.age - 10;
```

st1.age is a variable of type `int`, I can use it as a regular variable !

Advanced Types - Struct

We can initialize a struct variable at declaration time, just like with arrays

```
struct student {
    char name[100];
    int  age;
    double grade;
};
```

The initialization fields must be consistent with the fields types !

char int double

```
struct student st1 = {“mike”, 22, 77.4};
```

Advanced types - Typedef

typedef is used to define a new type

```
typedef type nameOfNewType;
```

```
typedef int myInt;
myInt c = 3;
```

C is of type `myInt`, which is equivalent to `int`

```
typedef int myIntArray[7];
myIntArray arr;
```

arr is of type `myIntArray`, which is equivalent to an array of 7 `int`

```
for(c=0; c<7; c++){
  arr[c] = 1;
}
```

Advanced types - Typedef

typedef is used to define a new type

```
struct student {
   char name[100];
   int  age;
   double grade;
};

struct student st1, st2;

st1.age = 3;
st2.age = st1.age - 10;
```

```
struct student {
    char name[100];
    int  age;
    double grade;
 };

 typedef struct student stud;
 stud st1, st2;

 st1.age = 3;
 st2.age = st1.age - 10;
```

unions.c

Advanced Types - Union

- Similar to struct, but all fields share same memory
- Same location can be given many different field names

```
struct value{
    int   iVal;
    float fVal;
};
```

```
union value{
    int   iVal;
    float fVal;
};
```

We can use the fields of the union only one at a time!

Advanced Types - Enum

- Designed for variables containing only a limited set of values
- Defines a set of **named integer constants**, starting from 0

```
enum name{ item1, item2, … , itemN};
```

```
                0       1    2       3      4      5        6
enum dwarf { BASHFUL, DOC, DOPEY, GRUMPY, HAPPY, SLEEPY, SNEEZY};

enum dwarf myDwarf = SLEEPY;

myDwarf = 1 + HAPPY;   // myDwarf  = SLEEPY = 5;

int x = GRUMPY + 1;    // x = 4;

printf(”dwarf %d\n”,BASHFUL); // ‘dwarf 0’
```

circc.c

Advanced Types - Const

`const` defines a variable whose value cannot be changed

```
const double PI = 3.14;

double r = 5, circ;

circ = 2 * PI * r;

PI = 7;
```

circ.c

Advanced Types - Const

`const` defines a variable whose value cannot be changed

```
const double PI = 3.14;

double r = 5, circ;

circ = 2 * PI * r;
```

~~PI = 7~~;

Once it’s initialized, a const variable cannot change value

circ.c

Advanced Types - Const

`const` defines a variable whose value cannot be changed

```
double computeCirc( const double r, const double PI){

   r++; PI++; ✗

   return(2 * r * PI);

}

/* main function */
int main(){

  const double PI = 3.14;

  double r = 5, circ, circ2;

  circ = 2 * PI * r;
  circ2 = computeCirc(r, PI);

  return 0;

}
```

circ.c

Advanced Types - Const

`const` defines a variable whose value cannot be changed

```
double computeCirc( double r, const double PI){

   r++;   V

   PI++;  ✗

   return(2 * r * PI);

}

/* main function */
int main(){

  const double PI = 3.14;

  double r = 5, circ, circ2;

  circ = 2 * PI * r;
  circ2 = computeCirc(r, PI);

  return 0;

}
```

circ.c

Advanced Types - Const

`const` defines a variable whose value cannot be changed

```
double computeCirc( double r, double PI){

   r++;    V

   PI++;   V

   return(2 * r * PI);

}

/* main function */
int main(){

  const double PI = 3.14;

  double r = 5, circ, circ2;

  circ = 2 * PI * r;
  circ2 = computeCirc(r, PI);

  return 0;

}
```

COMsW 1003-1

Introduction to Computer Programming in C

Lecture 11

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Announcements

- Grades for Homework 1 posted on Courseworks

- Homework 2 is due next Monday at the beginning of class

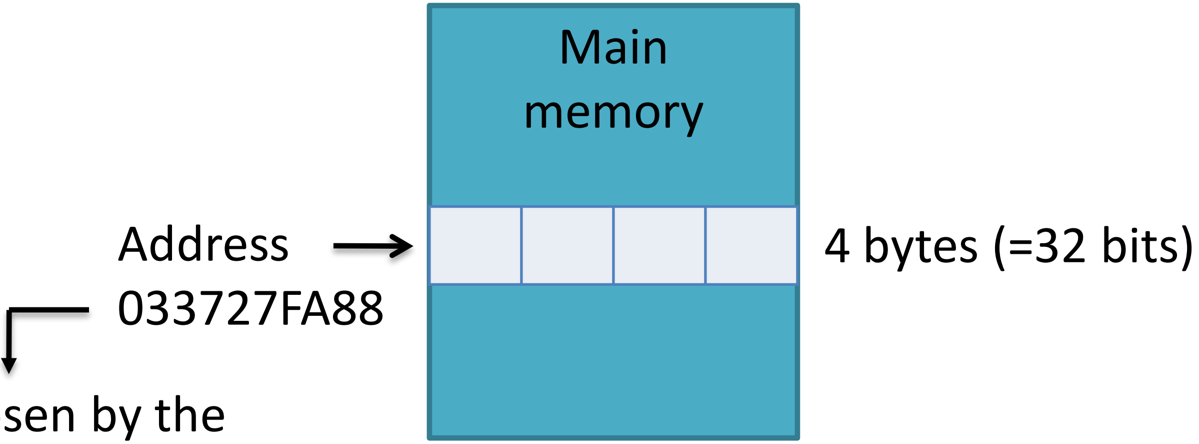
- Bring the printout to class!

Pointers

Pointers

Remember what happens when we declare a variable: the computer allocates memory for it.

```
int x;
```



Main memory

Address 033727FA88

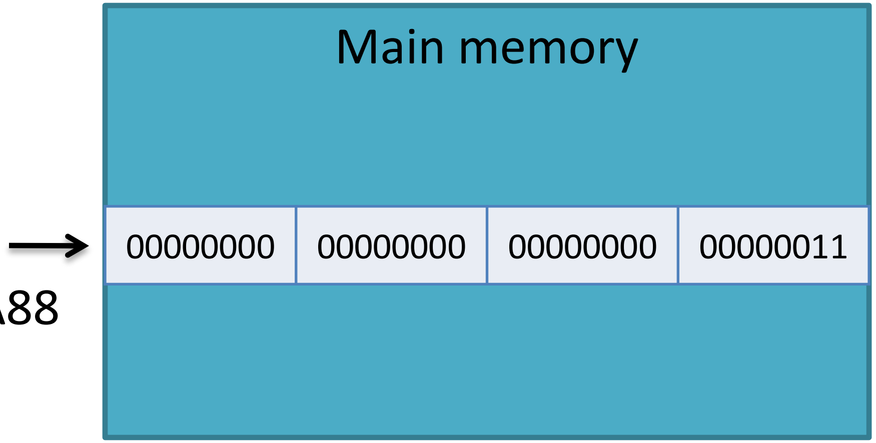
4 bytes (=32 bits)

Value chosen by the computer

Pointers

When we assign a value to a variable, the computer stores that value at the address in memory that was previously allocated for that variable.

```
int x;
x = 3;
```



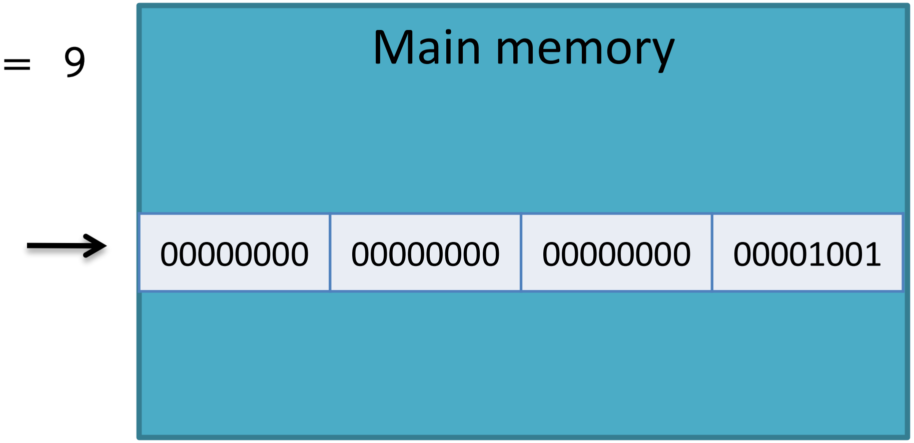
Main memory

Address
033727FA88

00000000	00000000	00000000	00000011

4 bytes (=32 bits)

```
x *= 3; // x = 9
```



Main memory

00000000	00000000	00000000	00001001

Pointers

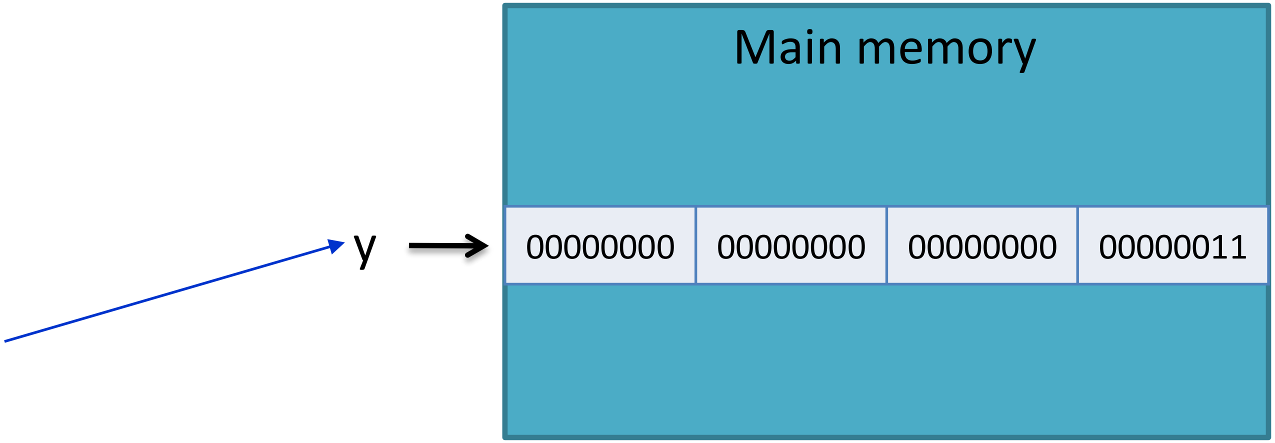
Pointers are variables for memory addresses.

They are declared using the * operator.

They are called pointers because they **point to the place in memory** where other variables are stored.

How can we know what the address in memory of a variable is?
The & operator.

```c
int x;
x = 3;

int *y;

y = &x;
```



y

Main memory

00000000	00000000	00000000	00000011

Pointers - Syntax

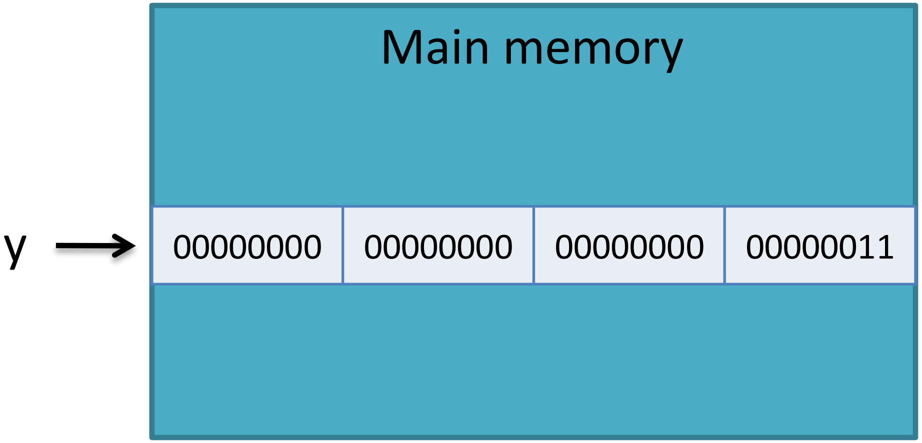
When we declare a pointer, we must specify the type of variable it will be pointing to

```
type *ptrName;
```

If we want to set a pointer to point to a variable, we must use the **&** operator

```
ptrName = &varName;
```

```
int x;
x = 3;

int *y;

y = &x;
```



Pointers : operators * and &

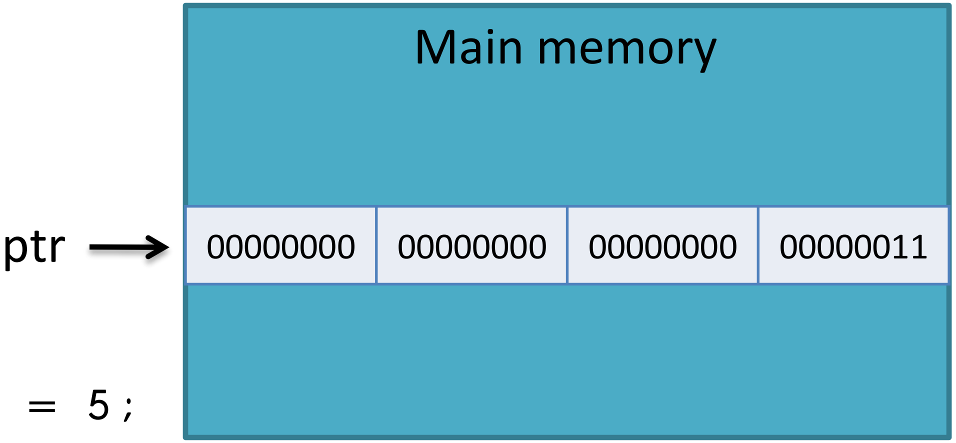
*** dereference operator** : gives the value in the memory pointed by a pointer (returns a value)

& reference operator: gives the address in memory of a variable (returns a pointer)

```
int x = 3;

int *ptr;

ptr = &x;


*ptr = 5; // x = 5;
```

Make `ptr` point to the address of `x`

Modify the value in address pointed by `ptr`



Main memory

ptr →

00000000	00000000	00000000	00000011

Pointers : operators * and &

***** **dereference operator** : gives the value in the memory pointed by a pointer (returns a value)

& **reference operator**: gives the address in memory of a variable (returns a pointer)

```
int x = 3;

int *ptr;
```

Make `ptr` point to the address of `x`

```
ptr = &x;
```

Modify the value in address pointed by `ptr`

```
*ptr = 5;    // x = 5;
```

Code	Meaning
`x`	Variable of type **int**
`ptr`	**Pointer** to an element of type int
`&x`	Pointer to x
`*ptr`	Variable of type int

Pointers : operators * and &

*** dereference operator** : gives the value in the memory pointed by a pointer (returns a value)

& reference operator: gives the address in memory of a variable (returns a pointer)

```
int x;

int *ptr;
```

V

```
&x

*ptr
```

✗

```
&ptr // pointer to a pointer

*x    // x is not a pointer
```

Pointers : operators * and &

*** dereference operator** : gives the value in the memory pointed by a pointer (returns a value)

& reference operator: gives the address in memory of a variable (returns a pointer)

```
int x;

int *ptr;
```

This is weird but actually ok, we will see its meaning later

V

```
&x

*ptr
```

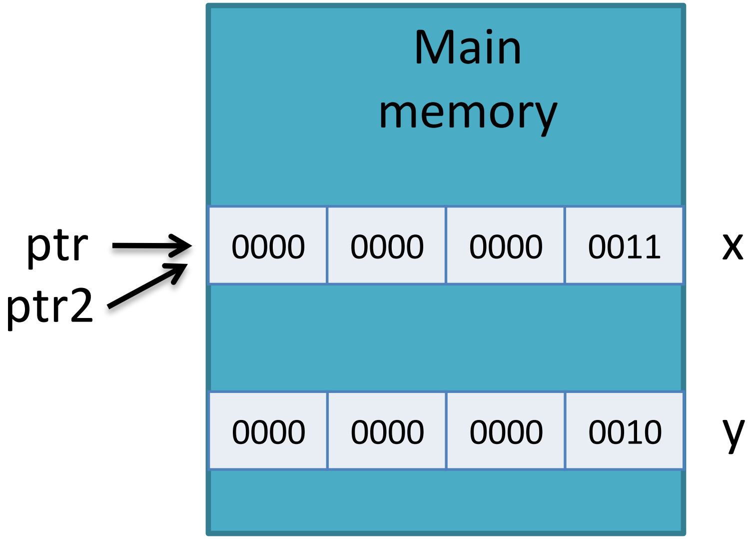
✗

```
&ptr // pointer to a pointer

*x    // x is not a pointer
```

Pointers

Multiple pointers can point to the same address

```c
int x = 3, y = 2;

int *ptr = &x;

int *ptr2 = ptr;
```



Main memory

ptr

ptr2

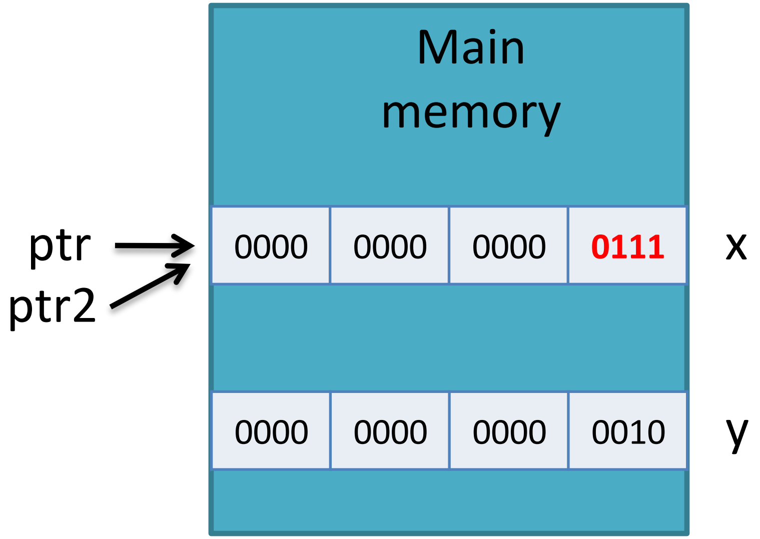
0000	0000	0000	0011	x
0000	0000	0000	0010	y

NOTE: first 4 bits omitted to save space

Pointers

Multiple pointers can point to the same address

```
int x = 3, y = 2;

int *ptr = &x;

int *ptr2 = ptr;

*ptr = 7;    // x = 7;
```



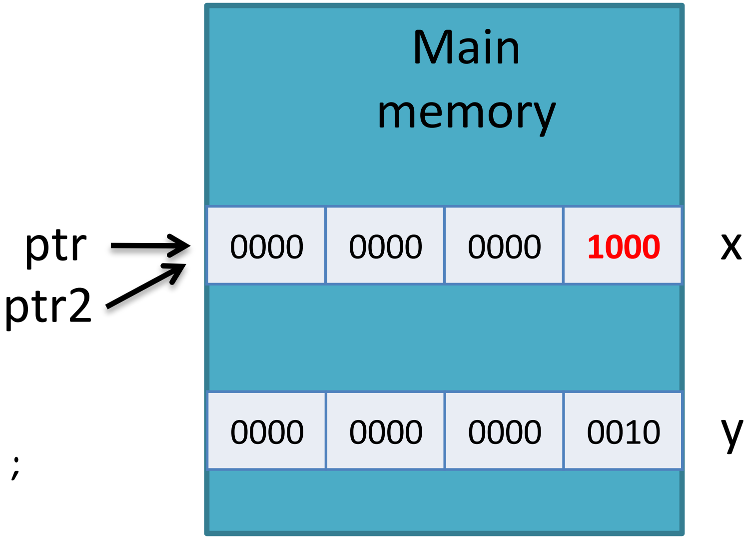
0000	0000	0000	0111	x
0000	0000	0000	0010	y

NOTE: first 4 bits omitted to save space

Pointers

Multiple pointers can point to the same address

```
int x = 3, y = 2;

int *ptr = &x;

int *ptr2 = ptr;

*ptr = 7;    // x = 7;
*ptr2 = *ptr2 + 1;  // x = 8;
```

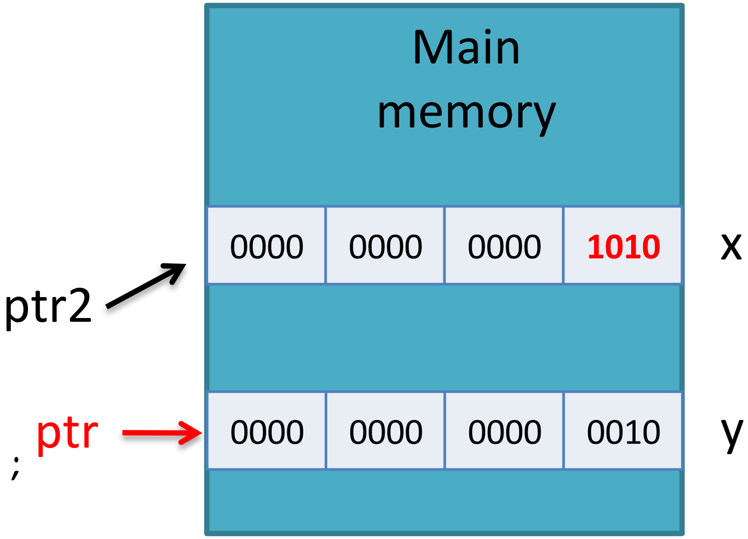


NOTE: first 4 bits omitted to save space

Pointers

Multiple pointers can point to the same address

```
int x = 3, y;

int *ptr = &x;

int *ptr2 = ptr;

*ptr = 7;    // x = 7;
*ptr2 = *ptr2 + 1;  // x = 8;

ptr = &y;

*ptr2 = 10; // x = 10;
```

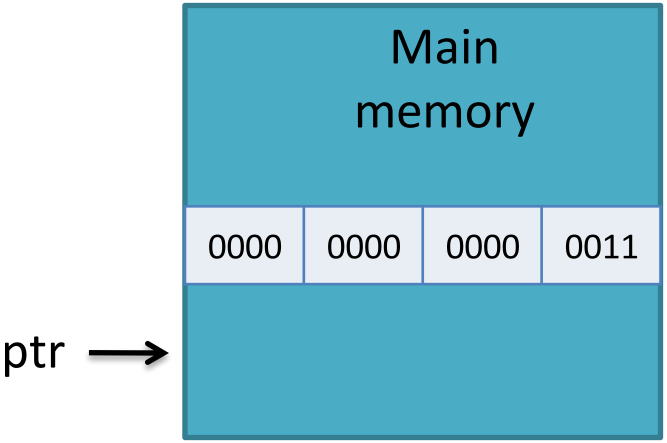


Ptr2 is still pointing to x, even if ptr changed

Pointers

Be careful when using incremental operators!

```
int x = 3;

int *ptr = &x;

*ptr++;    // x = ?
```

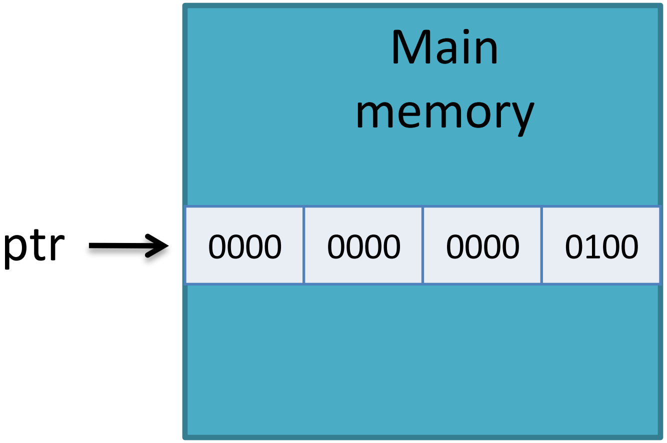


In this case I am incrementing `ptr`, NOT the value of the variable pointed by it!

Pointers

Be careful when using incremental operators!

```
int x = 3;

int *ptr = &x;

(*ptr)++;    // x = 4;
```



Pointers and Arrays

- When set a pointer to an array, the pointer points to the **first element** in the array

```
float arr[3] = {1, 2, 5};
float *pa;

pa = arr;
pa = &arr[0];
```

These two notations are equivalent

- C automatically keeps pointer arithmetic in terms of the size of the variable type being pointed to

```
arr[0]  ⟷  *pa
arr[1]  ⟷  *(pa+1)
arr[2]  ⟷  pa[2]
```

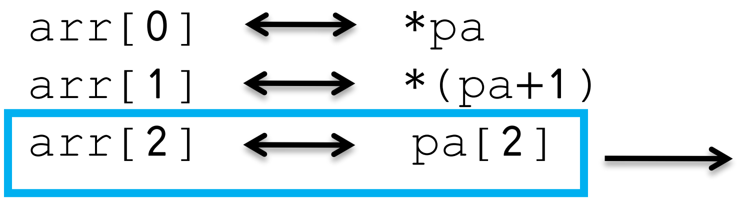
Pointers and Arrays

- When set a pointer to an array, the pointer points to the **first element** in the array

```
float arr[3] = {1, 2, 5};
float *pa;

pa = arr;
pa = &arr[0];
```

These two notations are equivalent

- C automatically keeps pointer arithmetic in terms of the size of the variable type being pointed to



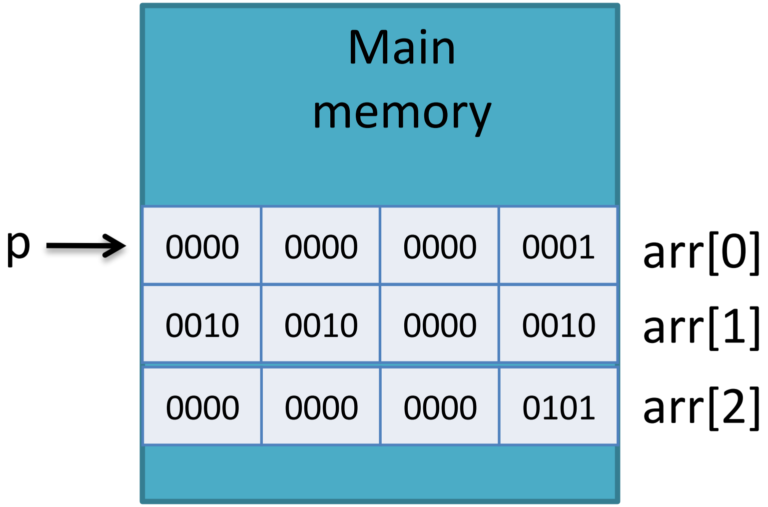
```
arr[0]  ⟷  *pa
arr[1]  ⟷  *(pa+1)
arr[2]  ⟷  pa[2]
```

Once we have set a pointer to the beginning of one array, we can use it as if it were the array itself!

Pointers and Arrays

When set a pointer to an array, the pointer points to the **first element** in the array

```
float arr[3] = {1, 2, 5};

float *p = arr;

*p = 5; // arr[0] = 5;
```



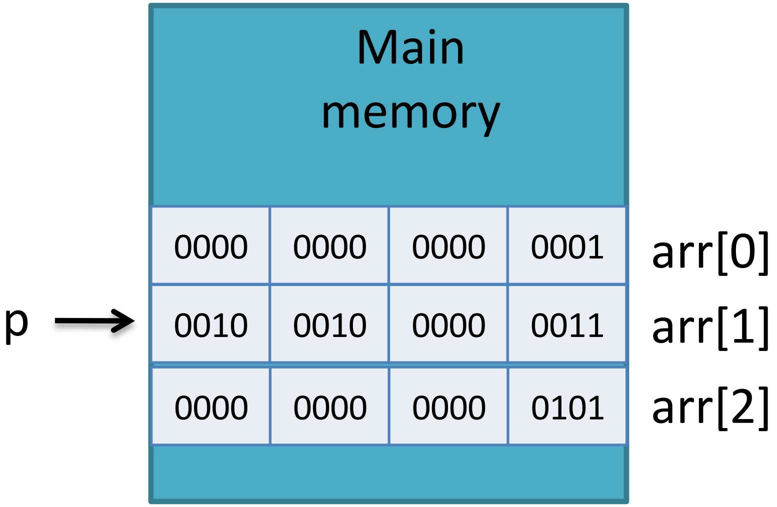
Main memory

p →	0000	0000	0000	0001	arr[0]
	0010	0010	0000	0010	arr[1]
	0000	0000	0000	0101	arr[2]

Pointers and Arrays

When set a pointer to an array, the pointer points to the **first element** in the array

```
float arr[3] = {1, 2, 5};

float *p = arr;

*p = 5; // arr[0] = 5;

p++;

*p = 3; // arr[1] = 3;
```



Main memory

	0000	0000	0000	0001	arr[0]
p →	0010	0010	0000	0011	arr[1]
	0000	0000	0000	0101	arr[2]

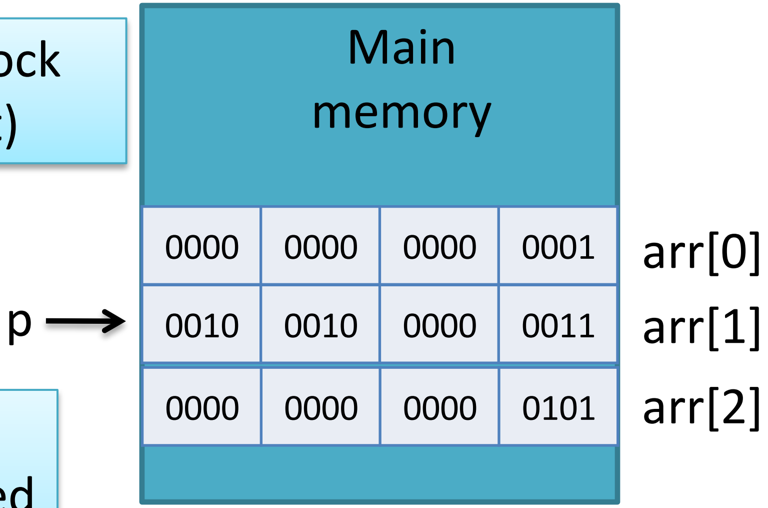
Pointers and Arrays

When set a pointer to an array, the pointer points to the first element in the array

```
float arr[3] = {1, 2, 5};

float *p = arr;

*p = 5; // arr[0] = 5;

p++;

*p = 3; // arr[1] = 3;
```

Note that for arrays, we do not need the reference & operator

p jumps in memory a block of 4 bytes (size of a float)



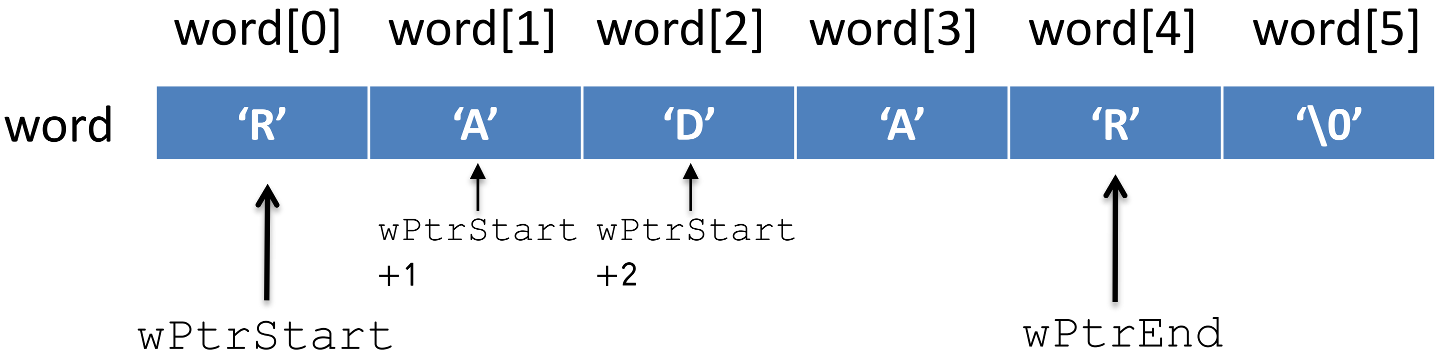
Main memory

	0000	0000	0000	0001	arr[0]
p →	0010	0010	0000	0011	arr[1]
	0000	0000	0000	0101	arr[2]

Remember: an array is a set of elements of the same type allocated **contiguously** in memory!

Pointers and Arrays

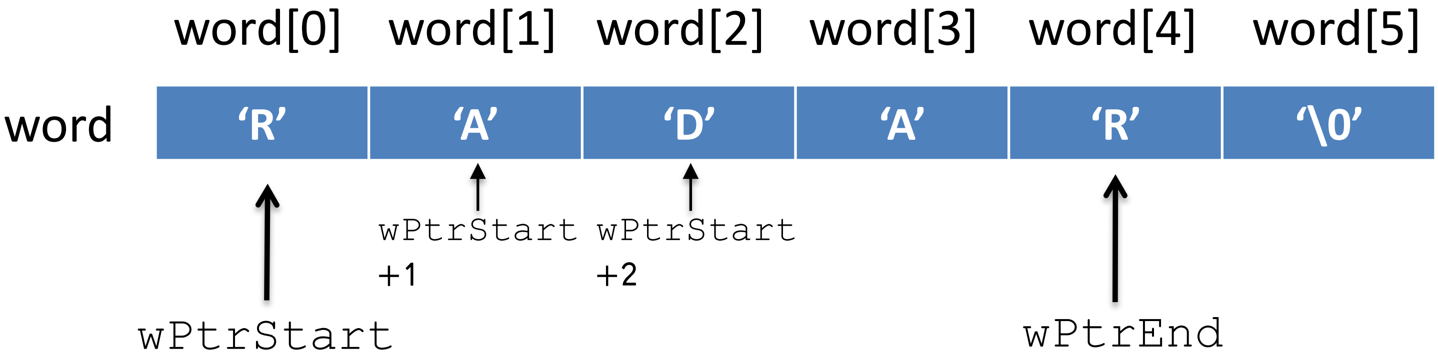
palindrome.c



```
char *wPtrStart = word;
char *wPtrEnd = wPtrStart + strlen(word)-1;

for( i=0 ; (i < strlen(word)/2) && (flag == 1) ; i++ ){

    if( *wPtrStart != *wPtrEnd ){
       flag = 0;
    }

    wPtrStart++;
    wPtrEnd--;
}
```

Pointers and Arrays

palindrome.c



	word[0]	word[1]	word[2]	word[3]	word[4]	word[5]
word	'R'	'A'	'D'	'A'	'R'	'\0'

wPtrStart

wPtrStart +1

wPtrStart +2

wPtrEnd

```
char *wPtrStart = word;
char *wPtrEnd = wPtrStart + strlen(word)-1;

for( i=0 ; (i < strlen(word)/2) && (flag == 1) ; i++ ){

    if( *wPtrStart != *wPtrEnd ){
       flag = 0;
    }

    wPtrStart++;
    wPtrEnd--;
}
```

When we increment or decrement, the pointers move by 1 byte (pointers to char)

Pointers : operators * and &

Now we know exactly what happens in sscanf !

```
sscanf( string, "format", &var1, …, &varN);
```

Pointers to the addresses in memory where var1,..,varN are stored !

Functions
Passing arguments by value/reference

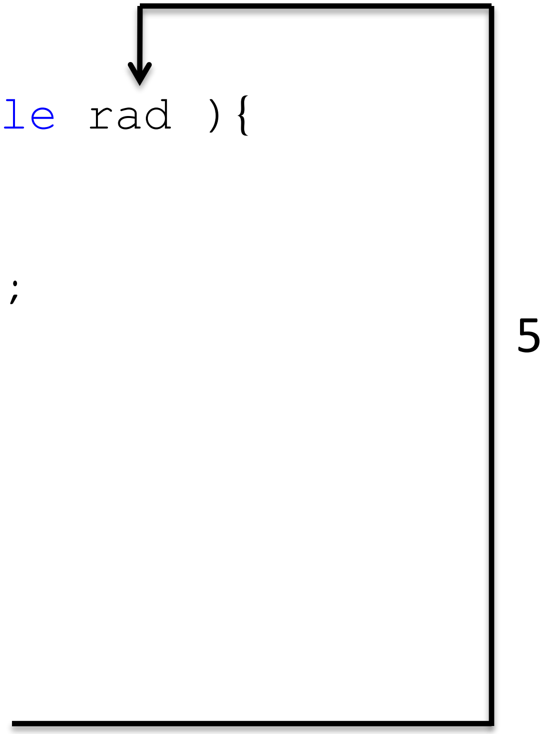
- Pass by value (what we have seen so far): the value of the variable used at invocation time is copied into a local variable inside the function

- Pass by reference : a pointer to the variable used at invocation time is passed to the function. We can modify the variable's value inside the function

Functions
Passing arguments by value/reference

- Pass by value (what we have seen so far): the value of the variable used at invocation time is copied into a local variable inside the function

```
double computeCirc( double rad ){
  rad = 2;
  return(2 * rad * 3.14);
}

int main(){
  double r = 5, circ;
  circ = computeCirc(r);
  return 0;
}
```



Functions
Passing arguments by value/reference

- Pass by value (what we have seen so far): the value of the variable used at invocation time is copied into a local variable inside the function

```
double computeCirc( double rad ){
  rad = 2;
  return(2 * rad * 3.14);
}

int main(){
  double r = 5, circ;
  circ = computeCirc(r);
  return 0;
}
```

`r` is not affected by anything we do inside the function

Functions
Passing arguments by value/reference

- Pass by reference : a pointer to the variable used at invocation time is passed to the function. We can modify the variable’s value inside the function

```
double computeCirc( double *rad ){
  *rad = 2;
  return(2 * (*rad) * 3.14);
}

int main(){
  double r = 5, circ;
  circ = computeCirc(&r);
  return 0;
}
```

Address of `r`

Functions

circPointer.c

Passing arguments by value/reference

- Pass by reference : a pointer to the variable used at invocation time is passed to the function. We can modify the variable’s value inside the function

```
double computeCirc( double *rad ){
  *rad = 2;
  return(2 * (*rad) * 3.14);
}

int main(){
  double r = 5, circ;
  circ = computeCirc(&r);
  return 0;
}
```

`r` has been modified!

COMsW 1003-1

Introduction to Computer Programming in C

Lecture 12

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Announcements

Homework 3 is out

- Due on Monday, 03/21/11 at the beginning of class, no exceptions

Midterm

- In class on Wednesday, 03/09/11
- Will cover everything up to Lecture 13 (included)
- Open books, open notes
- Closed electronic devices

Today

- Passing arguments to function by value vs. by reference (from Lec 11)
- Functions returning pointers
- Pointers of pointers

Functions Returning Pointers

- Naturally, a function can return a pointer
- This is a way to return an array, but must be **careful** about what has been **allocated** in memory

```
returnType * functionName( parmeters )
```

NOTE

`NULL` is the equivalent of zero for pointers splitString.c

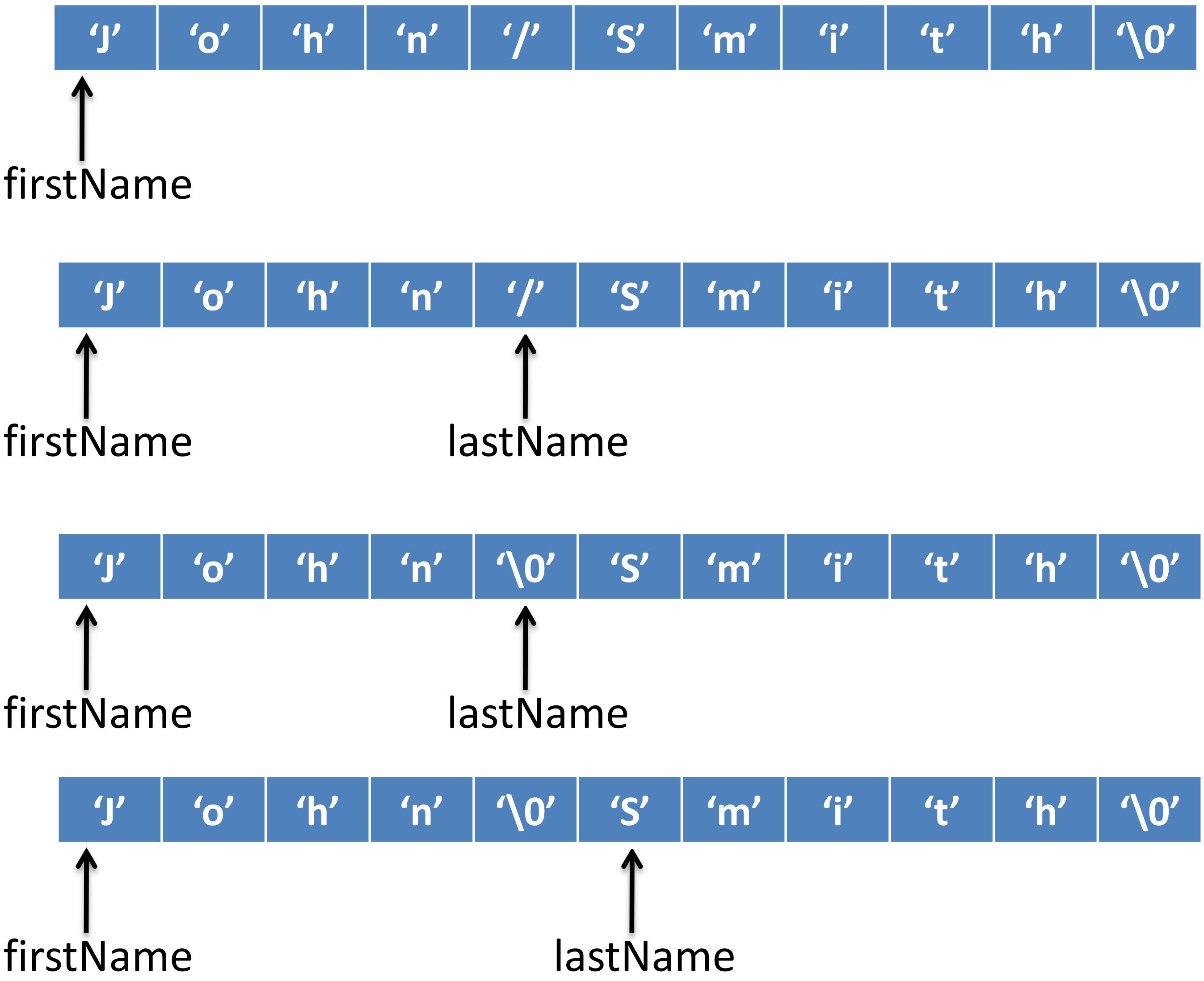
Functions Returning Pointers

Example: using pointers to return a string

Given a string of the type “firstNAme/lastName”
We want to split it into two separate entities to print

Functions Returning Pointers

splitString.c



POINT 1

‘J’	‘o’	‘h’	‘n’	‘/’	‘S’	‘m’	‘i’	‘t’	‘h’	‘\0’

firstName

POINT 2

‘J’	‘o’	‘h’	‘n’	‘/’	‘S’	‘m’	‘i’	‘t’	‘h’	‘\0’

firstName lastName

POINT 3

‘J’	‘o’	‘h’	‘n’	‘\0’	‘S’	‘m’	‘i’	‘t’	‘h’	‘\0’

firstName lastName

POINT 4

‘J’	‘o’	‘h’	‘n’	‘\0’	‘S’	‘m’	‘i’	‘t’	‘h’	‘\0’

firstName lastName

Const pointers

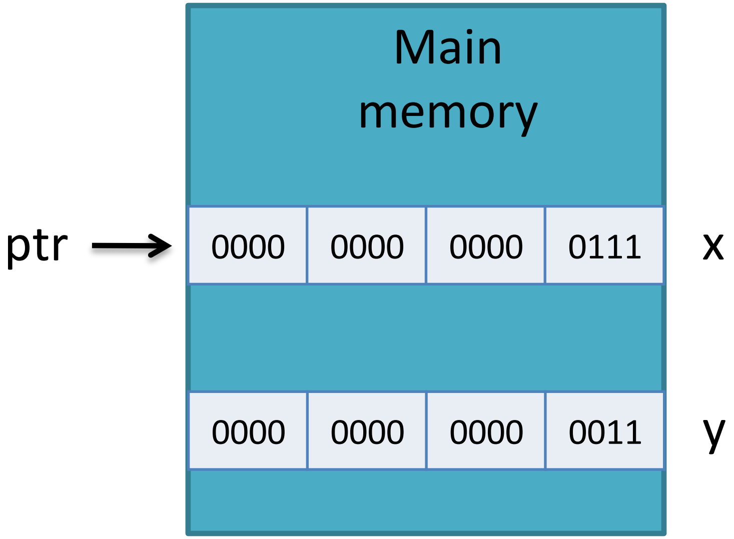
point.c

`const type *`

When we try to declare a pointer to be a constant like this, it means that the value at the address in memory it points cannot be modified

This does NOT mean that the pointer is constant, it can be changed!

```
int x = 7, y = 3;

const int *ptr = &x;

*ptr = 11;
```



Main memory

ptr →	0000	0000	0000	0111	x
	0000	0000	0000	0011	y

Const pointers

point.c

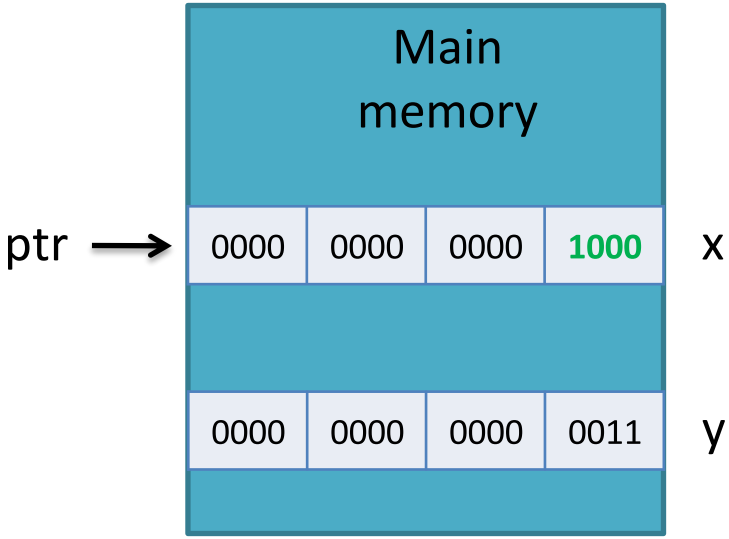
```
const type *
```

When we try to declare a pointer to be a constant like this, it means that the value at the address in memory it points cannot be modified

This does NOT mean that the pointer is constant, it can be changed!

```
int x = 7, y = 3;

const int *ptr = &x;

*ptr = 11;
```
✗

```
x = 8;
```
V



Const pointers

point.c

`const type *`

When we try to declare a pointer to be a constant like this, it means that the value at the address in memory it points cannot be modified

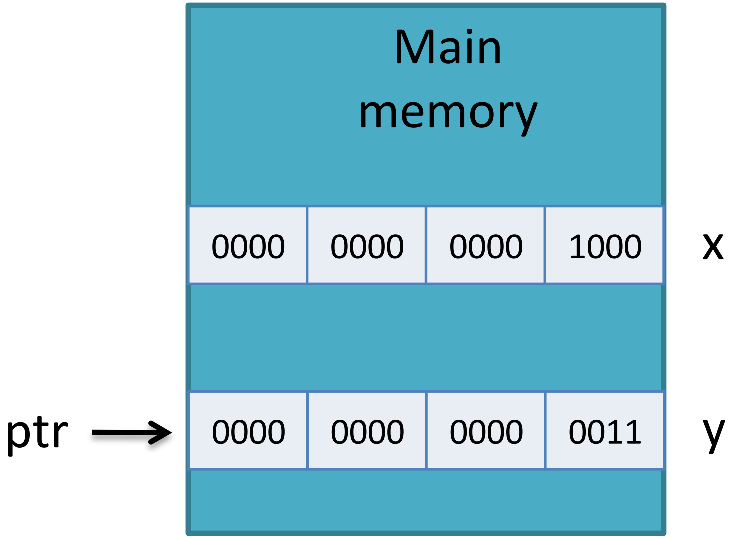
This does NOT mean that the pointer is constant, it can be changed!

```
int x = 7, y = 3;

const int *ptr = &x;
```

`*ptr = 11;` ✗

`x = 8;` V

`ptr = &y;` V



Main memory

0000	0000	0000	1000	x
0000	0000	0000	0011	y

Const pointers

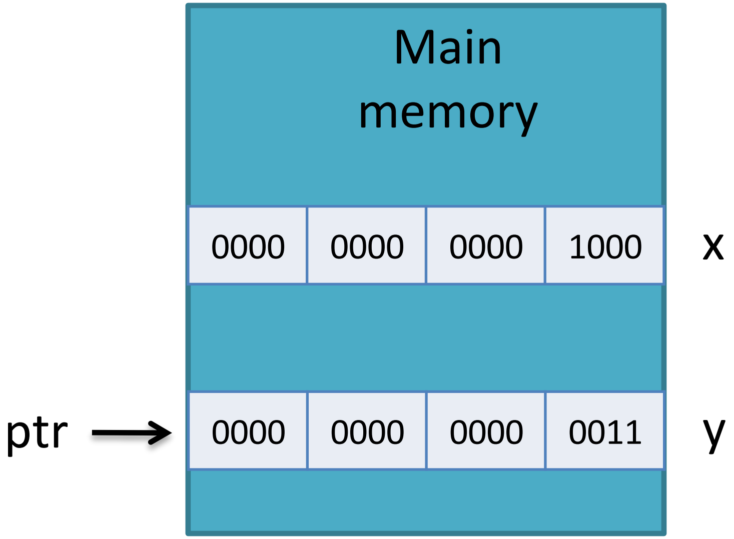
point.c

`const type *`

When we try to declare a pointer to be a constant like this, it means that the value at the address in memory it points cannot be modified

This does NOT mean that the pointer is constant, it can be changed!

```
int x = 7, y = 3;

const int *ptr = &x;

*ptr = 11;  ✗

x = 8;  V

ptr = &y;  V

*ptr = 9; ✗

printf("x = %d, y = %d\n",x,*ptr);
```



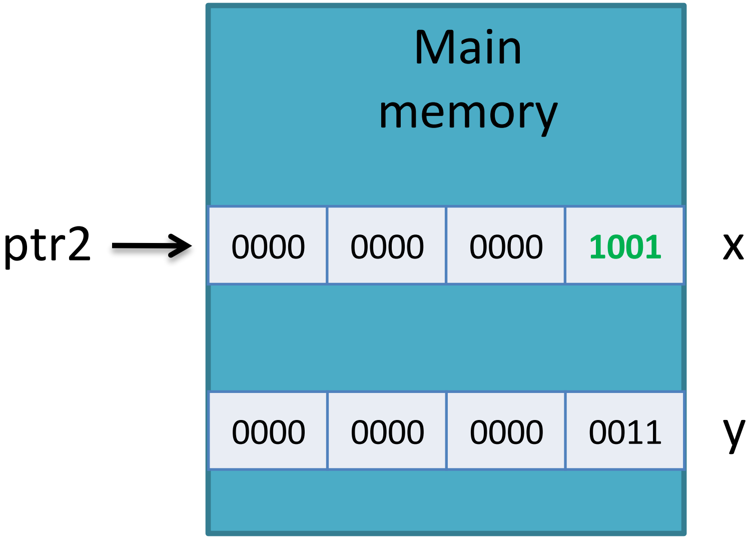
Const pointers

point.c

`type * const`

This is the declaration of a constant pointer. In this case, the pointer is fixed, but the value at the address it points to can be modified

```
int x = 7, y = 3;

int * const ptr2 = &x;

*ptr2 = 9; V

ptr2++;  ✗

ptr2 = &y; ✗
```



0000	0000	0000	1001
0000	0000	0000	0011

```
printf("x = %d, x = %d\n", x, *ptr2);
```

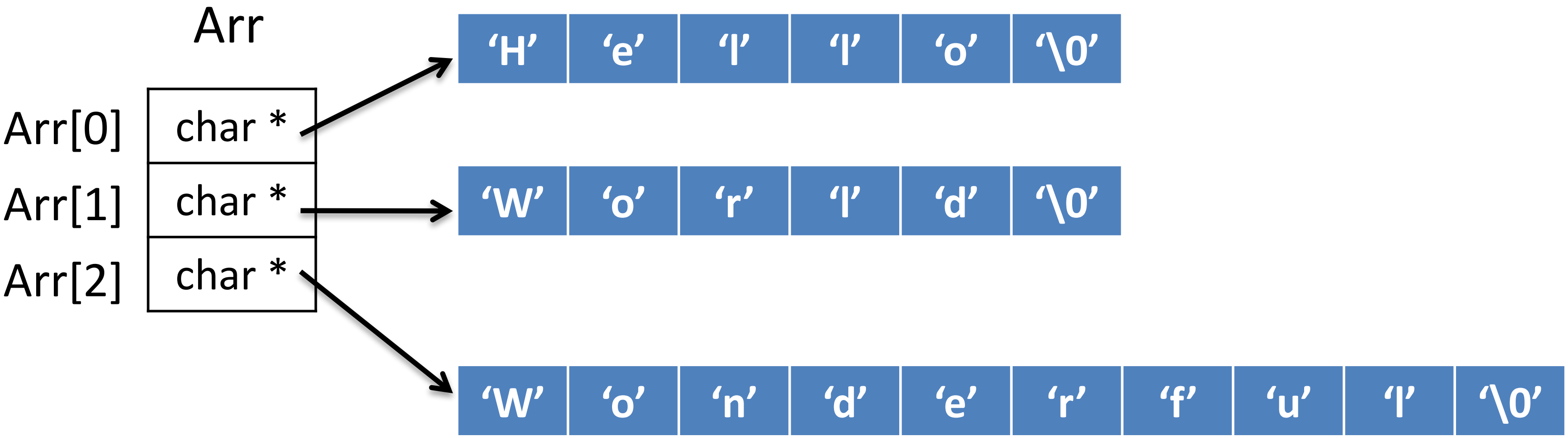
stringArrays.c

Arrays of strings

- An array Arr of 3 strings of variable length

```
char *Arr[3]={ “Hello”, ”World”, ”Wonderful” };

Arr[2] ⟷ Arr+2 // ”Wondeful”
```

- Arr is an array of **3** elements. Each element in Arr is of type **pointer to char**.



‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’				
‘W’	‘o’	‘r’	‘l’	‘d’	‘\0’				
‘W’	‘o’	‘n’	‘d’	‘e’	‘r’	‘f’	‘u’	‘l’	‘\0’

stringArrays.c

Arrays of strings

- An array Arr of 3 strings of variable length

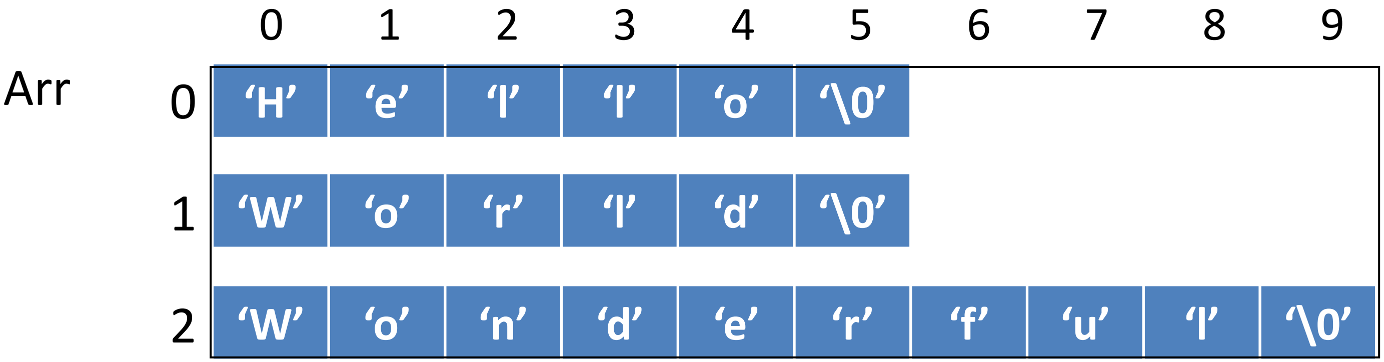
```
char *Arr[3]={ “Hello”, ”World”, ”Wonderful” };

Arr[2] ⟷ Arr+2 // ”Wondeful”
```

- An array Arr of 3 strings of maximum length = 15

```
char Arr2[3][15] = { “Hello2”, ”World2”, ”Wonderful2” };

Arr2[0]⟷ Arr2 // ”Hello2”

Arr2[1] ⟷ Arr2+1 // ”World2”
```

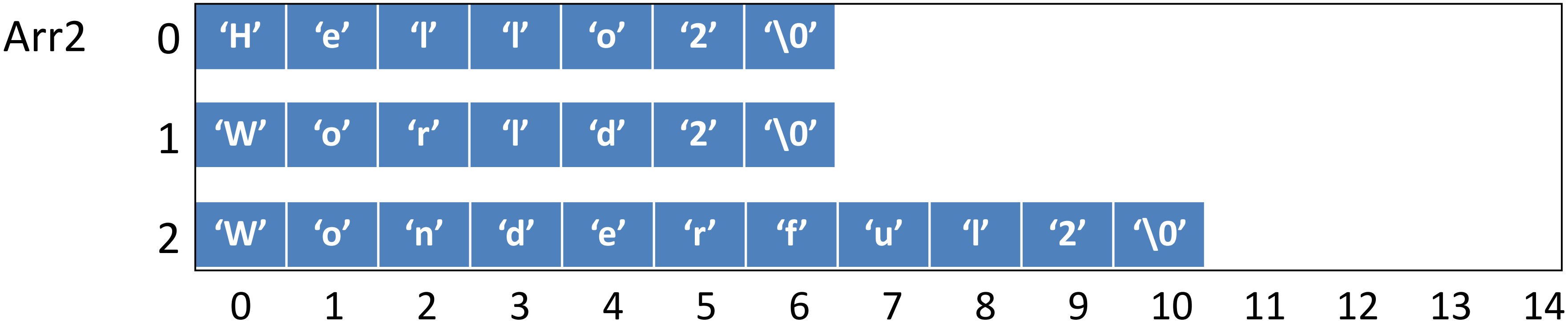
stringArrays.c

Pointers of pointers



Arr

	0	1	2	3	4	5	6	7	8	9
0	‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’				
1	‘W’	‘o’	‘r’	‘l’	‘d’	‘\0’				
2	‘W’	‘o’	‘n’	‘d’	‘e’	‘r’	‘f’	‘u’	‘l’	‘\0’



Arr2

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	‘H’	‘e’	‘l’	‘l’	‘o’	‘2’	‘\0’								
1	‘W’	‘o’	‘r’	‘l’	‘d’	‘2’	‘\0’								
2	‘W’	‘o’	‘n’	‘d’	‘e’	‘r’	‘f’	‘u’	‘l’	‘2’	‘\0’				

stringArrays.c

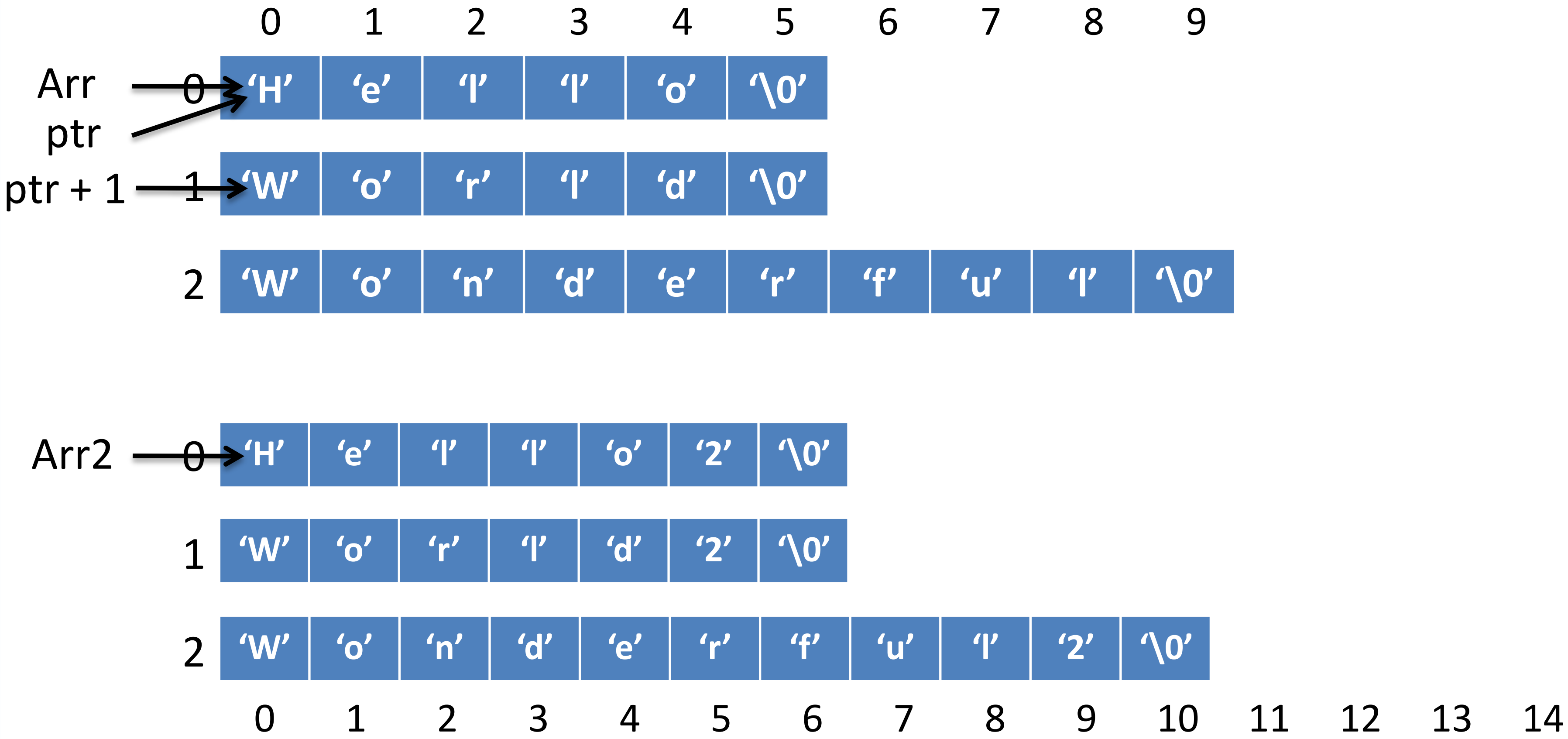
Pointers of pointers

- A pointer can point to another pointer
- In a sense, it’s the equivalent of matrices!

```
int x = 3;

int *p = &x;

int **p2 = &p;

x = 2;   ⟷   *p = 2;   ⟷   **p2 = 2;


char *Arr[3]={ “Hello”, ”World”, ”Wonderful” };

char **ptr;

ptr = Arr;
```

stringArrays.c

Pointers of pointers



	0	1	2	3	4	5	6	7	8	9
0	‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’				
1	‘W’	‘o’	‘r’	‘l’	‘d’	‘\0’				
2	‘W’	‘o’	‘n’	‘d’	‘e’	‘r’	‘f’	‘u’	‘l’	‘\0’

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	‘H’	‘e’	‘l’	‘l’	‘o’	‘2’	‘\0’								
1	‘W’	‘o’	‘r’	‘l’	‘d’	‘2’	‘\0’								
2	‘W’	‘o’	‘n’	‘d’	‘e’	‘r’	‘f’	‘u’	‘l’	‘2’	‘\0’				

Pointers of pointers

stringArrays.c

```
char *Arr[3]={ "Hello", "World", "Wonderful" };
char **ptr;
ptr = Arr;
```

`*(*(ptr+1)+2)` ?

Pointers of pointers

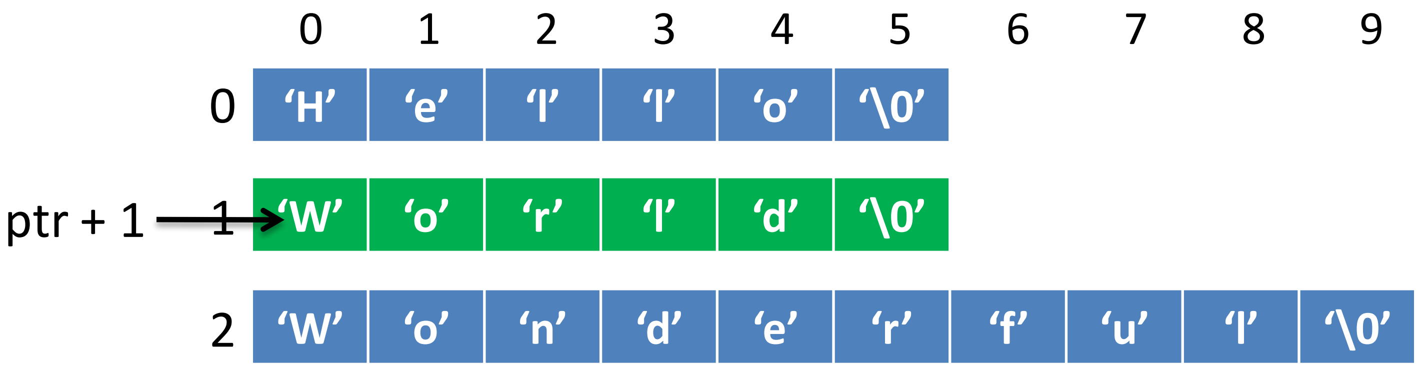
stringArrays.c

```
char *Arr[3]={ “Hello”, ”World”, ”Wonderful” };
char **ptr;
ptr = Arr;
```

((ptr+1)+2)

1 . ptr+1

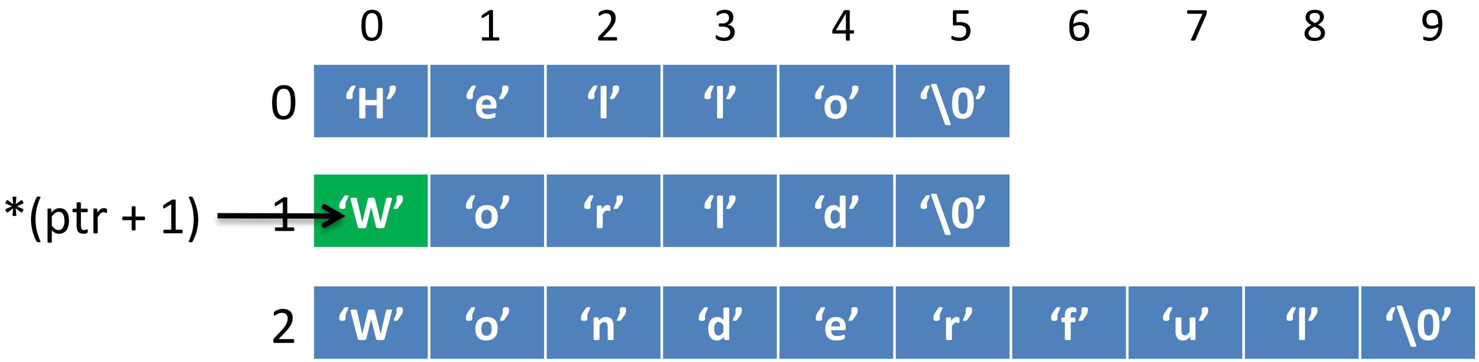
ptr+1 points to the whole line



	0	1	2	3	4	5	6	7	8	9
0	‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’				
1	🟩 ‘W’	🟩 ‘o’	🟩 ‘r’	🟩 ‘l’	🟩 ‘d’	🟩 ‘\0’				
2	‘W’	‘o’	‘n’	‘d’	‘e’	‘r’	‘f’	‘u’	‘l’	‘\0’

2 . *(ptr+1)

*(ptr+1) points to the first element of the line



	0	1	2	3	4	5	6	7	8	9
0	‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’				
1	🟩 ‘W’	‘o’	‘r’	‘l’	‘d’	‘\0’				
2	‘W’	‘o’	‘n’	‘d’	‘e’	‘r’	‘f’	‘u’	‘l’	‘\0’

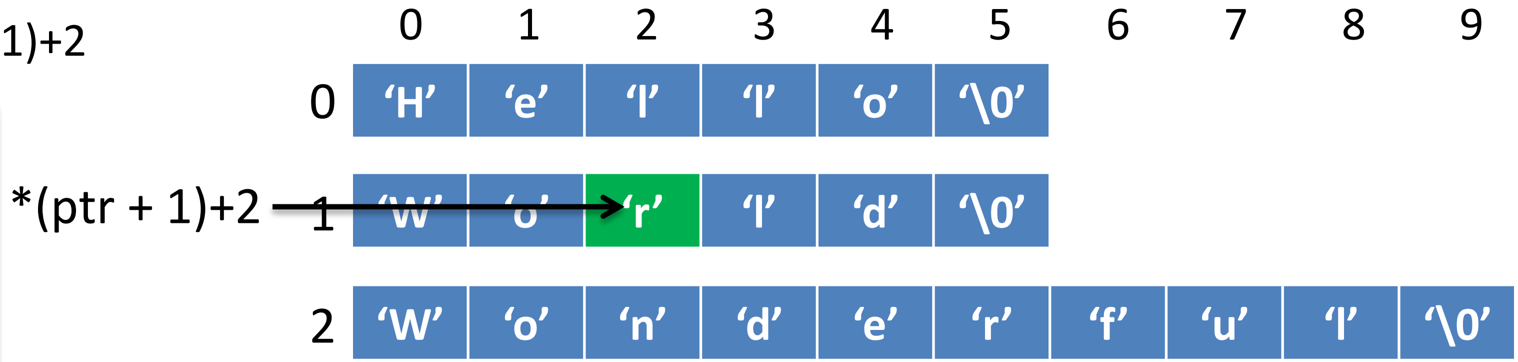
Pointers of pointers

stringArrays.c

```
char *Arr[3]={ "Hello", "World", "Wonderful" };
char **ptr;
ptr = Arr;
```

((ptr+1)+2)

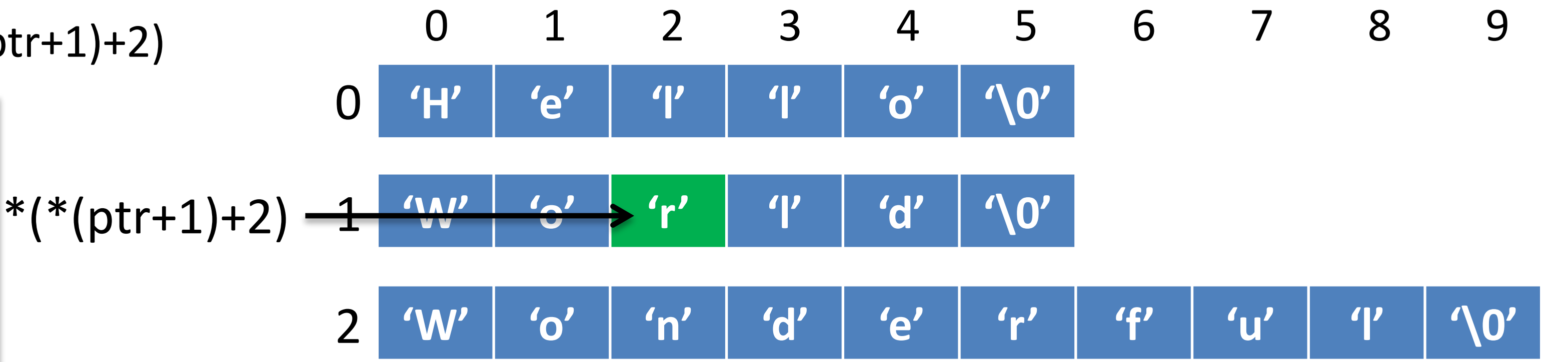
3 . *(ptr+1)+2



	0	1	2	3	4	5	6	7	8	9
0	‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’				
1	‘W’	‘o’	‘r’	‘l’	‘d’	‘\0’				
2	‘W’	‘o’	‘n’	‘d’	‘e’	‘r’	‘f’	‘u’	‘l’	‘\0’

*(ptr+1)+2 points to the third element of the line

2 . *(*(ptr+1)+2)



	0	1	2	3	4	5	6	7	8	9
0	‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’				
1	‘W’	‘o’	‘r’	‘l’	‘d’	‘\0’				
2	‘W’	‘o’	‘n’	‘d’	‘e’	‘r’	‘f’	‘u’	‘l’	‘\0’

Now we get the value stored at the address we pint

Pointers of pointers

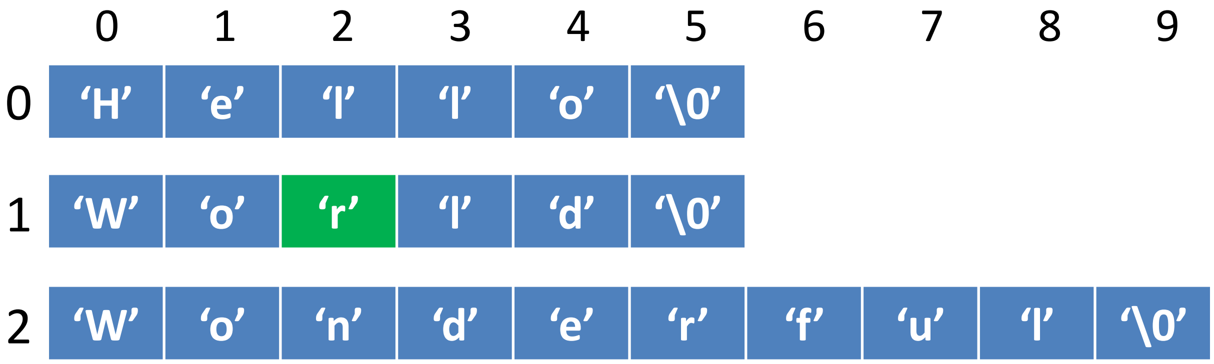
stringArrays.c

```
char *Arr[3]={ “Hello”, ”World”, ”Wonderful” };
char **ptr;
ptr = Arr;
```

((ptr+1)+2)

Avoid this notation!

ptr[1][2] is much better!



	0	1	2	3	4	5	6	7	8	9
0	‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’				
1	‘W’	‘o’	‘r’	‘l’	‘d’	‘\0’				
2	‘W’	‘o’	‘n’	‘d’	‘e’	‘r’	‘f’	‘u’	‘l’	‘\0’

Pointers vs. Arrays

	Arrays		Pointers
1D array of 5 int	`int x[5];`	⟷	`int *xPtr;`
2D array of 6 int 2x3 matrix	`int y[2][3];`	⟷	`int **yPtr;`
2D array of 4 int 2x2 matrix	`int* z[2]={{1,2},{2,1}};`	⟷	`int **zPtr;`
1D array of 5 char string	`char c[] = “mike”;`	⟷	`char *cPtr;`

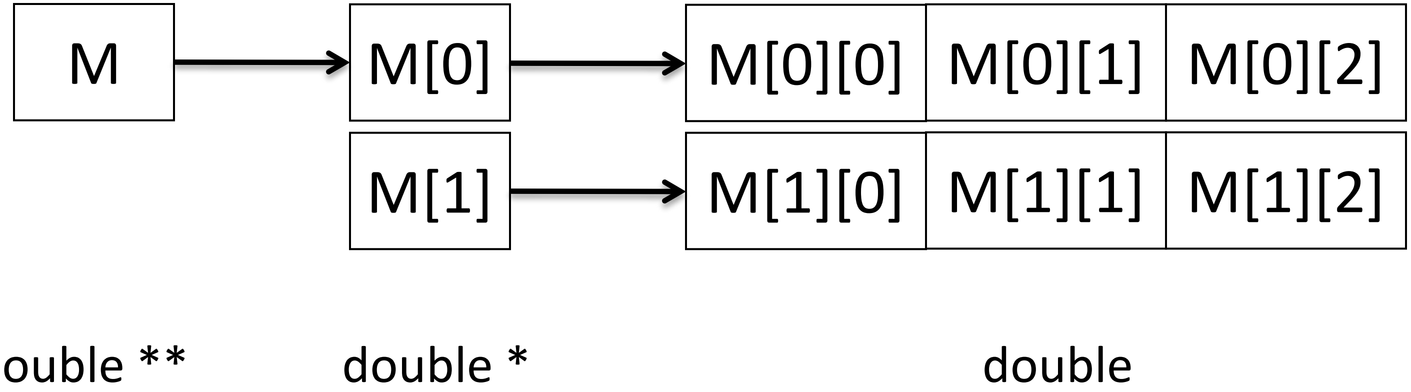
Space has been allocated in memory for the arrays

Space has been allocated in memory only for the pointers variables, **NOT** for the arrays they will point to.
The DIMENSIONS of the arrays are UNKNOWN

Multidimensional Arrays

2x3 matrix of double

```
double M0[2][3];
double *M1[2] = M0;
double **M = M0;
```



M

M[0]

M[1]

M[0][0] M[0][1] M[0][2]

M[1][0] M[1][1] M[1][2]

double **

double *

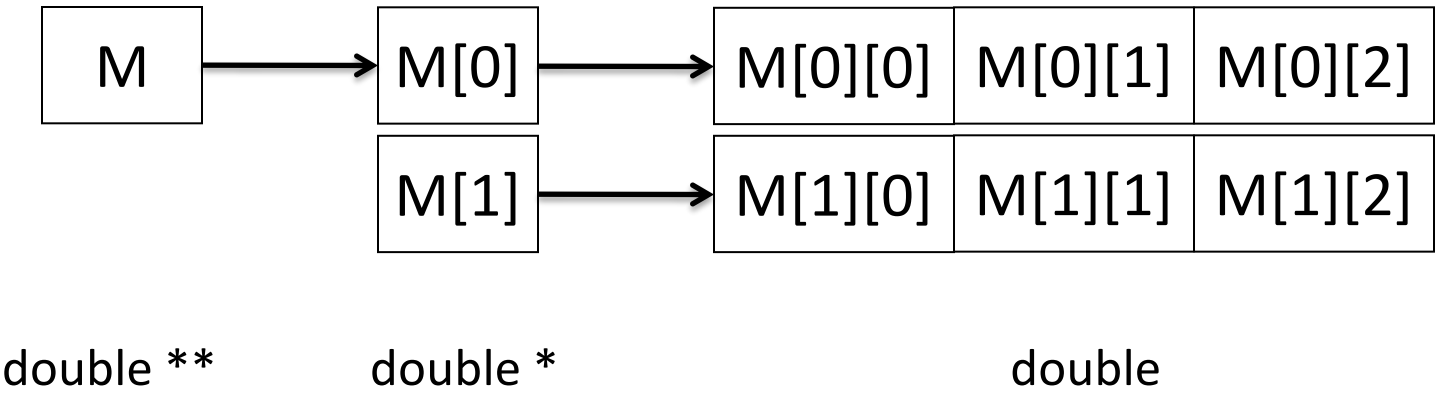
double

Multidimensional Arrays

2x3 matrix of double

```
double M0[2][3];
double *M1[2] = M0;
double **M = M0;
```

The difference between M0, M1 and M is that
M1 and M can have ANY SIZE !



M

M[0]

M[1]

M[0][0] M[0][1] M[0][2]

M[1][0] M[1][1] M[1][2]

double ** double * double

COMsW 1003-1

Introduction to Computer Programming in C

Lecture 13

Spring 2011

Instructor: Michele Merler

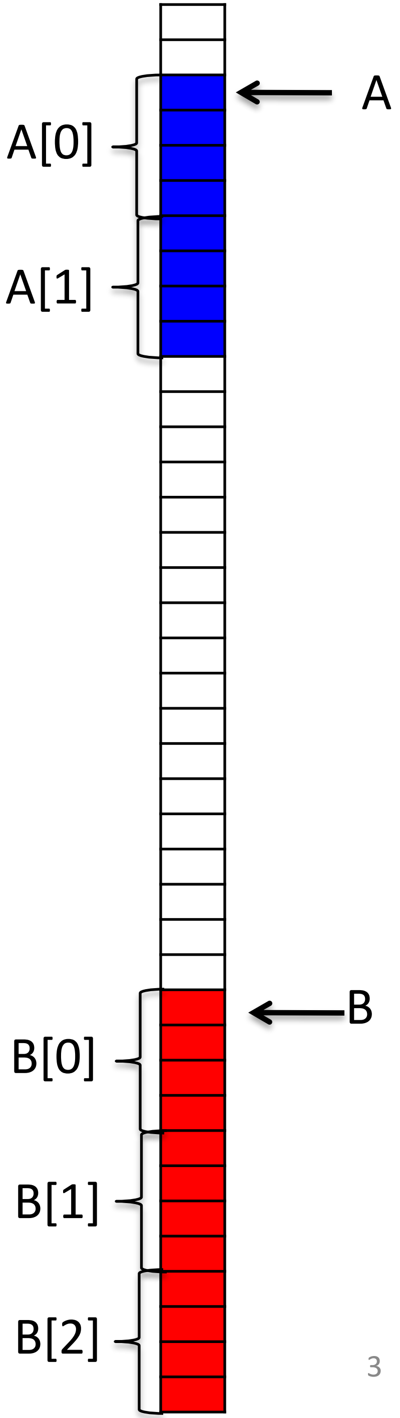
http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Today

- Finish pointers (from Lecture 12)
- FILE I/O

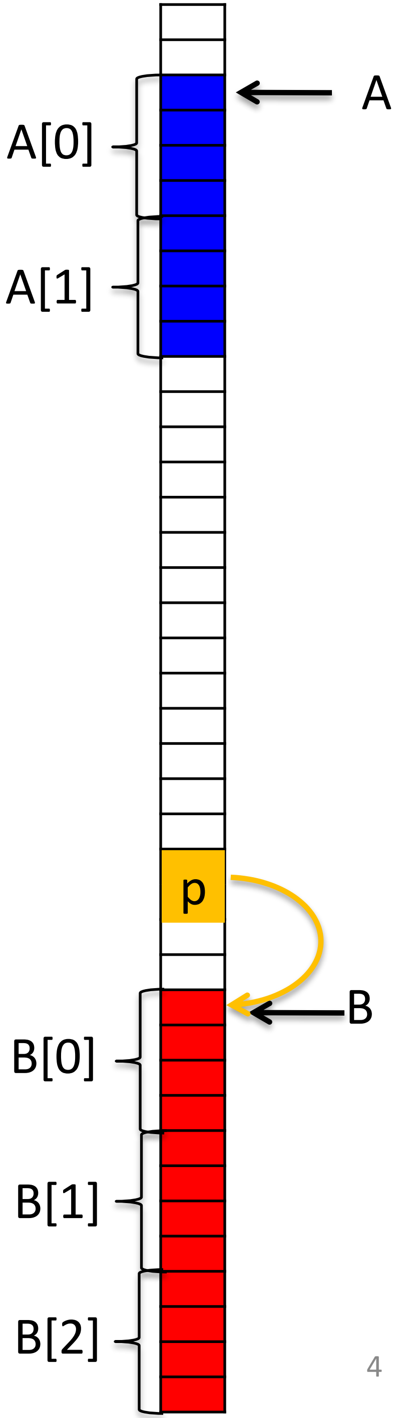
Pointers of pointers

```
float A[2] = { 1, 2 };
float B[3] = { 7, 1, 5};
```



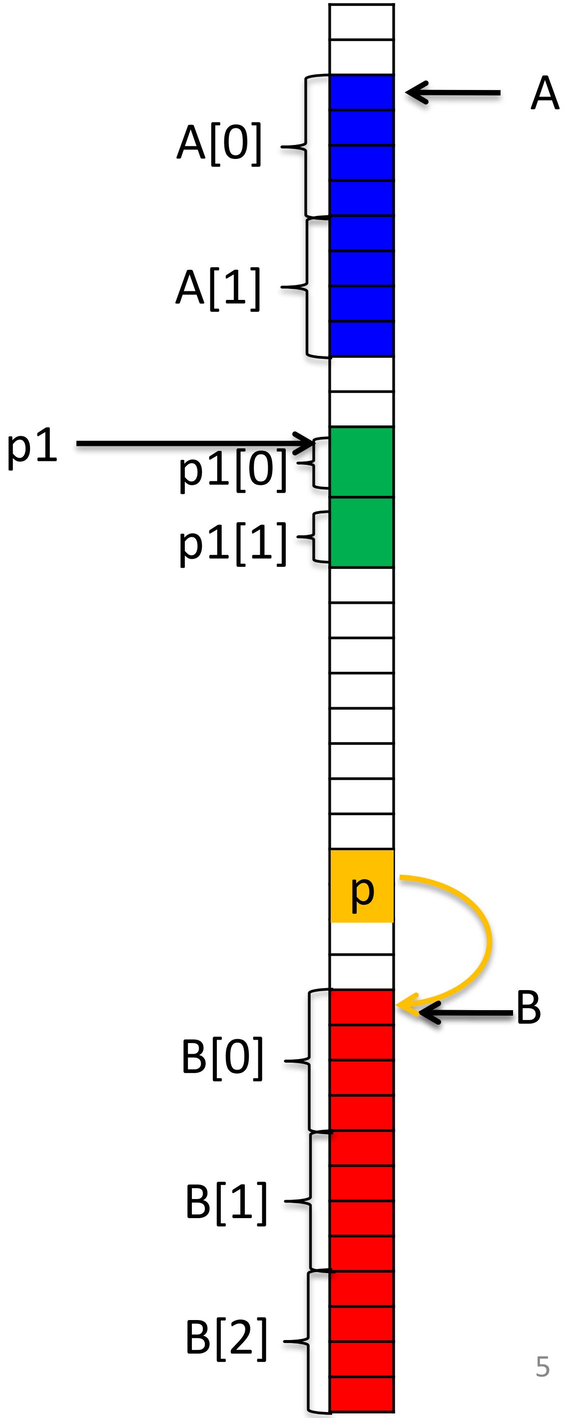
Pointers of pointers

```
float A[2] = { 1, 2 };
float B[3] = { 7, 1, 5};

float *p = B;
```



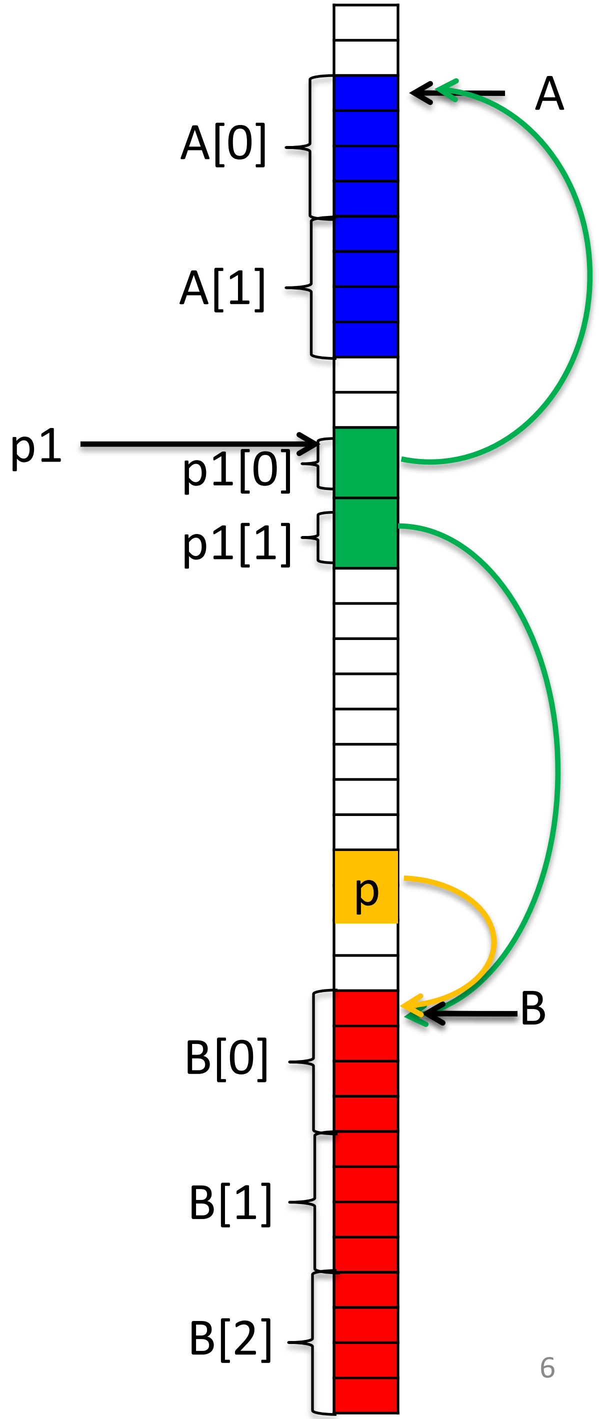
Pointers of pointers

```
float A[2] = { 1, 2 };
float B[3] = { 7, 1, 5};

float *p = B;

float *p1[2];
```



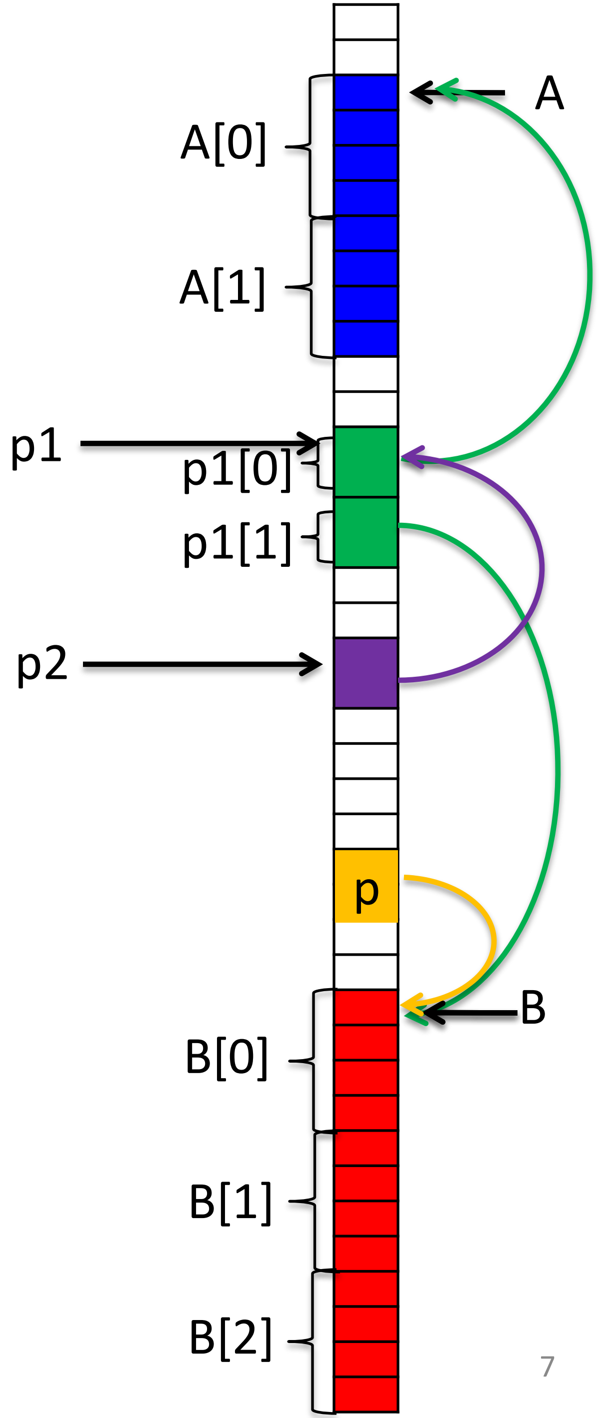
Pointers of pointers

```
float A[2] = { 1, 2 };
float B[3] = { 7, 1, 5};

float *p = B;

float *p1[2];
p1[0] = A;  // p1[0] is a pointer to float
p1[1] = B;  // p1[1] is a pointer to float
```



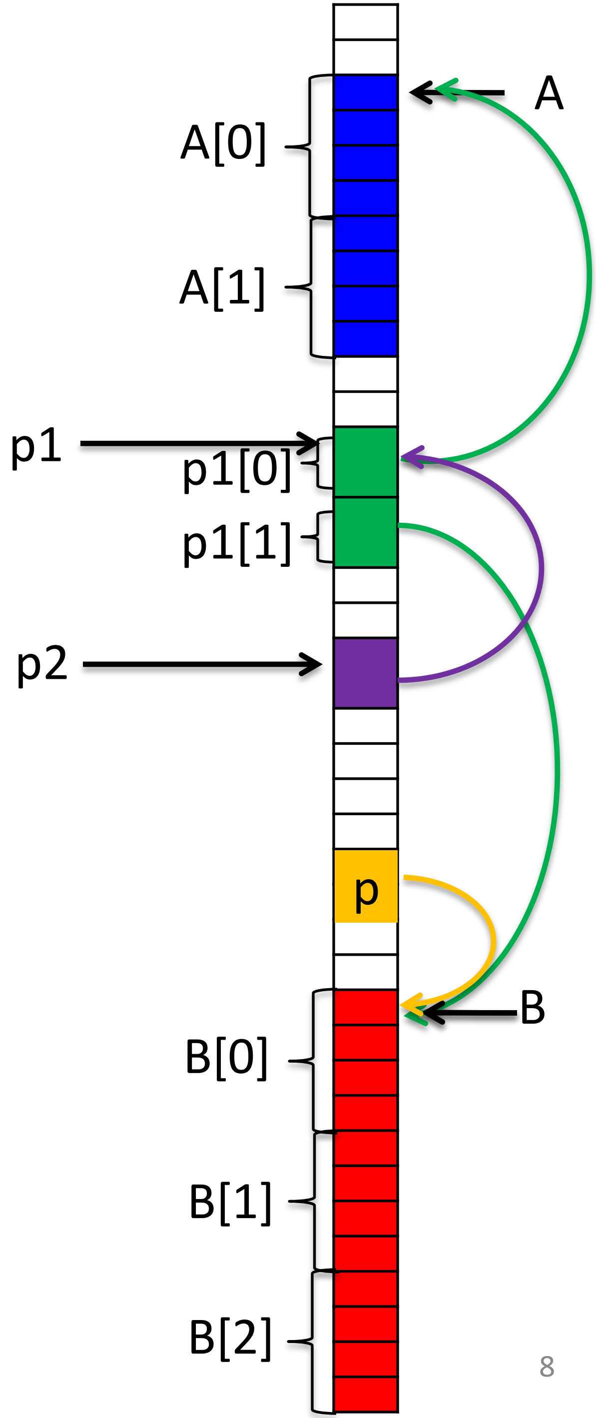
Pointers of pointers

```
float A[2] = { 1, 2 };
float B[3] = { 7, 1, 5};

float *p = B;

float *p1[2];
p1[0] = A;
p1[1] = B;

float **p2 = p1;
```



Pointers of pointers

```
float A[2] = { 1, 2 };
float B[3] = { 7, 1, 5};

float *p = B;

float *p1[2];
p1[0] = A;
p1[1] = B;

float **p2 = p1;

float f1 = p2[0][2]; // f1 = A[2] = ✗
float f2 = p2[1][2]; // f2 = B[2] = 5
float f3 = p2[0][1]; // f3 = A[1] = 2
```



Files Input/Output

Files I/O

- So far we have seen functions to read/write to command line (standard input/output)

- The same functions can be used to read/write to files

- (f)printf(), (f)scanf(), fgets()

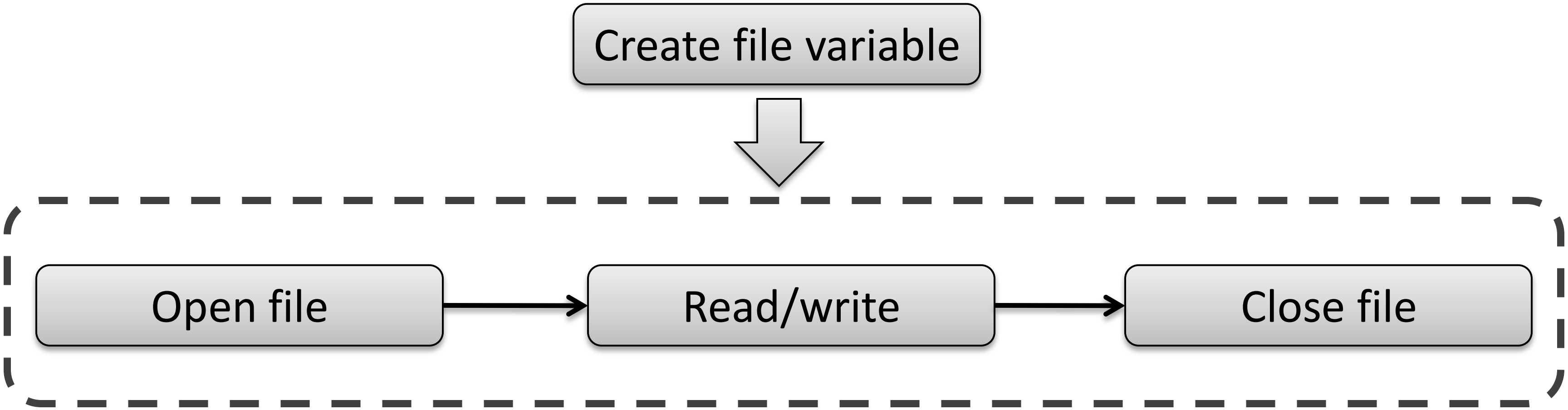
- All those functions are included in the `<stdio.h>` library

Files I/O Pipeline

- Files have a special type of variable associated with them: `FILE *`

- In order to read/write to a file, we must first OPEN it

- After we are done, we must CLOSE the file



Create file variable

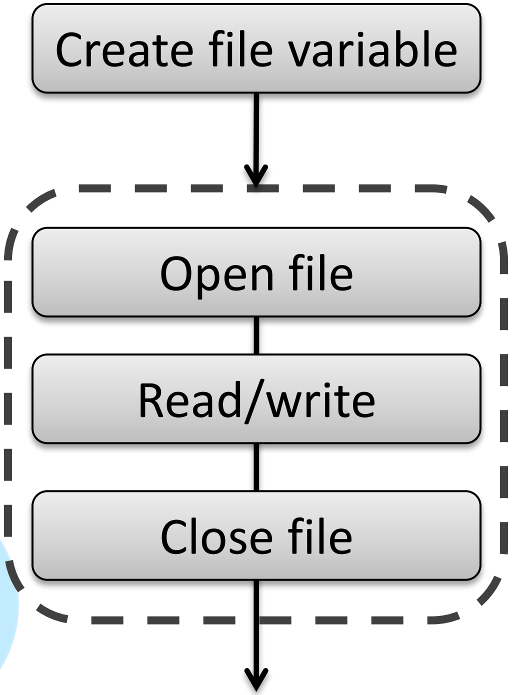
Open file

Read/write

Close file

Files I/O

- Files have a special type of variable associated with them: `FILE *`
- In order to read/write to a file, we must first OPEN it
- After we are done, we must CLOSE the file



Create file variable

Open file

Read/write

Close file

```
FILE *fVar;

fVar = fopen( fileName, mode);

/* read, write or append */

fclose(fVar);
```

fopen()

```
FILE * fopen( char *fileName, char *mode);
```

- `fileName` is a regular string with the name of the file
- `mode` determines the type of I/O we want to do
  - "r" : read
  - "w" : write, `fileName` is created if it did not exist
  - "a" : append, write to existing file, starting at the end
  - "b" : file is binary (associated with other modes, for example "wb" means write binary, "rb" read binary, etc.)
  - "r+" : read and write
  - "w+" : read and write , `fileName` is created if it did not exist
- In case of failure (for example trying to read from a non-existing file) fopen() returns NULL

fclose()

```
int fclose( FILE *fVar );
```

- `fVar` is a file variable (type FILE *)
- fclose() returns
  - 0 on success
  - non-zero for error

Stdin, stdout, stderr

- C provides 3 files (or filestreams) which are always open:
  - `stdin` : standard input, read from command line
  - `stdout` : standard output, write to command line
  - `stderr` : standard error, write to command line
- They are used as default values for various I/O functions

inOut.c

Read Functions

- fgetc() : read a single character

```
int fgetc( FILE *fVar )
```

Returns the special flag EOF if it has reached the end of the file

- fgets() : read a string, one line at a time

```
char* fgets( char* string, size_t size, FILE *fVar )
```

Returns `string` if successful, `NULL` is error or found EOF

inOut.c

Read Functions

- fscanf() : read a formatted line

```
int fscanf( FILE *fVar,“format1 … formatN”, &var1,…,&varN)
```

Reads one line from a file

Returns the number of variables successfully converted

inOut.c

Write Functions

- fputc() : write a single character

```
int fputc( char ch, FILE *fVar )
```

Returns ch if successful , the special flag EOF if there is an error

- fputs() : write a string

```
int fputs ( const char *string, FILE *fVar )
```

Returns a nonzero number if successful, EOF if there is an error

inOut.c

Write Functions

- fprintf() : print to file a formatted line

```
int fprintf( FILE *fVar,“format1 … formatN”, var1,…,varN)
```

Prints one line to a file

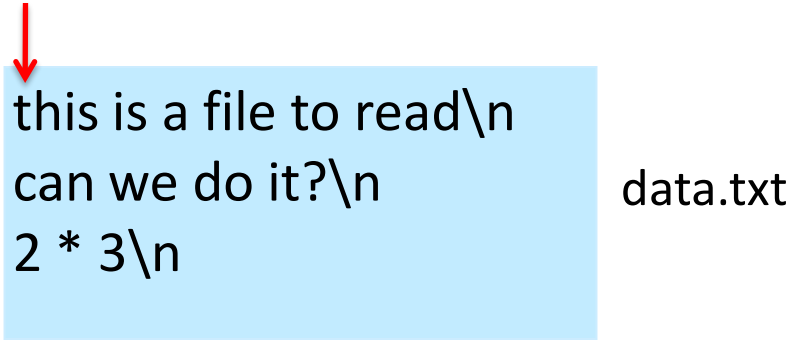
Returns the number of variables successfully converted

inOut.c

Read/Write to Files

- C has an internal pointer to the current position in the opened file
- After each read/write operation the pointer is updated

```
FILE *inFile = fopen("data.txt","r");
```

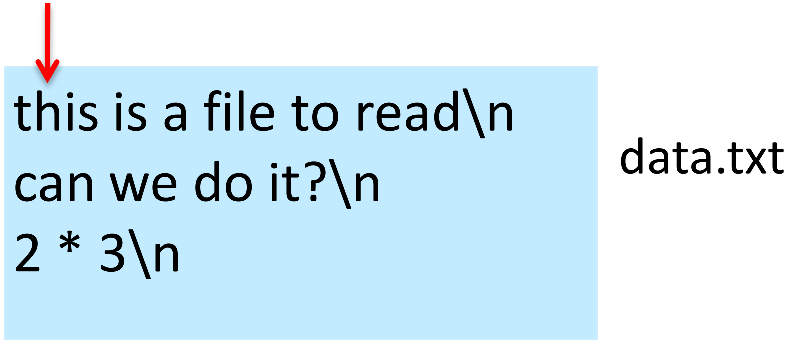


this is a file to read\n
can we do it?\n
2 * 3\n

data.txt

```
int ch = fgetc(inFile);
```

```
ch = 't'
```



this is a file to read\n
can we do it?\n
2 * 3\n

data.txt

feof()

- feof() checks if we reached the end of a file, without having to use fget(), fscanf() etc.

```
int feof( FILE *fVar )
```

Returns a value different from zero if reached end of file , zero otherwise

```
FILE *inFile = fopen( “data.txt” ,” r” );
```

```
while(1) {

    int ch = fgetc(inFile);

    if( ch == EOF ){
          break;
    }
}
```

```
while( !feof(inFile) ) {

    int ch = fgetc(inFile);

}
```

Summary of Functions

Name	Input	Output
fprintf()	formatted text + args	file
printf()	formatted text + args	stdout
sprintf()	formatted text + args	string
fputc(), fputs()	char,string	file
fscanf()	file	formatted text + args
scanf()	stdin	formatted text + args
sscanf()	string	formatted text + args
fgetc(), fgets()	file	(char) int, string

Buffered Output

- The OS does not write directly to a file stream
- For efficiency, it first prints to a buffer (= local place-holder in main memory)
- When the buffer is full, it prints it all to the file stream
- If we want to write in a specific moment, without buffering, we can us the function fflush()

```
int fflush( FILE *fVar )
```

Returns 0 if successful, EOF in the case of error

Buffered Output

```
printf(“starting\n”);


do_step1();
printf(“done with 1\n”);


do_step2();
printf(“done with 2\n”);


do_step3();
printf(“done with 3\n”);
```

```
printf(“starting\n”);
fflush(stdout);

do_step1();
printf(“done with 1\n”);
fflush(stdout);

do_step2();
printf(“done with 2\n”);
fflush(stdout);

do_step3();
printf(“done with 3\n”);
fflush(stdout);
```

Prints to buffer, after last printf() prints to stdout

After each printf() prints to stdout

File Formatting

- It is a good habit to create data files with HEADERS, especially when dealing with large amount of data
- HEADERS are one or two lines at the beginning of a file specifying the size of the data and some other info
- With headers, a program knows how to properly read a file

```
VectorTable
cols    7
rows    3
0       2       5       7       8       22      16
10      66      52      7       8       82      6
99      1       34      34      87      22      97
```

File Formatting

- It is a good habit to create data files with HEADERS, especially when dealing with large amount of data
- HEADERS are one or two lines at the beginning of a file specifying the size of the data and some other info
- With headers, a program knows how to properly read a file

header

```
VectorTable
cols     7
rows     3
0        2        5        7        8        22       16
10       66       52       7        8        82       6
99       1        34       34       87       22       97
```

File Formatting

- Ideally, format should be readable by humans and by computer programs

- Computer programs are not very robust, so must be specific (i.e. tab versus spaces)

- When you have huge amounts of data, you can give up on human-readability and use BINARY format for efficiency

- Example: color_histogram table

Binary Files

In order to read/write to binary files, we must use the “rb” / “wb” flags in the option of fopen()

```
size_t fread(void *ptr, size_t s, size_t n, FILE *f);

size_t fwrite(const void *ptr, size_t s, size_t n, FILE *f);
```

- ptr = (pointer) array where we want to store the data we read/ we want to write
- s = size of each element in the array ptr
- n = number of elements in the array ptr
- f = file to read from/write to

`size_t` is a C type to indicate the size (in bytes) of an element . You can think of it as a special integer.
For example, `sizeof()` returns a variable of type `size_t`

COMsW 1003-1

Introduction to Computer Programming in C

Lecture 14

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Announcements

Homework 4 out on Wednesday, due on Monday April 11th

Homework 3 solution out later today

Today

- Midterm Solution
- Finish FILE I/O (from Lecture 13)
- C standard libraries

Midterm Solution

Midterm Solution uploaded to Shared Files in Courseworks

Midterm Statistics

- Average grade: 72
- Standard deviation: 17

C Standard Libraries

- C provides a series of useful functions already implemented in standard libraries
- We have already seen some (stdio.h, string.h)
- In order to use the functions in a library, we must include the library header

```
#include <libraryName.h>
```

C Standard Libraries

C Standard Libraries

- stdio.h : input/output
- string.h : functions on strings
- stdlib.h : utility functions
- math.h : mathematical functions
- ctype.h : character class test
- assert.h : diagnostics
- limits.h and float.h : implementation-defined limits
- time.h : date and time functions
- A few more

C Standard Libraries

- stdio.h : input/output
- string.h : functions on strings
- stdlib.h : utility functions
- math.h : mathematical functions
- ctype.h : character class test
- assert.h : diagnostics
- limits.h and float.h : implementation-defined limits
- time.h : date and time functions
- A few more

testLibraries.c

stdio.h

- Standard input and output
- Input/output from command line (keyborad)
  - `fprintf(), fgets(), sscanf()`
- Input/output from files
  - `FILE, fopen(), fclose()`

testLibraries.c

string.h

Operations involving strings

```
string s1, s2;
char c;
```

- `int n = strcmp( s1, s2)` : compare s1 and s2, if(s1==s2) -> n = 0
- `int len = strlen(s1)` : return length of s1
- `char *pc = strchr(s1, c)`: return pointer to first occurrence of c in s1
- `char *ps = strstr(s1, s2)`: return pointer to first occurrence of string s2 in s1, or NULL if not present
- `char *strcpy(s1, s2)` : copy string s2 into s1, return s1
- `char *strcat(s1, s2)` : append s2 to s1 (concatenate), return s1
- `char *strtok(s1, s2)` : split long strings into pieces, or tokens

stdlib.h

testLibraries.c

Number conversions

- `float nf = atof(const char *s)` : converts string s to float
- `int n  = atoi(const char * s)` : convert string s to int

Memory allocation

`malloc(), free()` : memory management

Other utilities

- `int n = rand()` : returns a (pseudo) random int between 0 and constant RAND_MAX
- `void srand(unsigned int n)` : seeds rand generator
- `system(`*`string`*` s)` : runs ***s*** in OS

testLibraries.c

math.h

- Mathematical functions
- Often needs to be specially linked when compiling because takes advantage of specialized math hardware in processor

```
gcc -lm –Wall -o myProgram myProgram.c
```

```
double functionName(  double c )
```

- `sin(x), cos(x), tan(x)`
- `exp(x), log(x), log10(x)` : e^x , natural and base-10 logarithm
- `pow(x,y)` : x^y
- `sqrt(x)` : square root
- `ceil(x), floor(x)` : closest int above or below
- `y = fabs(x)` : absolute value , if x = -3.2, y will be 3.2

ctype.h

testLibraries.c

Utility functions to check for types of char

```
int functionName(  unsigned char c )
```

- `isalpha(c)` : check if c is an alphabet character ‘a’-’z’, ‘A’-’Z’
- `isdigit(c)` : check if c is digit ‘0’-’9’
- `isalnum(c)` : isalpha(c) or isdigit(c)
- `iscntrl(c)` : control char (i.e. \n, \t, \b)
- `islower(c) , isupper(c)` : lowercase/uppercase

Return value is 0 if false , != 0 if true

testLibraries.c

ctype.h

Utility functions to convert from lower case to upper case

```
char functionName(char c )
```

- `d = tolower(c)` : if c is ‘T’, d will be ‘t’
- `d = toupper(c)` : if c is ‘m’ , d will be ‘M’

limits.h and float.h

testLibraries.c

Contain various important constants such as the minimum and maximum possible values for certain types, sizes of types, etc.

- CHAR_BIT (bits in a char)
- INT_MAX, CHAR_MAX, LONG_MAX

  (maximum value of int, char, long int)
- INT_MIN, CHAR_MIN, LONG_MIN
- FLT_DIG (decimal digits of precision)
- FLT_MIN, FLT_MAX (min. and max. value of float)
- DBL_MIN, DBL_MAX (and of double precision float)

testLibraries.c

time.h

Provides new **type** to represent time, `time_t`

- `time_t time(NULL)` : returns current time
- `time_t clock()` : returns processor time used by program since beginning of execution
- strftime(A, sizeof(A), “formatted text”, time struct) :

format text with placeholders:

%a weekday
%b month
%c date and time
%d day of month
%H hour

testLibraries.c

assert.h

- Provides a macro to check if critical conditions are met during your program
- Nice way to test programs

```
assert( expression )
```

If the expression is false, the program will print to command line:

Assertion failed: `expression` , file *filename* , line *lineNumber*

More

- **stdarg.h** : allows you to create functions with variable argument lists
- **signal.h** - provides constants and utilities for standardized error codes for when things go wrong

COMsW 1003-1

Introduction to Computer Programming in C

Lecture 15

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Announcements

Homework 4 out, due April 11th at the beginning of class

Read CPL Chapter 5

Today

- Finish C Standard Libraries

- Pointers to void

- Begin Dynamic Memory Allocation

Review : operators * and &

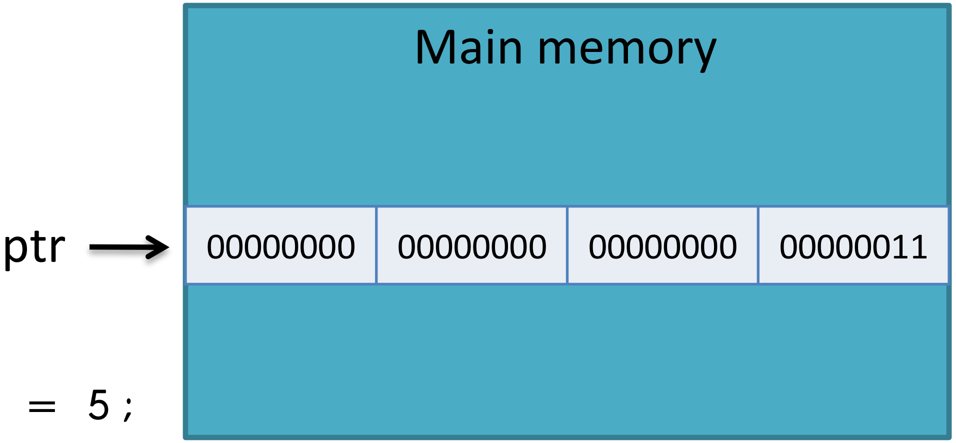
* **dereference operator** : gives the value in the memory pointed by a pointer (returns a value)

& **reference operator**: gives the address in memory of a variable (returns a pointer)

```
int x = 3;

int *ptr;

ptr = &x;


*ptr = 5; // x = 5;
```

Make `ptr` point to the address of `x`

Modify the value in address pointed by `ptr`



Review : Pointers of pointers

- A pointer can point to another pointer
- In a sense, it’s the equivalent of matrices!

```
int x = 3;

int *p = &x;

int **p2 = &p;

x = 2;   ⟷   *p = 2;   ⟷   **p2 = 2;


char *Arr[3]={ “Hello”, ”World”, ”Wonderful” };

char **ptr;

ptr = Arr;
```

Review: Pointers vs. Arrays

	Arrays		Pointers
1D array of 5 int	`int x[5];`	↔	`int *xPtr;`
2D array of 6 int 2x3 matrix	`int y[2][3];`	↔	`int **yPtr;`
2D array of 4 int 2x2 matrix	`int* z[2]={{1,2},{2,1}};`	↔	`int **zPtr;`
1D array of 5 char string	`char c[] = “mike”;`	↔	`char *cPtr;`

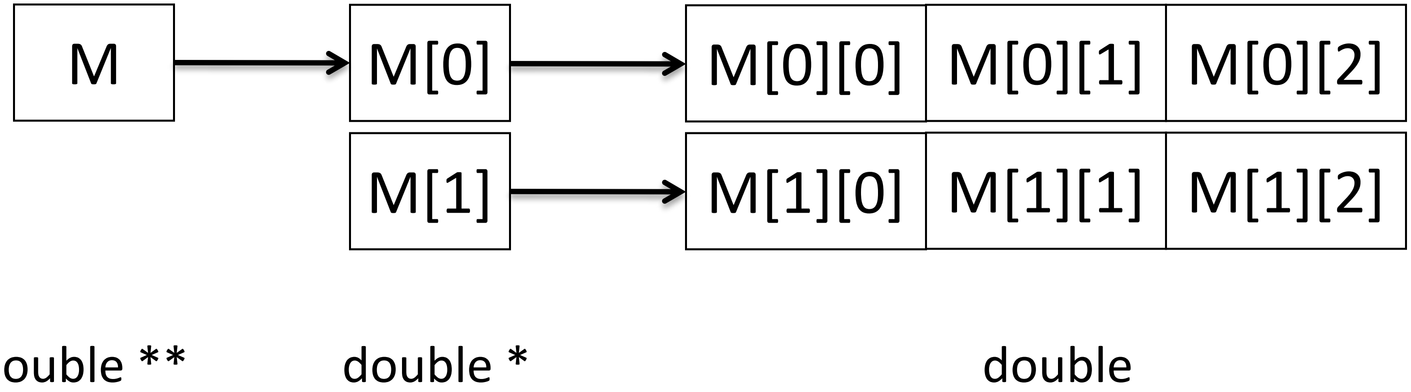
Space has been allocated in memory for the arrays

Space has been allocated in memory only for the pointers variables, **NOT** for the arrays they will point to.
The DIMENSIONS of the arrays are UNKNOWN

Multidimensional Arrays

2x3 matrix of double

```
double M0[2][3];
double *M1[2] = M0;
double **M = M0;
```



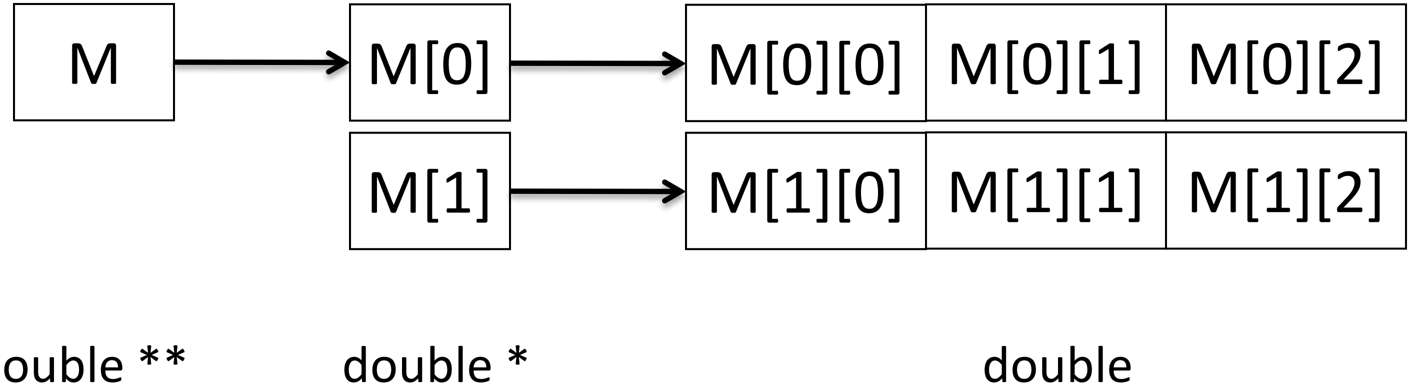
Multidimensional Arrays

2x3 matrix of double

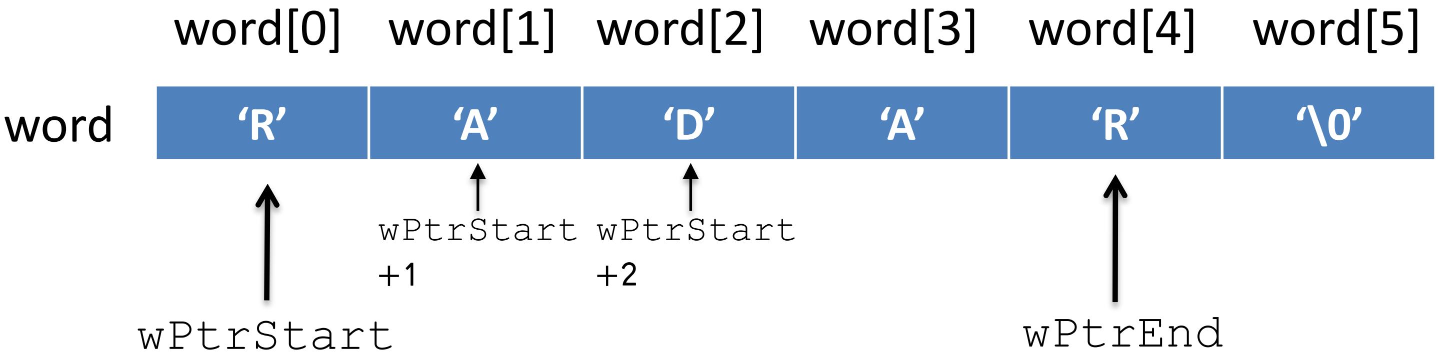
```
double M0[2][3];

double *M1[2] = M0;

double **M = M0;
```

The difference between M0, M1 and M is that

M1 and M can have ANY SIZE !



Review : Pointers and Arrays



	word[0]	word[1]	word[2]	word[3]	word[4]	word[5]
word	‘R’	‘A’	‘D’	‘A’	‘R’	‘\0’

`wPtrStart`

`wPtrStart` +1

`wPtrStart` +2

`wPtrEnd`

```
char word[8] = “RADAR”;

char *wPtrStart = word;
```

`char*` is a string

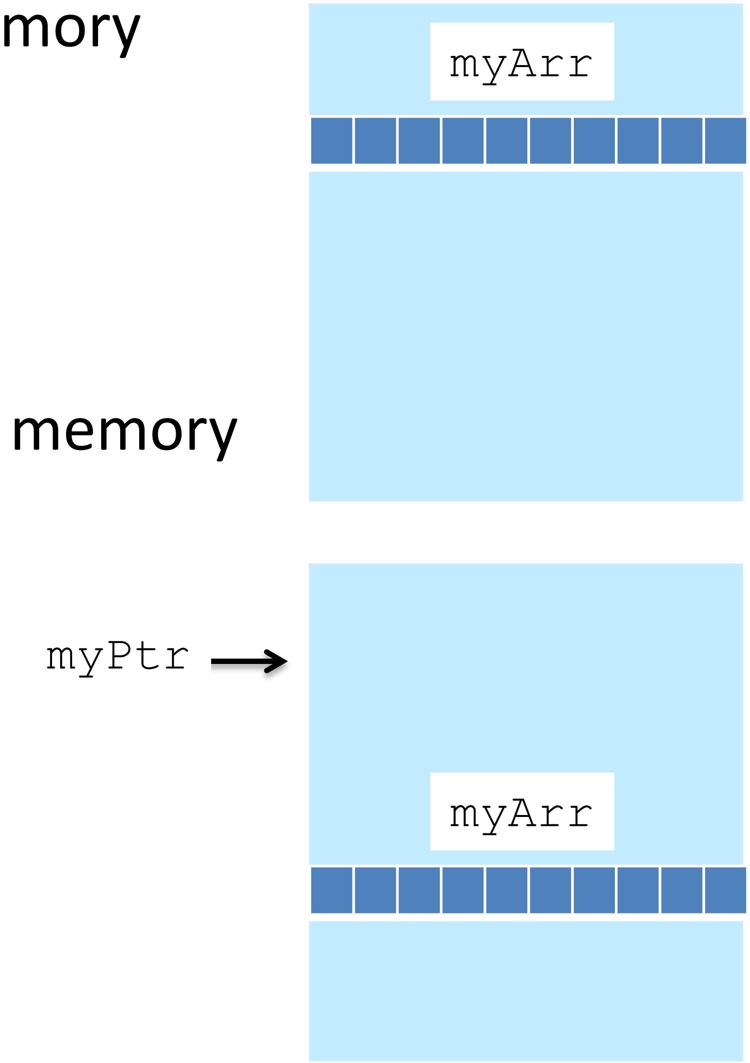
Pointers vs. Arrays

- Arrays represent actual memory **allocated** space

```
int myArr[10];
```

- Pointers **point** to a place in memory

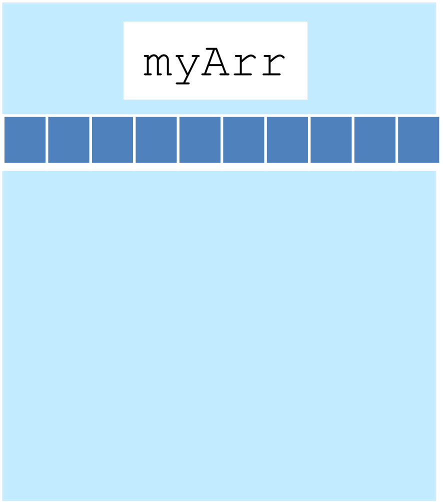
```
int *myPtr;
```



Pointers vs. Arrays

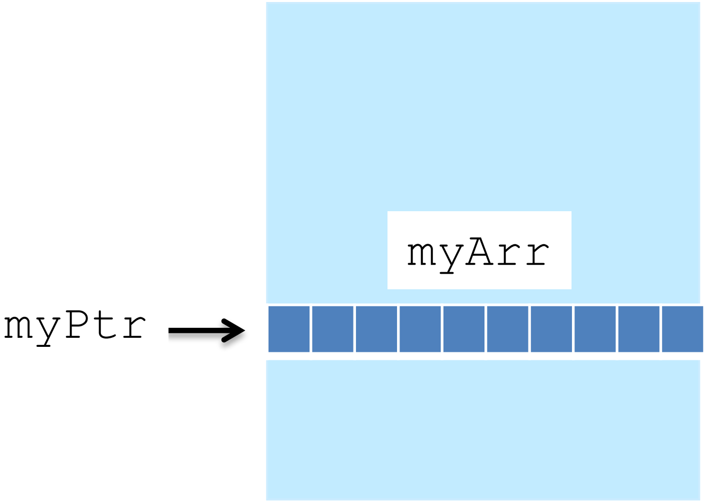
- Arrays represent actual memory **allocated** space

```
int myArr[10];
```



- Pointers **point** to a place in memory

```
int *myPtr;
myPtr = myArr;
```



sizeof()

- So far, we have been using sizeof() to determine the length of a string (including ‘\0’)

- sizeof() is a more general function, that returns the size, measured in bytes, of a variable or a type

```
size_t sizeof( var )
```

- `size_t` can be used (implicitly casted) as an integer

Void *

voidFun.c

`void *` means a pointer of ANY type

Sometimes functions can use `void *` as argument and return type.

This allows the programmer to specify the type of pointer to use at **invocation time**

This is a form of function overloading (popular in C++)

```
void *function_name( void *arg1, … , void *argN )
```

Void *

```
int i;
double d;
int *pi;
double *pd;

void *pv;
```

```
pi = &d;          // Compiler warning
pd = &i;          // Compiler warning

pv = &i;          // OK
printf("%d\n", *pv);   // Compiler error
printf("%d\n", *(int *)pv);     // OK

pv = &d;                       // OK
printf("%f\n", *pv);   // Compiler error
printf("%f\n", *(double *)pv); // OK

pv = &i;                     // OK
d = *(double *)pv;  // Runtime error
```

Void *

voidFun.c

Example

```
void *pointElement( void *A, int ind, int type ){

    if( type == 1 ){
        return( A + sizeof(int) * ind  );
    }
}

int main(){

   int M[3] = {1 , 2, 3};
   int element = 1;

   int *M2 = (int *) pointElement( M , element, 1);

}
```

Void *

voidFun.c

Example

```
void *pointElement( void *A, int ind, int type ){

   if( type == 1  ){
       return( A + sizeof(int) * ind  );
   }
}

int main(){

  int M[3] = {1 , 2, 3};
  int element = 1;

  int *M2 = (int *) pointElement( M , element, 1);

}
```

Explicit cast

Dynamic Memory Allocation

Functions related to DMA are in the library **stdlib.h**

```
void *malloc( size_t numBytes )
```

Allocates `numBytes` bytes in memory (specificaly, in a part of memory called heap)

The elements in the allocated memory are not initialized

Returns a pointer to the allocated memory on success, or NULL on failure

```
void *calloc( size_t numElements,  size_t size )
```

Allocates `size*numElements` bytes in memory

All elements in the allocated memory are set to zero

Returns a pointer to the allocated memory on success, or NULL on failure

Dynamic Memory Allocation

Example: create an array of 10 integers `int myArr[10];`

- Malloc()

  Example

  ```
  int *myArr = (int *) malloc( 10 * sizeof(int) );
  ```

- Calloc()

  Example

  ```
  int *myArr = (int *) calloc( 10 , sizeof(int) );
  ```

Dynamic Memory Allocation

Functions related to DMA are in the library **stdlib.h**

```
void *realloc(void *ptr, size_t size)
```

Changes the size of the allocated memory block pointed by *ptr* to *size*

Returns a pointer to the allocated memory on success, or NULL on failure

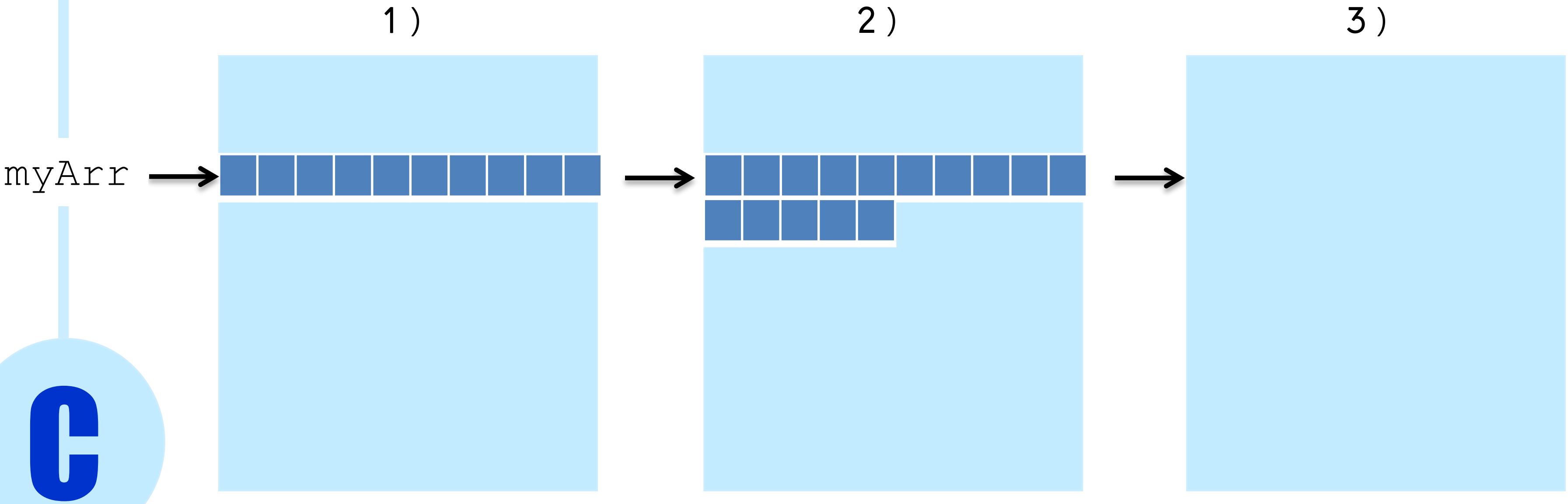
```
void free(void *ptr)
```

De-allocates (frees) the space in memory pointed by `ptr`

Dynamic Memory Allocation

Example: create an array of 10 integers, resize it to 15, then free the space in memory

```
1) int *myArr = (int *) malloc( 10 * sizeof(int) );

2) myArr = realloc( myArr, 15 * sizeof(int) );

3) free( myArr );
```



commandLine.c

Dynamic Memory Allocation

Example: reading an indefinitely long command line

So far we have been reading strings from command line using an array

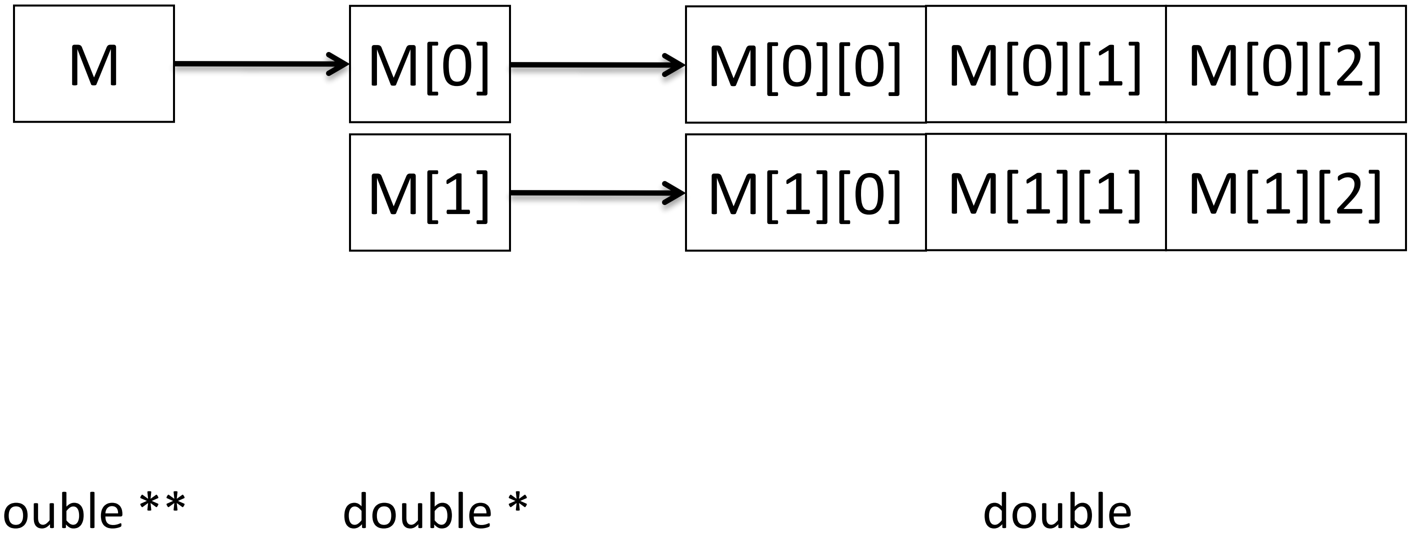
```
char line[100];
fgets( line, sizeof(line), stdin);
```

What if the user enters a command with 105 characters?

Dynamic Memory Allocation

Multidimensional Arrays

2x3 matrix of double



M

M[0]

M[1]

M[0][0] M[0][1] M[0][2]

M[1][0] M[1][1] M[1][2]

double **

double *

double

Dynamic Memory Allocation

Multidimensional Arrays

2x3 matrix of double

```
double** M = (double**) malloc( 2 * sizeof(double *) );

int i;
for ( i = 0 ; i<2; i++ ){
    M[i] = malloc( 3 * sizeof(int) );
}

/* use M as a regular 2-dimensional array */

for ( i = 0 ; i<2; i++ ){
    free( M[i] );
}
free( M );
```

Memory Leaks

Space in the heap is LIMITED, therefore we must be careful and free memory

There are two cases in whish freeing memory becomes impossible:

- when we move a pointer after allocating memory

```
int N = 40000;

char *str = “Hello”;

char *giantString = malloc(N*sizeof(char));

giantString = str;
```

Now we cannot find anymore the location of the block of allocated memory

Memory Leaks

Space in the heap is LIMITED, therefore we must be careful and free memory

There are two cases in whish freeing memory becomes impossible:

- if we reallocate memory using the same pointer

```
int N = 40000;

char *giantString = malloc(N*sizeof(char));

/* do something */

giantString = malloc(N*sizeof(char));
```

`giantString` now points to a newly allocated block of memory, the location of the previous one is lost

COMSW 1003-1

Introduction to Computer Programming in C

Lecture 17

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Review - Arrays of strings

- An array Arr of 3 strings of variable length

```
char *Arr[3]={ “Hello”, ”World”, ”Wonderful” };

Arr[2] = Arr+2 // ”Wondeful”
```

- An array Arr of 3 strings of maximum length = 15

```
char Arr2[3][15] = { “Hello2”, ”World2”, ”Wonderful2” };

Arr2[0] = Arr2 // ”Hello2”

Arr2[1] = Arr2+1 // ”World2”
```

Program’s Inputs

- When we run a program, sometimes we want to pass some input arguments to it
- This can be done by writing them in the command line, immediately after the program name
- The program’s inputs must be **separated by spaces**

Example

The program sumTwoNumbers sums two numbers.

We can pass the two input numbers directly when we invoke the program’s executable (instead of the usual I/O operations, such as printing to command line the message “please insert two numbers:”, followed by fgets() etc.)

```
./sumTwoNumbers  3  5
```

Clarguments.c

Command Line Arguments

- Input parameters of the function main()
- `argc, argv`

```
int main( int argc, char* argv[] )
```

`argc`	• Integer • Specifies the **number** of arguments on the command line (including the program name)
`argv`	• Array of strings • Contains the actual **arguments** on the command line • First element is the name of the program

Command line arguments

It is a good habit, especially when a program takes input arguments, to specify in a **header** on the top of the main file:

- Program name and purpose
- Program usage: syntax to use to invoke (run) the program with input arguments
- Description of input arguments
- Description of output from the program

It is common to add a **–help** option to print the relevant information about program usage and input arguments calculator.c

Command line arguments

Example

Program calculator, reads two numbers, the operator, and prints the result

Linux Wildcard Characters

Linux has a series of wildcard characters * ? [ ]

* Represents strings of arbitrary length containing any possible character

***** all items (directories and files) - with or without a suffix

r* items beginning with the letter "r"

boot* items beginning with "boot"

mem all items contain "mem" anywhere in the name

***.png** items having the suffix of ".png" - that end in ".png"

We must be very careful when we use wildcard characters as input, because argc and argv recognize them!

Linux Wildcard Characters

Linux has a series of wildcard characters * ? [ ]

? Represents one single character which has any possible value

?.txt items starting with only one character and ending in ".txt"
Examples: b.txt and 3.txt

memo?.sxw items beginning with "memo", having a single character after "memo", and having the suffix of ".sxw"
Examples: memo1.sxw and memoh.sxw - not memo23.sxw

memo??.sxw items beginning with "memo", having a two characters (only) after "memo", and having the suffix of ".sxw"
Examples: memo21.sxw and memok9.sxw - not memos.sxw

We must be very careful when we use wildcard characters as input, because argc and argv recognize them!

Linux Wildcard Characters

Linux has a series of wildcard characters * ? [ ]

[ ] Represents intervals of characters values

[a-z]* items that begin with any lower case letter and end in any other characters

[A-Z]-list.dat items that begin with any upper case letter and end in "-list.dat"

[a-zA-Z]report.sxc items that begin with any lower case or upper case letter and end in "report.sxc“

[e-t].c items that begin with any lower case letter between ‘e’ and ‘t’ and end in “.c”

We must be very careful when we use wildcard characters as input, because argc and argv recognize them!

Homework 3 Solution

COMsW 1003-1

Introduction to Computer Programming in C

Lecture 18

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Modular Programming

Review - Header files

- Header files are fundamentally libraries
- Their extension is .h
- They contain function definitions, variables declarations, macros
- In order to use them, the preprocessor uses the following code

```
#include <nameOfHeader.h>

#include “nameOfHeader.h”
```

→ For standard C libraries

→ For user defined headers

- So far, we have used predefined C header files, but we can create our own! (more on this next week)

Modular Programming

- So far we have seen only small programs, in one single file

- What about bigger programs? Need to keep them organized, especially if multiple people work on the same project

- They are organized in multiple, organized parts : MODULES

Modules

- A module is “a collection of functions that perform related tasks” [PCP Ch18]

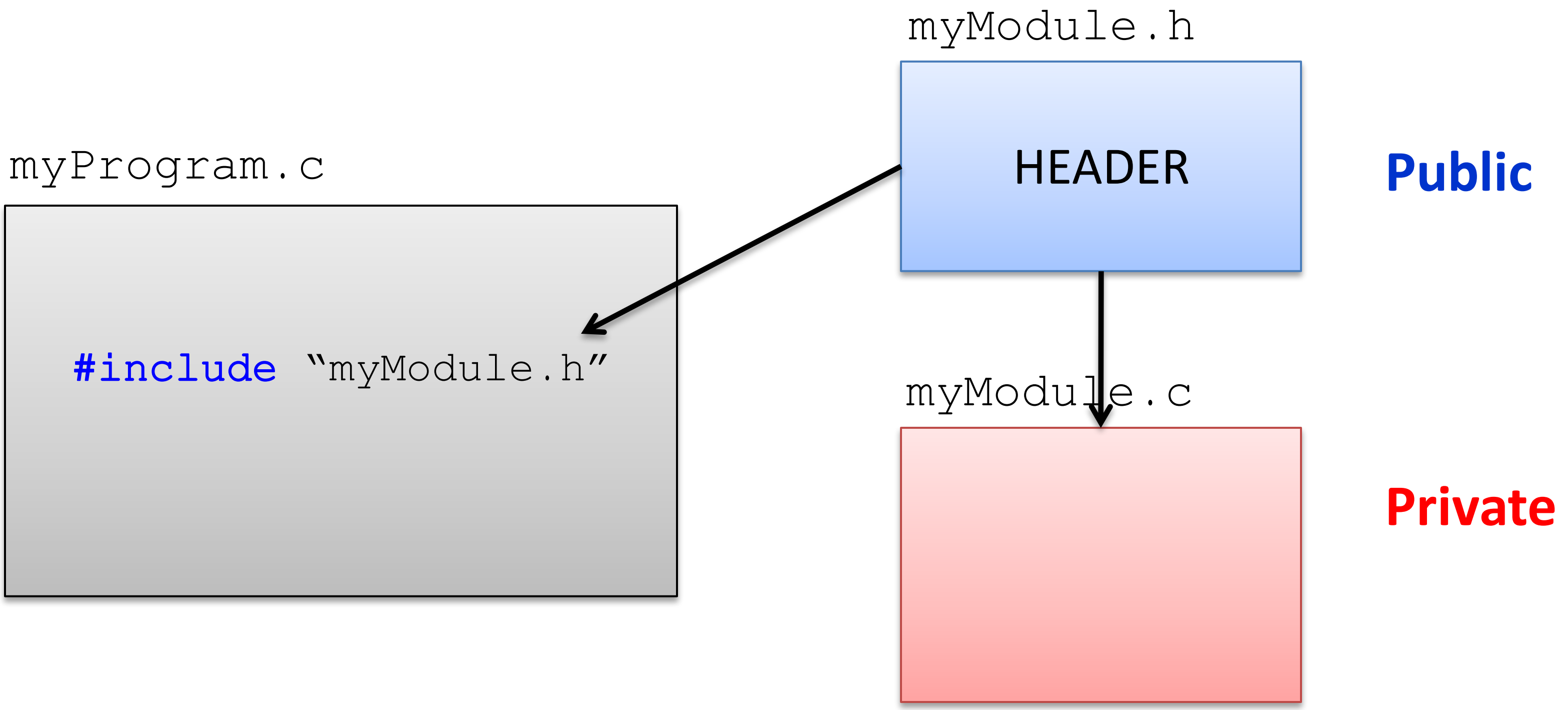
- A module is basically a **user defined library**

- Two parts:
  - Public : tells the user how to use the functions in the module. Contains declaration of data structures and functions

  - Private : implements the functions in the module

Modules

- Two parts:
  - **Public** : tells the user how to use the functions in the module. Contains definition of data structures and functions
  - **Private** : implements the functions in the module



Header

- A header should contain:
  - A section describing what the module does
  - Common constants
  - Common structures
  - Public functions declarations
  - **Extern** declarations for public variables

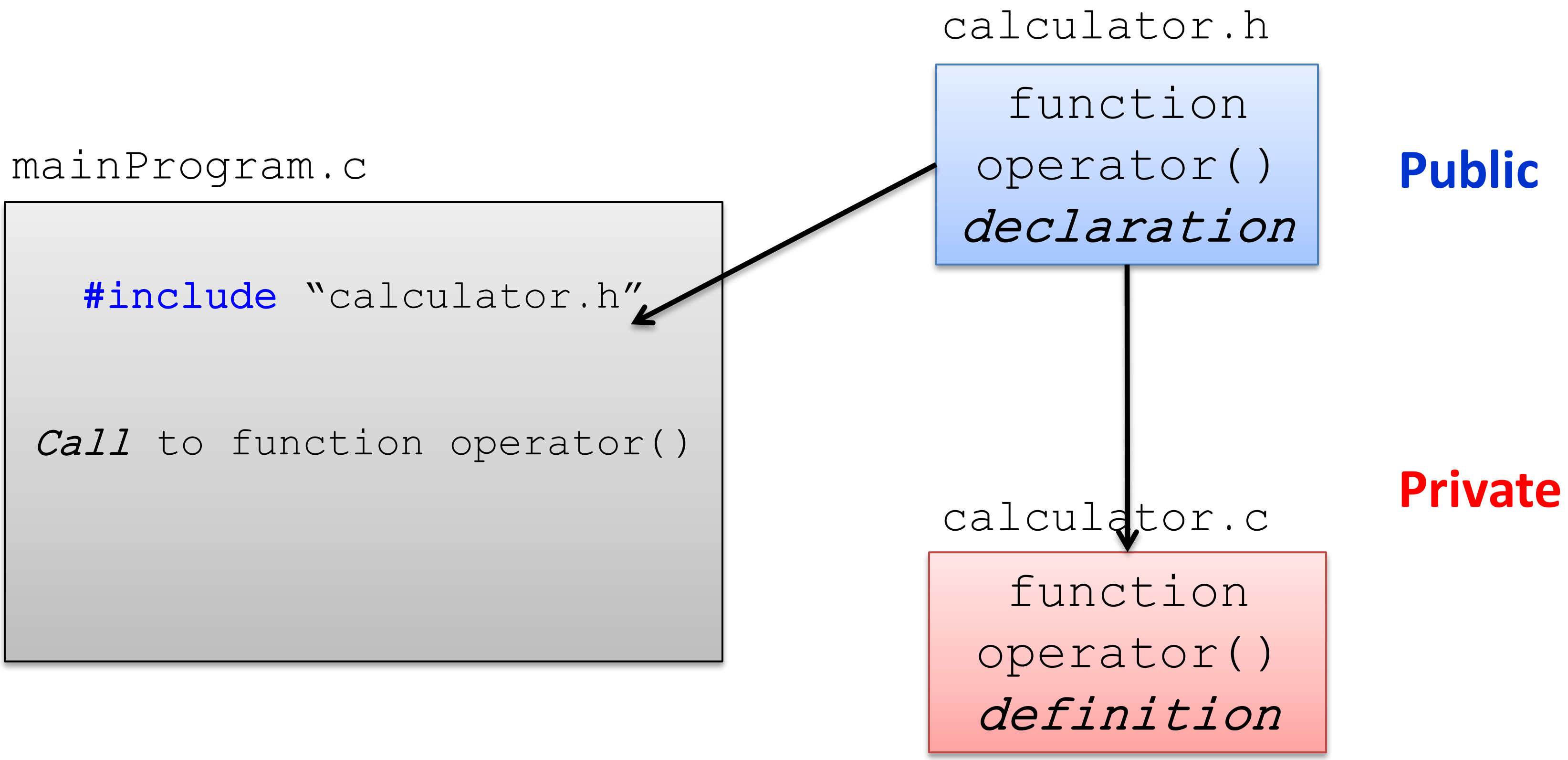
Function Declaration vs. Definition

- All identifiers in C need to be declared before they are used, including functions
- Function declaration needs to be done before the first call of the function
- The **declaration** (or **prototype**) includes
  - return type
  - number and type of the arguments

- The function **definition** is the actual implementation of the function
- Function definition can be used as implicit declaration

Modules

mainProgram.c
calculator.h
calculator.c



`mainProgram.c`

```
#include "calculator.h"

Call to function operator()
```

`calculator.h`

`function operator()` *declaration*

Public

`calculator.c`

`function operator()` *definition*

Private

Compile modules together

- We need a way to “glue” the modules together
- We need to compile not only the main program file, but also the user defined modules that the program uses
- Solution : makefile

Makefile

- `make` routine offered in UNIX (but also in other environments)

- `make` looks at the file named *Makefile* in the same folder and invokes the compiler according to the **rules** in *Makefile*

Makefile – Single file

```
#-----------------------------------------------------------#
#     Makefile for UNIX system                              #
#     using a GNU C compiler (gcc)                          #
#-----------------------------------------------------------#

# this is a comment

oldCalculator: oldCalculator.c
      gcc –Wall -o oldCalculator oldCalculator.c
```

Makefile – Single file

```
#-------------------------------------------------------#
#     Makefile for UNIX system                          #
#     using a GNU C compiler (gcc)                      #
#-------------------------------------------------------#
```

```
# this is a comment
```

Comments start with a # sign

```
oldCalculator: oldCalculator.c
	gcc –Wall -o oldCalculator oldCalculator.c
```

Rule: gcc command we are used to

The second statement MUST start with a TAB!

Makefile – Single file

```
#-----------------------------------------------------#
#     Makefile for UNIX system                        #
#     using a GNU C compiler (gcc)                    #
#-----------------------------------------------------#

CC=gcc
CFLAGS=-Wall

oldCalculator: oldCalculator.c
	$(CC) $(CFLAGS) -o oldCalculator oldCalculator.c
```

Makefile – Single file

```
#---------------------------------------------------#
#     Makefile for UNIX system                      #
#     using a GNU C compiler (gcc)                  #
#---------------------------------------------------#
```

```
CC=gcc
CFLAGS=-Wall
```

macros

```
oldCalculator: oldCalculator.c
	$(CC) $(CFLAGS) -o oldCalculator oldCalculator.c
```

Rule: gcc command we are used to

The second statement MUST start with a TAB!

Makefile

- Macros

```
name=data
$(name) ⟶ data
```

Whenever $(name) is found, it gets substituted with data
Same as object-type macros for Preprocessor

- Rules

```
target: source [source2] [source3] …
        command
        command2
        command3
           ⋮
```

UNIX compiles `target` from `source` using `command`

Default command is `$(CC) $(CFLAGS) -c source`

↘ ↙
Predefined by make

Makefile – Single file

```
#-------------------------------------------------------#
#     Makefile for UNIX system                          #
#     using a GNU C compiler (gcc)                      #
#-------------------------------------------------------#

CC=gcc
CFLAGS=-Wall

oldCalculator: oldCalculator.c
	$(CC) $(CFLAGS) -o oldCalculator oldCalculator.c

clean:
	rm -f oldCalculator
```

Makefile – Single file

```
#-------------------------------------------------------#
#     Makefile for UNIX system                          #
#     using a GNU C compiler (gcc)                      #
#-------------------------------------------------------#
```

```
CC=gcc
CFLAGS=-Wall
```

macros

```
oldCalculator: oldCalculator.c
	$(CC) $(CFLAGS) -o oldCalculator oldCalculator.c
```

Rule: gcc command we are used to

The second statement MUST start with a TAB!

```
clean:
	rm -f oldCalculator
```

Rule: Clean up files

Makefile

- If I have multiple rules, I can use the name of the target to execute only the rule I want
- By default, make executes only the first rule

Example

```
$make clean
```

Makefile – Multiple Modules

```
#---------------------------------------------------------#
#     Makefile for UNIX system                            #
#     using a GNU C compiler (gcc)                        #
#---------------------------------------------------------#

CC=gcc
CFLAGS=-Wall

mainCalc : mainProgram.c  calculator.o
       $(CC) $(CFLAGS) -o mainCalc mainProgram.c calculator.o

calculator.o : calculator.c calculator.h
       $(CC) $(CFLAGS) -c calculator.c

clean:
       rm -f calculator.o mainProgram
```

Makefile – Multiple Modules

```
#-------------------------------------------------------#
#     Makefile for UNIX system                          #
#     using a GNU C compiler (gcc)                      #
#-------------------------------------------------------#

CC=gcc
CFLAGS=-Wall

mainCalc : mainProgram.c  calculator.o
       $(CC) $(CFLAGS) -o mainCalc mainProgram.c calculator.o

calculator.o : calculator.c calculator.h
       $(CC) $(CFLAGS) -c calculator.c

clean:
       rm -f calculator.o mainCalc
```

We must use the –c option to compile a module instead of an executable!

Makefile

- Rules

```
target:  source [source2] [source3] …
         command
         command2
         command3
              ⋮
```

UNIX compiles `target` from `source` using `command`

Default command is `$(CC) $(CFLAGS) -c source`

make is smart: it compiles only modules that need it

If target has already been compiled and source did not change, make will skip this rule

```
target:
         command
```

This rule instead is ALWAYS executed by the compiler, because source is not specified in the first line

calculator.h
calculator.c

Extern/Static Variables

- `Extern` is used to specify that a variable or function is **defined outside** the current file

  When same variable is used by different modules, `extern` is a way to declare a global variable which can be used in all modules

- `Static` is used to specify that a variable is local to the current file (for global variables)

  Remember the use for local variables (Lec7): local static means permanent

COMSW 1003-1

Introduction to Computer Programming in C

Lecture 19

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Basic Data Structures

Basic Data Structures

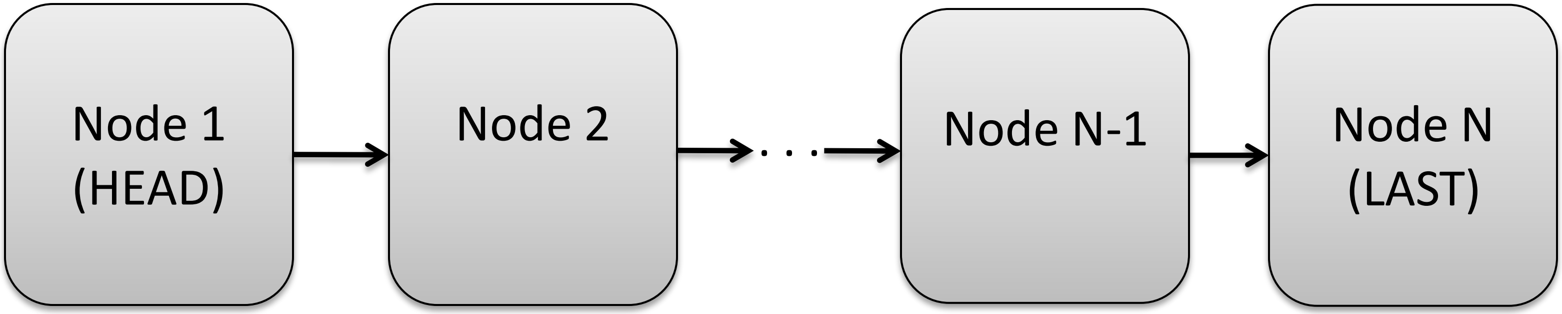
- So far, the only data structures we have seen to store data have been arrays (and structs)

- There are other (and potentially more useful) data structures that can be used
  - Lists
  - Trees

- Benefits:
  - Dynamically grow and shrink is easy
  - Search is faster

Linked Lists

- A chain of elements
- First element is called HEAD
- Each element (called NODE) points to the next
- The last node does not point to anything
- Like a treasure hunt with clues leading one to another



structPoint.c

Pointers to structs

- Pointers can point to any type, including structs
- There is a particular way of accessing fields in a struct through a pointer: the **->** operator

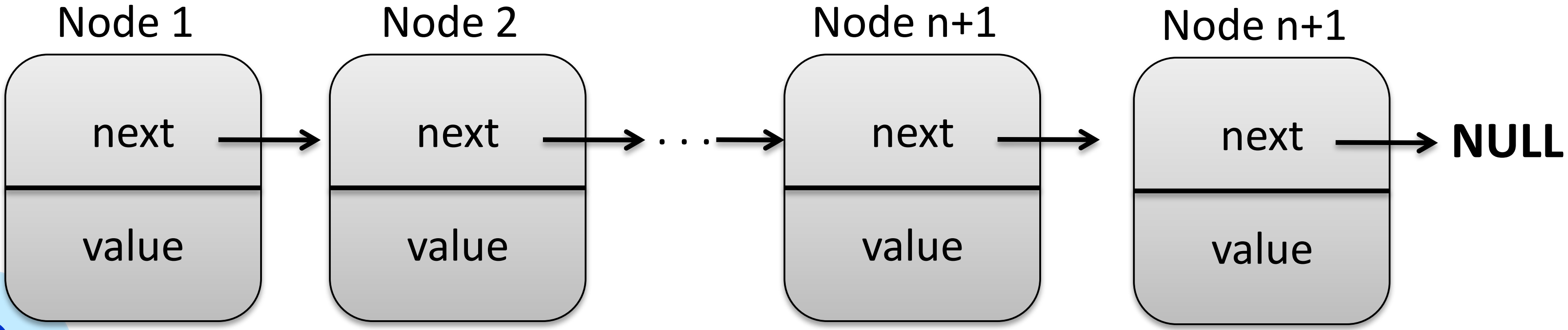
```
struct person {
        int age;
        char *name;
}

struct person p1 = {15, "Luke"};
struct person *ptr = &p1;
ptr->age = 20;              // (*ptr).age = 20;
printf("%s\n", ptr->name);
```

linkedList.c

Linked Lists

- Structure declaration for a node of a linked list

```
struct ll_node {
        int value;
        struct ll_node *next;
};
typedef struct ll_node node;
```



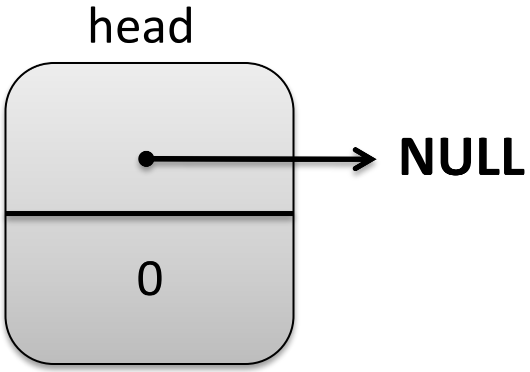
Linked Lists Initialization

```
struct ll_node {
    int value;
    struct ll_node *next;
};
```

```
node *head = (node *) malloc(sizeof(node));
head->value = 0;
head->next = NULL;
```

- First node (HEAD) of the list is just a pointer to the list, it not counted as an actual node in the list
- Value set to 0 (could be any number, maybe a counter)
- The list is still empty, there is only HEAD, so next is NULL (end of the list)

Linked Lists Initialization



```
node *head = (node *) malloc(sizeof(node));
head->value = 0;
head->next = NULL;
```

- First node (HEAD) of the list is just a pointer to the list, it not counted as an actual node in the list
- Value set to 0 (could be any number, maybe a counter)
- The list is still empty, there is only HEAD, so next is NULL (end of the list)

Linked Lists

Insert node in front

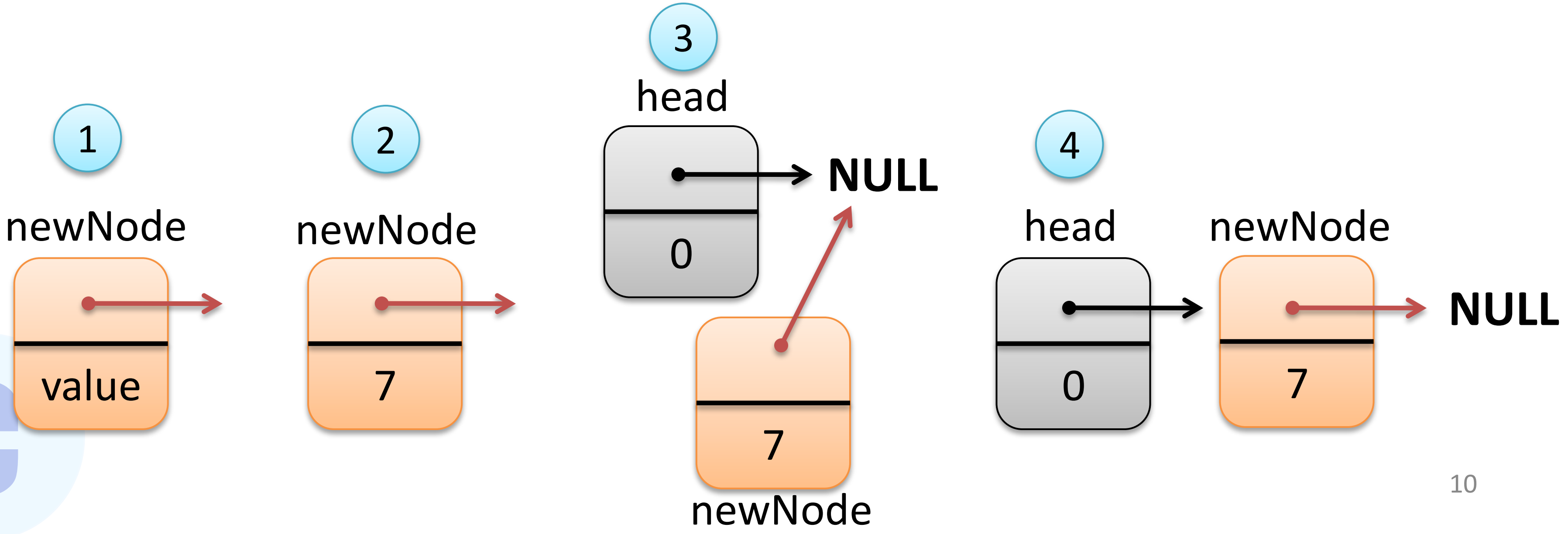
```
struct ll_node {
    int value;
    struct ll_node *next;
};
```

```
int addNodeFront( int val, node *head ){
    node *newNode = (node *) malloc(sizeof(node));
    newNode->value = val;
    newNode->next = head->next;
    head->next = newNode;
    return 0;
}
```

Linked Lists - Insert node in front

```
int addNodeFront( int val, node *head ){
  1) node *newNode = (node *) malloc(sizeof(node));
  2) newNode->value = val;
  3) newNode->next = head->next;
  4) head->next = newNode;
     return 0;
}
```

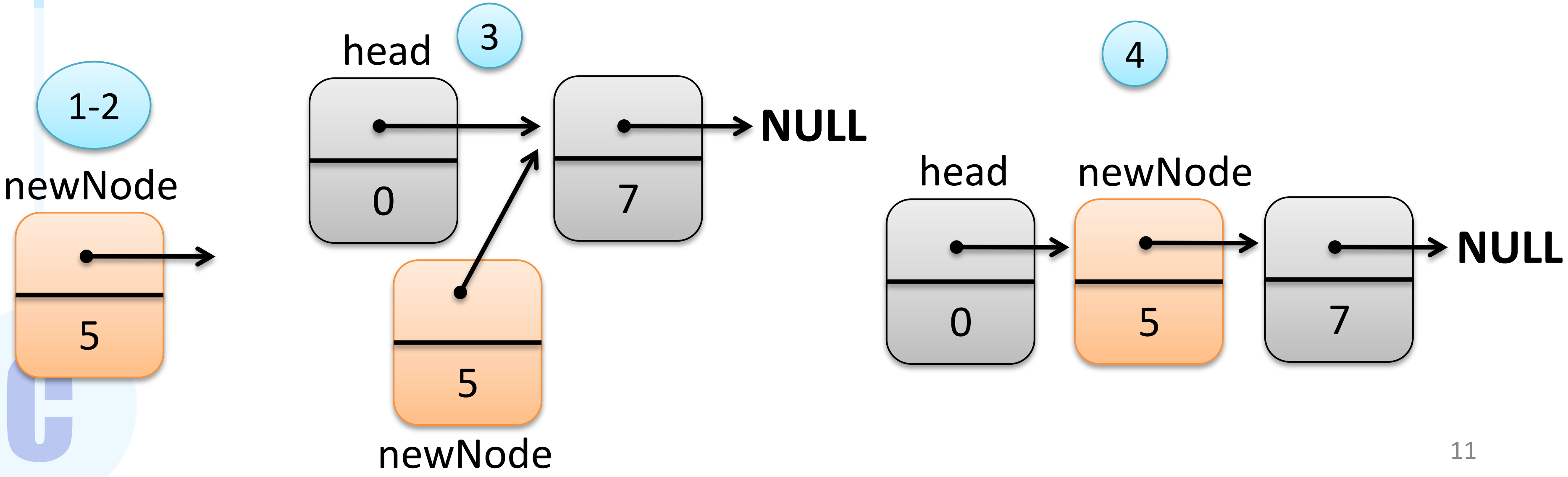
```
addNodeFront( 7, head );
```



Linked Lists - Insert node in front

```
int addNodeFront( int val, node *head ){
   1) node *newNode = (node *) malloc(sizeof(node));
   2) newNode->value = val;
   3) newNode->next = head->next;
   4) head->next = newNode;    return 0;
}
```

```
addNodeFront( 7, head );
addNodeFront( 5, head );
```



Linked Lists
Insert node at position N

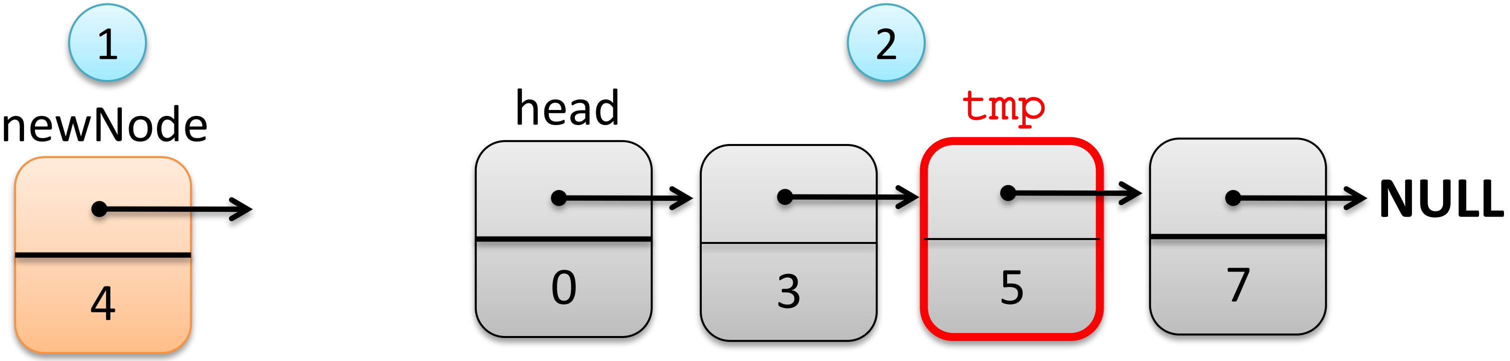
```
struct ll_node {
    int value;
    struct ll_node *next;
};
```

```
int addNode( int val, node *head, int pos ){
    node *newNode = (node*) malloc( sizeof(node) );
    newNode->value = val;
    int i;
    node *tmp = head;
    for(i=0 ; i<pos; i++)
          tmp = tmp->next;
    newNode->next = tmp->next;
    tmp->next = newNode;
    return 0;
}
```

Linked Lists - Insert node at position N

```
int addNode( int val, node *head, int pos ){
 1)   node *newNode = (node*) malloc( sizeof(node) );
      newNode->value = val;
 2)   node *tmp = head;
      for(i=0 ; i<pos; i++)
          tmp = tmp->next;
 3)   newNode->next = tmp->next;
 4)   tmp->next = newNode;
      return 0;
}
```

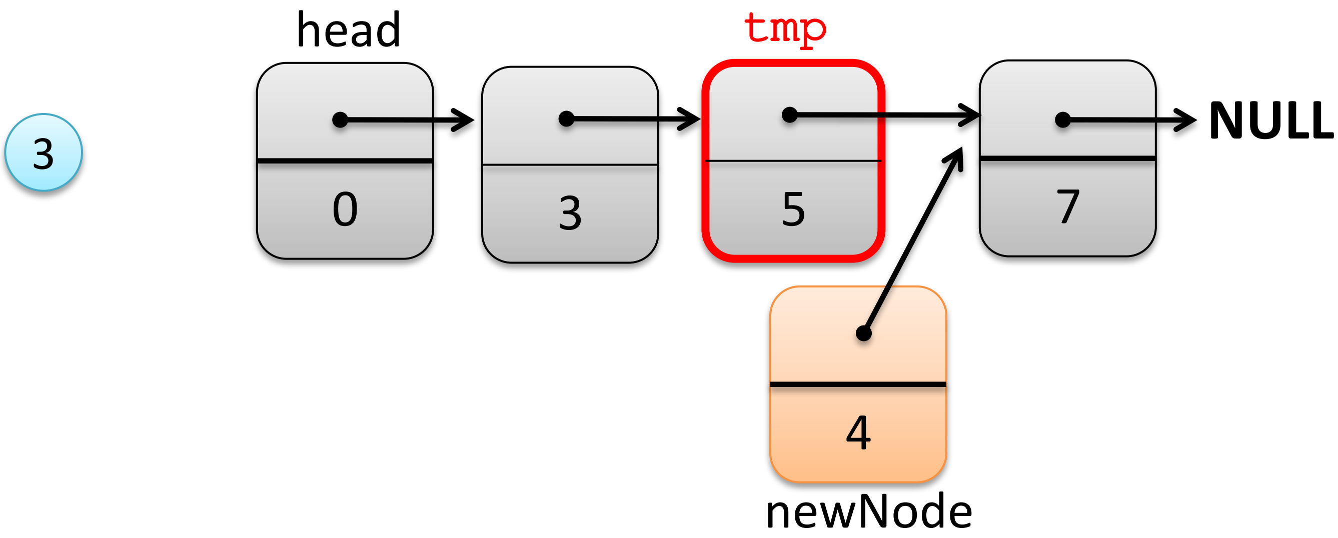
```
addNode( 4, head, 2 );
```



Linked Lists - Insert node at position N

```
int addNode( int val, node *head, int pos ){
 2)   node *tmp = head;
      for(i=0 ; i<pos; i++)
          tmp = tmp->next;
 3)   newNode->next = tmp->next;
 4)   tmp->next = newNode;
      return 0;
}
```

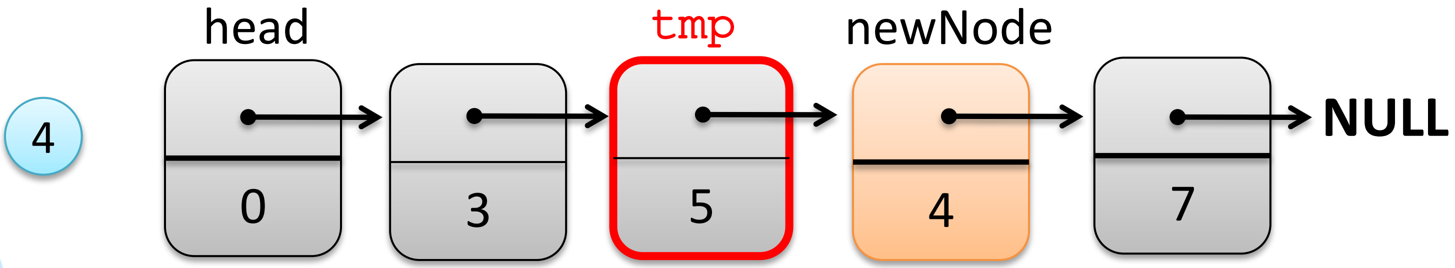
`addNode( 4, head, 2 );`



Linked Lists - Insert node at position N

```
int addNode( int val, node *head, int pos ){
     node *tmp = head;
 2)  for(i=0 ; i<pos; i++)
         tmp = tmp->next;
 3)  newNode->next = tmp->next;
 4)  tmp->next = newNode;
     return 0;
}
```

`addNode( 4, head, 2 );`



Linked Lists
Delete Node

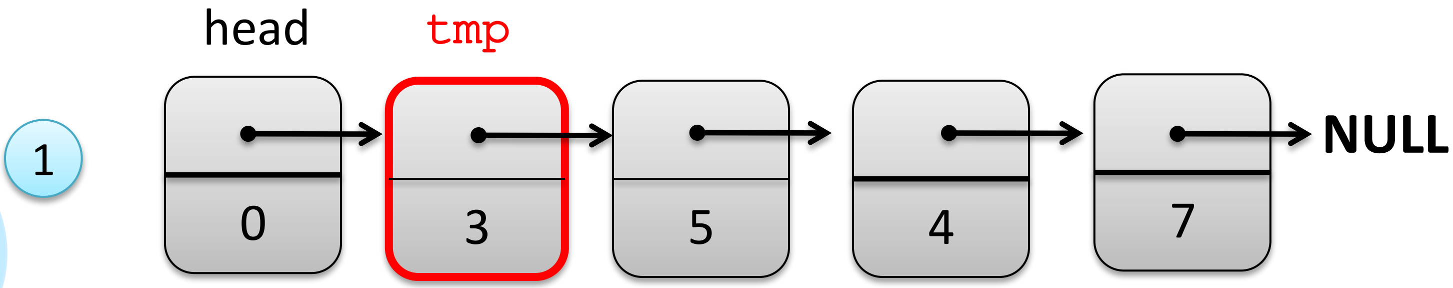
```
struct ll_node {
    int value;
    struct ll_node *next;
};
```

```
int removeNodePosition( node *head, int pos ){
    int i;
    node *tmp = head;
    for(i=0 ; i<pos; i++)
        tmp = tmp->next;
    node* tmp2 = tmp->next;
    tmp->next = tmp->next->next;
    free(tmp2);
    return 0;
}
```

Linked Lists - Delete Node

```
int removeNodePosition( node *head, int pos ){
     int i;
 1)  node *tmp = head;
     for(i=0 ; i<pos; i++)
        tmp = tmp->next;
 2)  node* tmp2 = tmp->next;
     tmp->next = tmp->next->next;
 3)  free(tmp2);
     return 0;
}
```

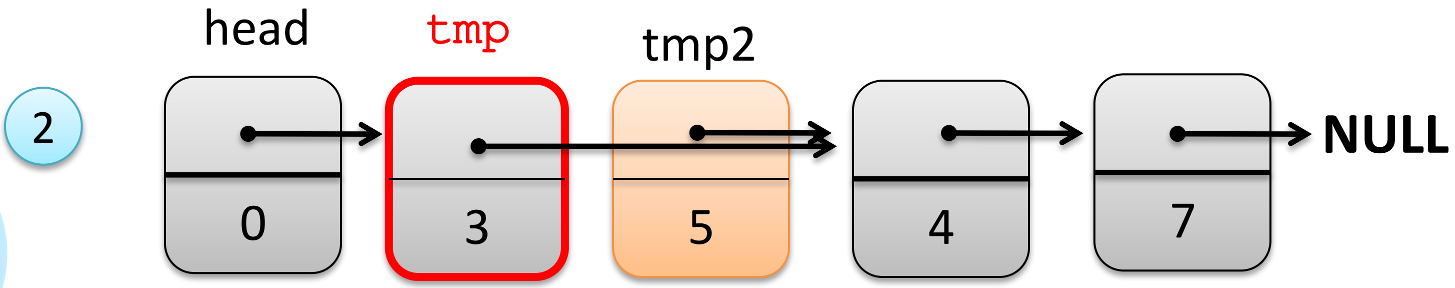
```
removeNode( head, 1 );
```



Linked Lists - Delete Node

```
int removeNode( node *head, int pos ){
        int i;
 1)     node *tmp = head;
        for(i=0 ; i<pos; i++)
            tmp = tmp->next;
 2)     node* tmp2 = tmp->next;
        tmp->next = tmp->next->next;
 3)     free(tmp2);
        return 0;
}
```

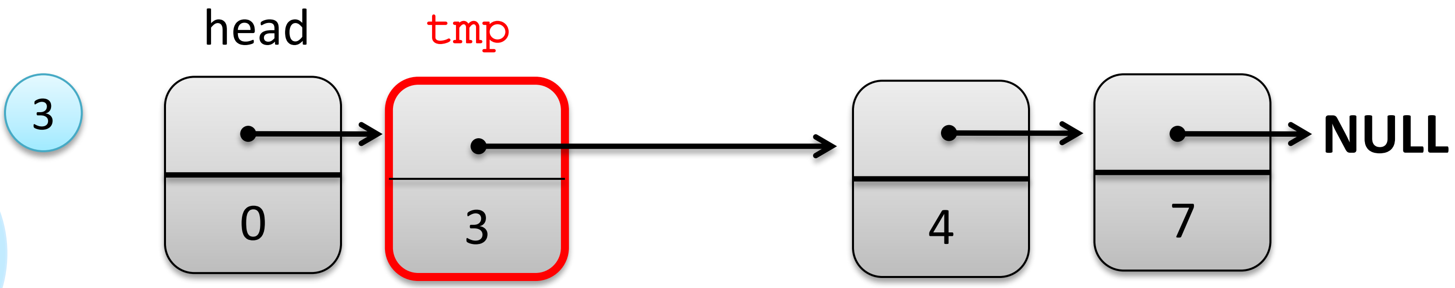
`removeNode( head, 1 );`



Linked Lists - Delete Node

```
int removeNode( node *head, int pos ){
     int i;
 1)  node *tmp = head;
     for(i=0 ; i<pos; i++)
        tmp = tmp->next;
 2)  node* tmp2 = tmp->next;
     tmp->next = tmp->next->next;
 3)  free(tmp2);
     return 0;
}
```

`removeNode( head, 1 );`



Linked Lists
Delete Whole List

```
struct ll_node {
    int value;
    struct ll_node *next;
};
```

```
int destroyList( node **head ){
    node *tmp;
    while( (*head)->next != NULL ){
        tmp = (*head);
        (*head) = (*head)->next;
        free(tmp);
    }
    return 0;
}
```

```
destroyList( &head );
```

Linked Lists
Delete Whole List

```
struct ll_node {
    int value;
    struct ll_node *next;
};
```

```
int destroyList( node **head ){
    node *tmp;
    while( (*head)->next != NULL ){
        tmp = (*head);
        (*head) = (*head)->next;
        free(tmp);
    }
    return 0;
}
```

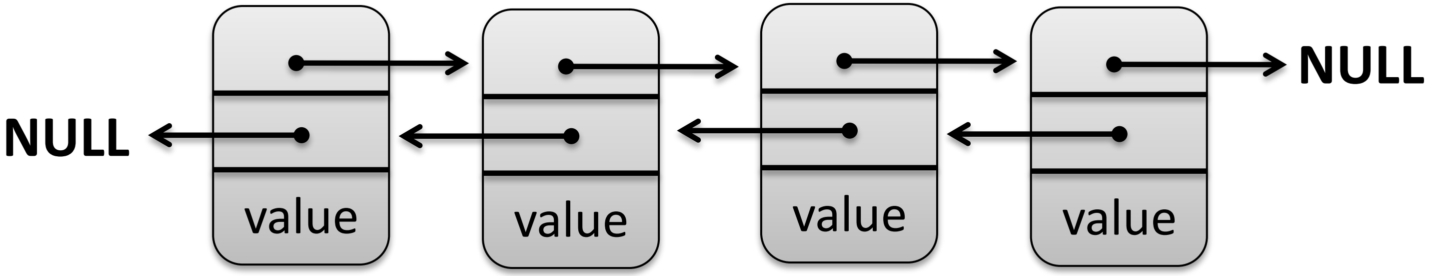
I need to pass head by reference, because I am changing it within the function

```
destroyList( &head );
```

Doubly linked lists

- Pointer to next AND previous node
- Faster backtracking

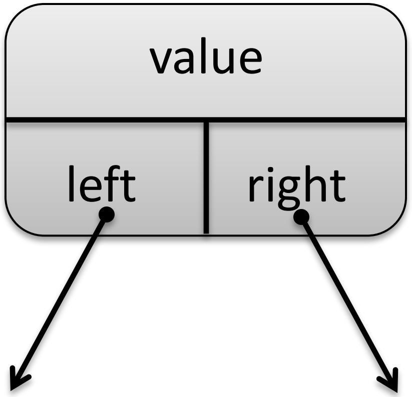
```
struct dll_node {
    int value;
    struct dll_node *prev;
    struct dll_node *next;
};
```



Binary Trees

- Like lists, but each node has a pointer to two elements:
  - Left has a value < current node
  - Right has a value > current node
- First node is called ROOT

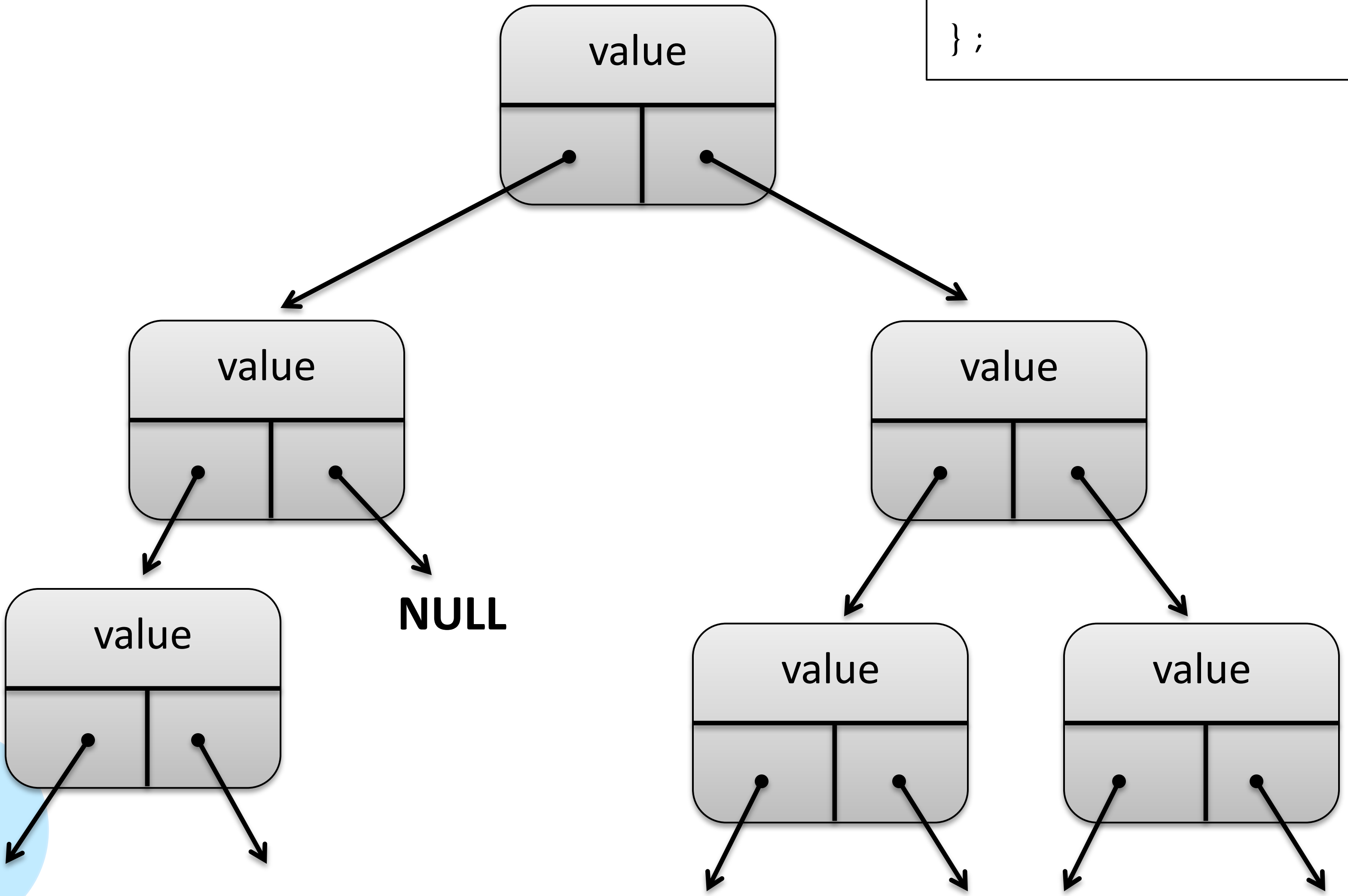
```
struct t_node {
    int value;
    struct t_node *left;
    struct t_node *right;
};
```



Binary Trees

- Left has a value < current node
- Right has a value > current node

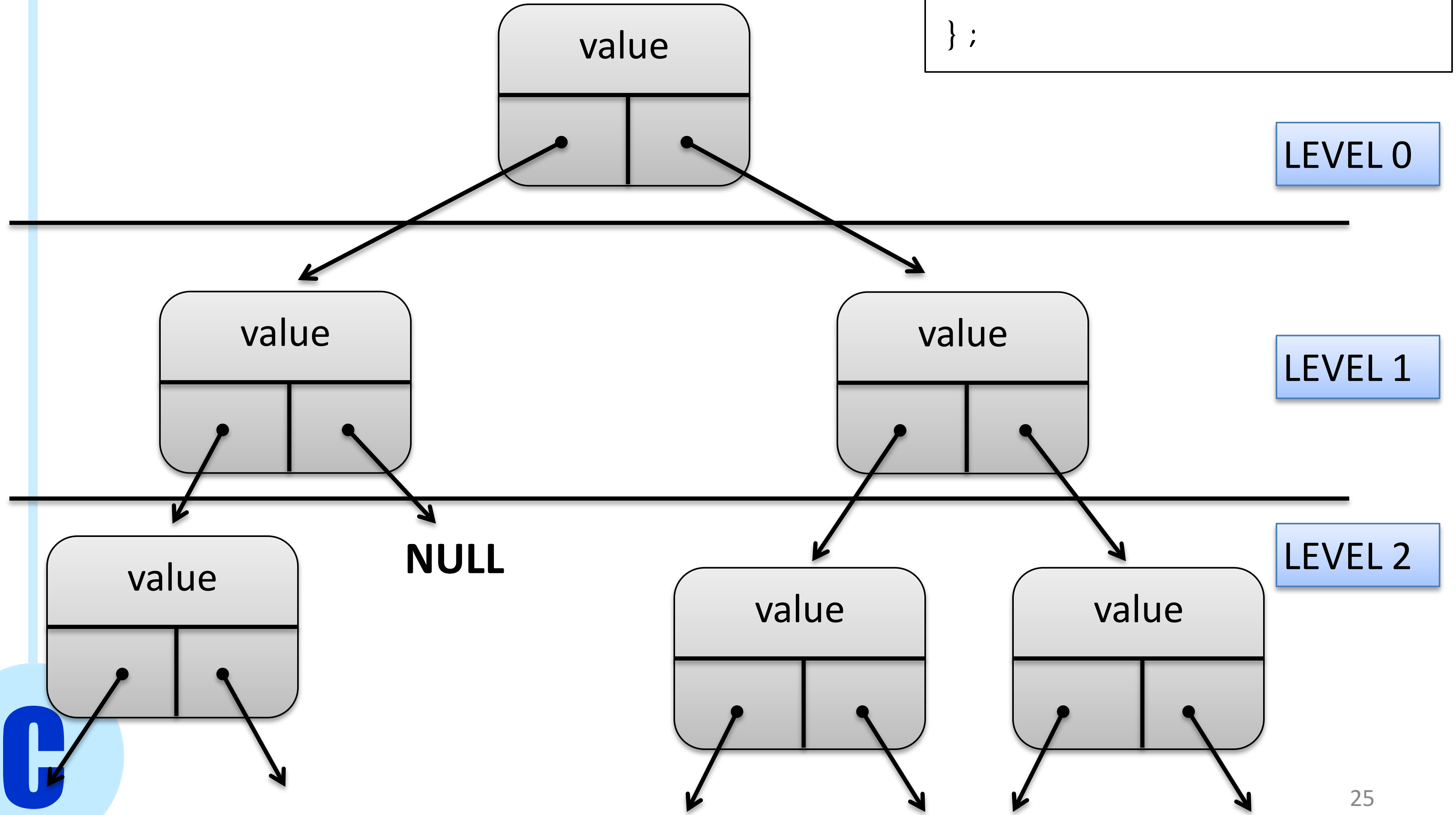
```
struct t_node {
  int value;
  struct t_node *left;
  struct t_node *right;
};
```



Binary Trees

- Left has a value < current node
- Right has a value > current node

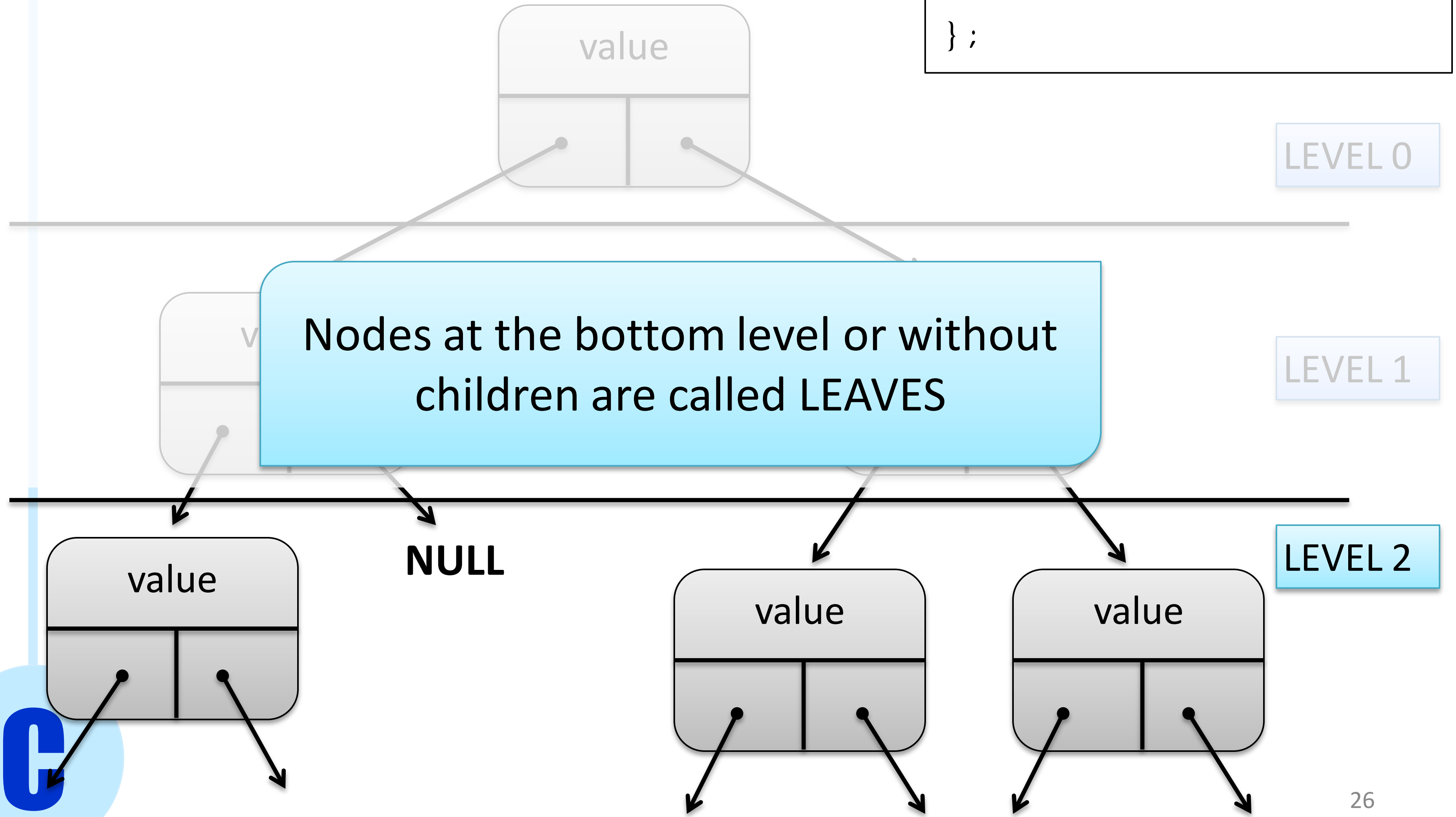
```
struct t_node {
  int value;
  struct t_node *left;
  struct t_node *right;
};
```



Binary Trees

- Left has a value < current node
- Right has a value > current node

```
struct t_node {
  int value;
  struct t_node *left;
  struct t_node *right;
};
```



Nodes at the bottom level or without children are called LEAVES

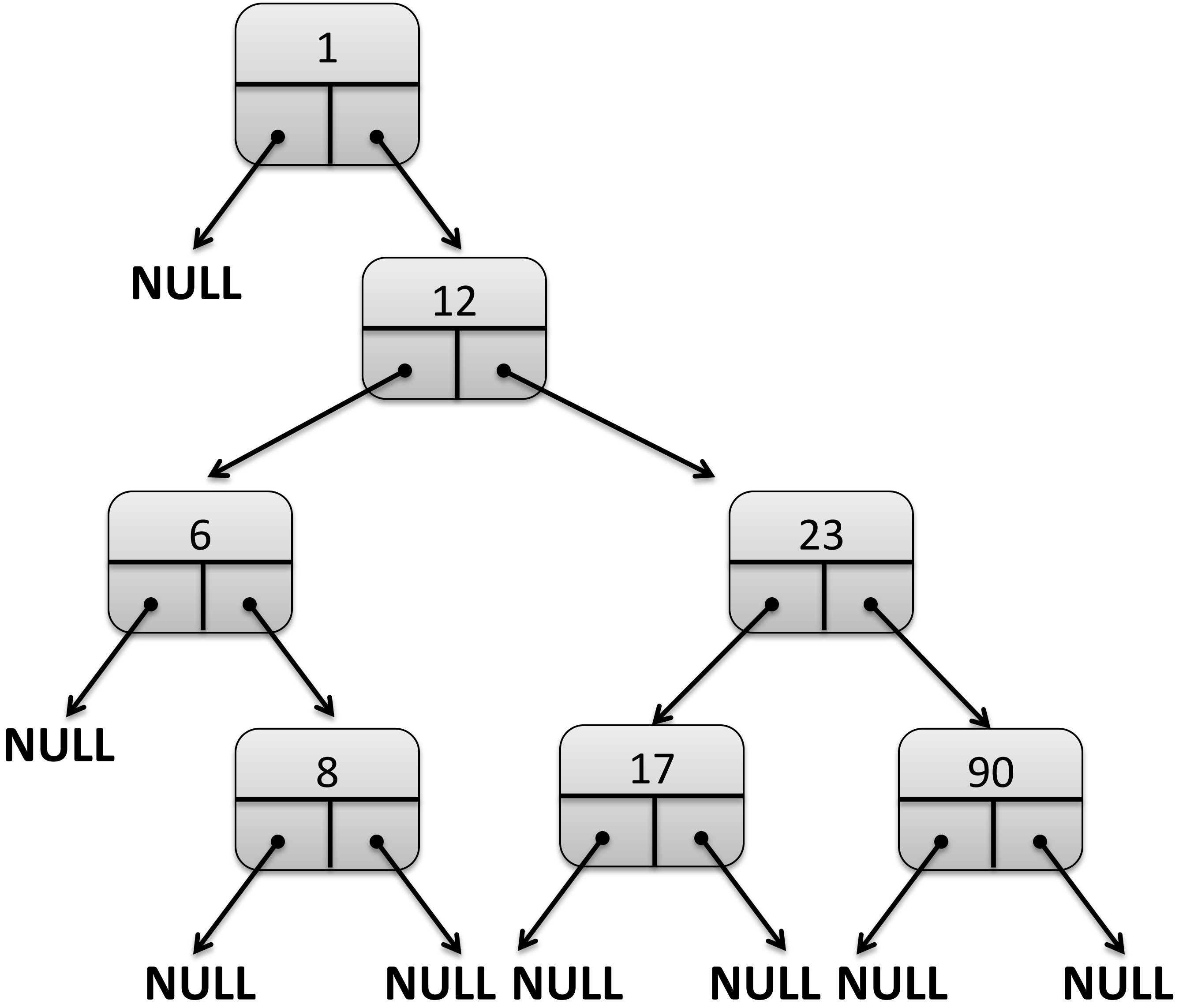
Binary Trees

Inserting number **x into a Binary Tree:**

```
1.  Start at root

2.  if (current node is NULL)
          create new node and set node's value to x
3.  else

          if (x >= current node's value )
               follow right pointer
          else
               follow left pointer

          Go to 1
```

Binary Trees

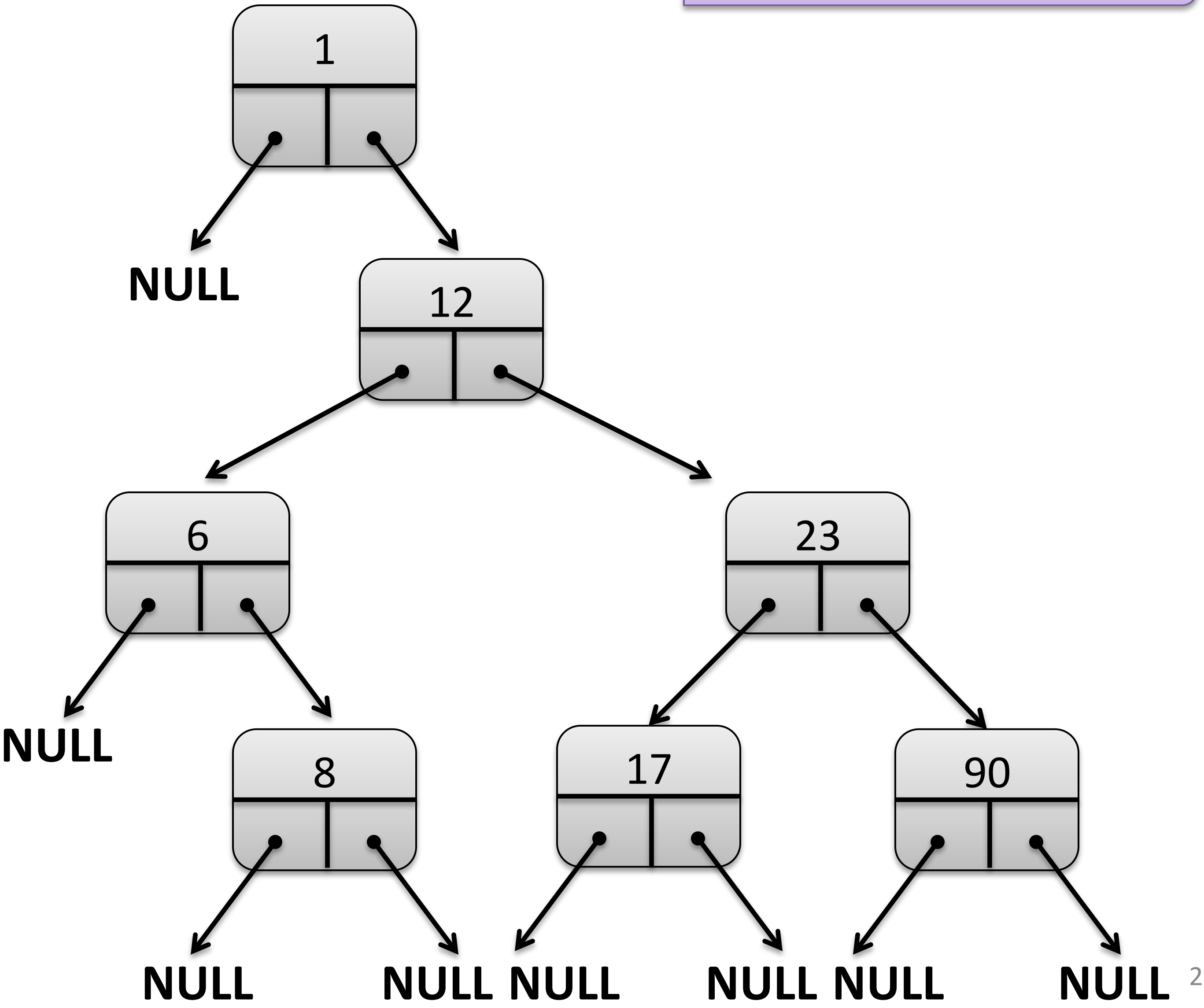
Example: [1 12 6 23 17 90 8]



Binary Trees

Example: [1 12 6 23 17 90 8]

Find all elements < 10

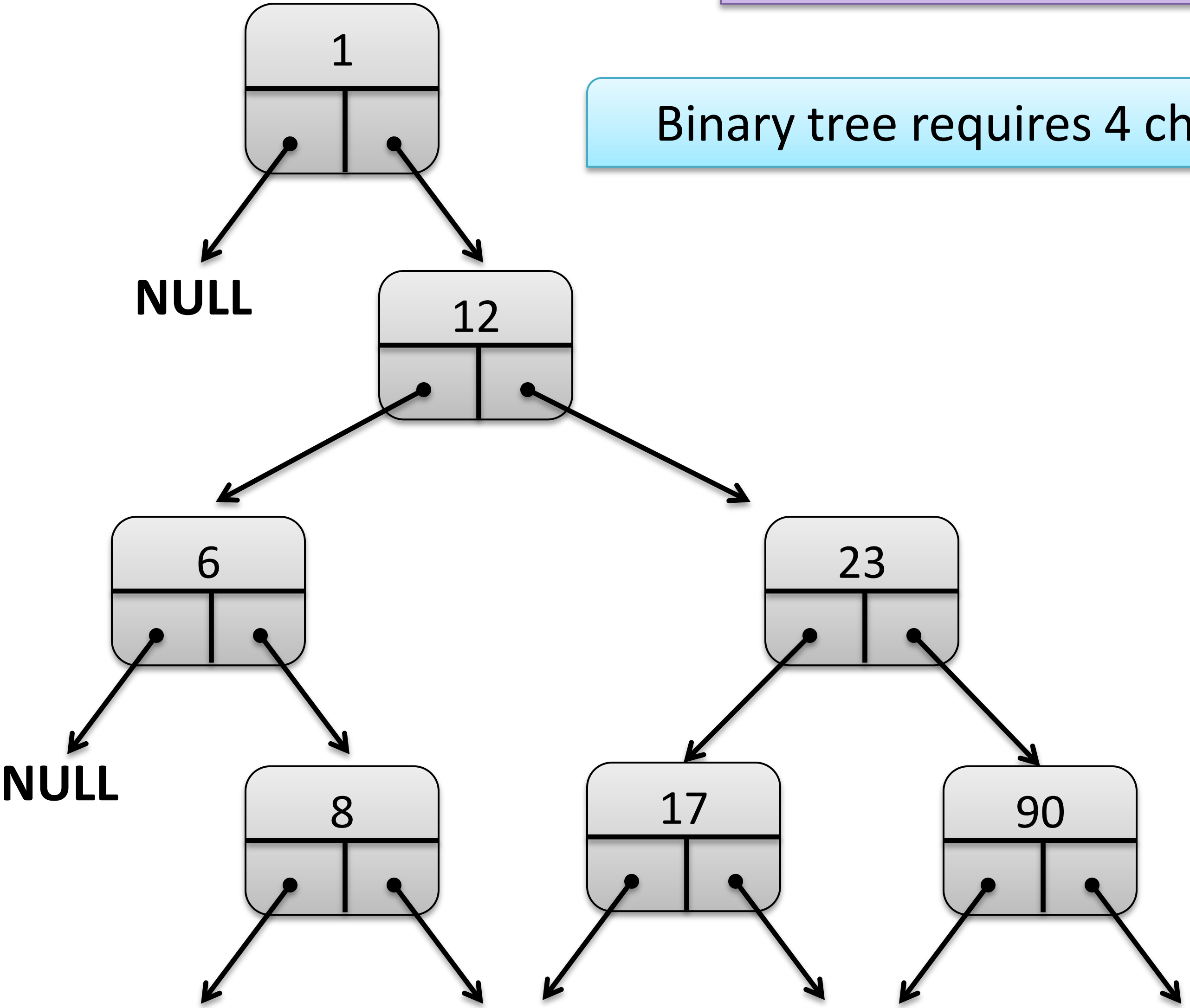


Binary Trees

Example: [1 12 6 23 17 90 8]

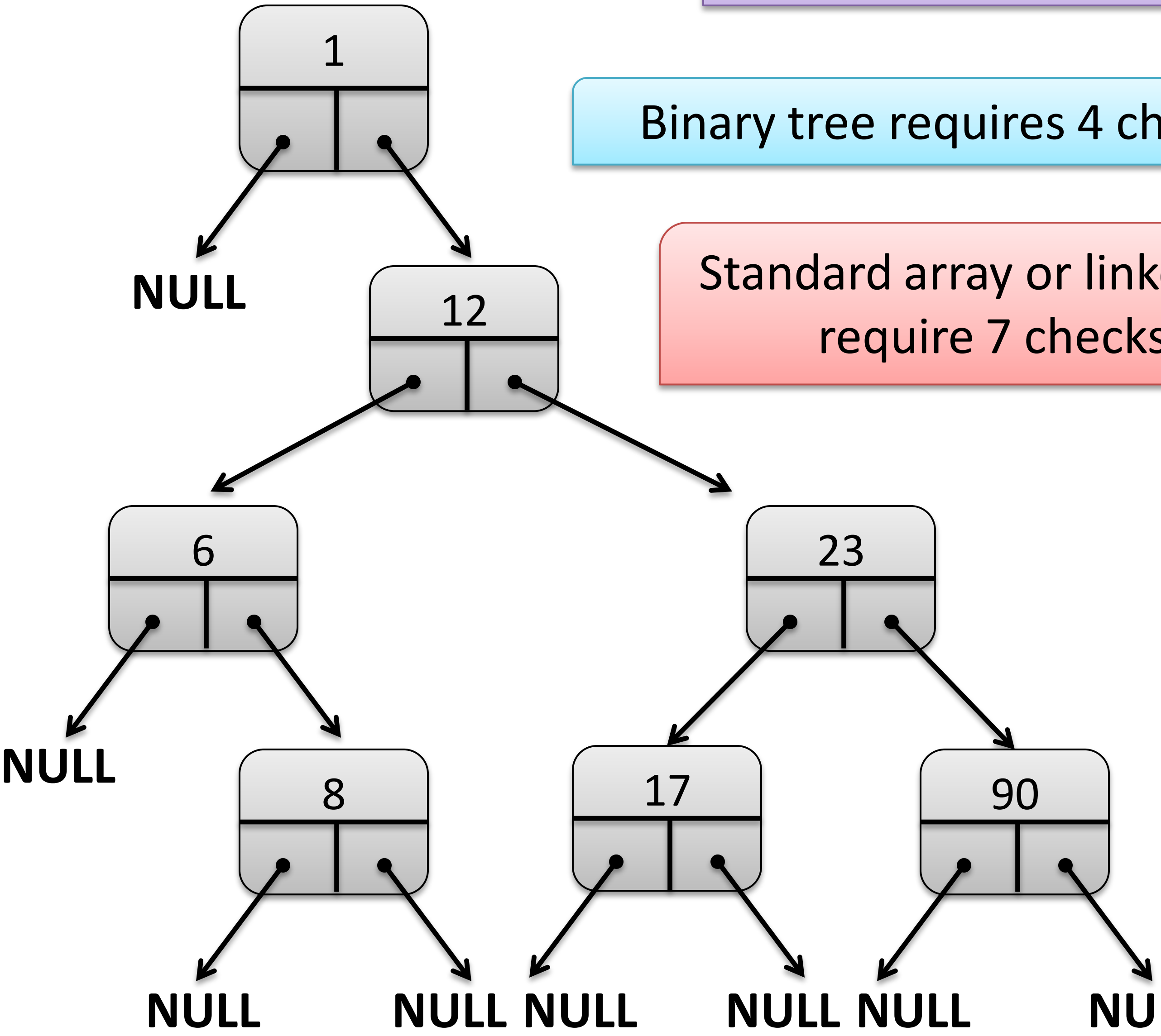
Find all elements < 10

Binary tree requires 4 checks



Binary Trees

Example: [1 12 6 23 17 90 8]



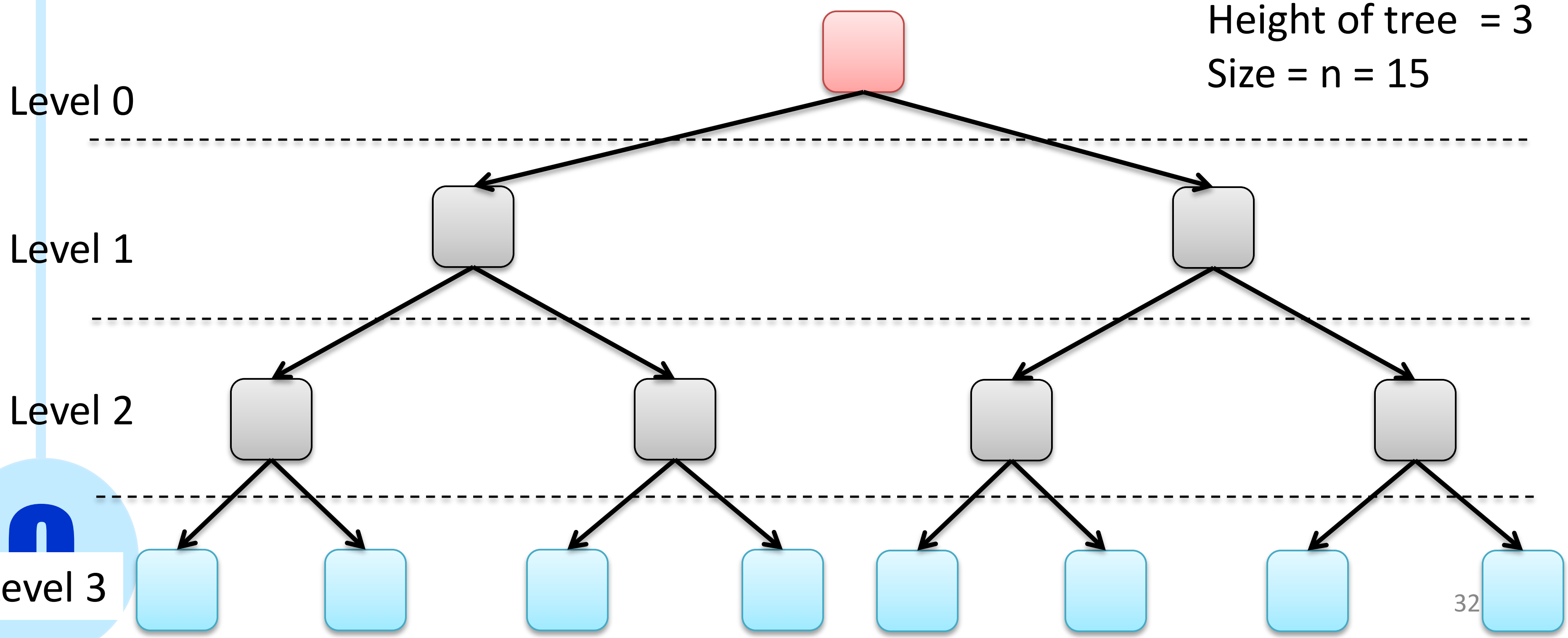
Find all elements < 10

Binary tree requires 4 checks

Standard array or linked list require 7 checks

Trees Definitions

- Root : node with no parents. Leaf : node with no children
- Depth (of a node) : path from root to node
- Level: set of nodes with same depth
- Height or depth (of a tree) : maximum depth
- Size (of a tree) : total number of nodes
- Balanced binary tree : depth of all the leaves differs by at most 1.



Read PCP Chapter 17

COMSW 1003-1

Introduction to Computer Programming in C

Lecture 20

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Announcements

- HW5 out this Wednesday,
  - Due on Wednesday, April 27th before class
- Final on Monday May 9th, from 9am to 12pm, in class
  - Same format as Midterm

Today

- Quick review of linked lists
- Binary Trees
- Complexity Analysis

Introduction to Complexity Analysis

Measuring Algorithms

- In Computer Science, we are interested in finding a function that defines the quantity of some resource consumed by a particular algorithm

- This function is often referred to as a **complexity** of the algorithm

- The resources we usually investigate are
  - running time
  - memory requirements

Measuring Algorithms

- We want to express complexity in the most general way possible

- Running time and space typically depend on input size

For varying input sizes, we can write time and space requirements as functions of ***n*.**

- Algorithms run on different machines

For varying implementation, we use a description independent from constant factors.

Example

Given an array X of 10 elements of type `int`

X	**7**	**1**	**44**	**2**	**34**	**9**	**12**	**7**	**33**	**12**

Complexity analysis

- What is the running time (RT) of an algorithm that sums the elements in the array?
- How much space (SP) in memory is used by that algorithm?

```
int X[10];
int i, sum = X[0];

for(i=1; i<10;i++){
    sum += X[i];
}
```

Example

Given an array X of 10 elements of type `int`

X	**7**	**1**	**44**	**2**	**34**	**9**	**12**	**7**	**33**	**12**

Complexity analysis

- What is the running time (RT) of an algorithm that sums the elements in the array?
- How much space (SP) in memory is used by that algorithm?

```
int X[10];
int i, sum = X[0];

for(i=1; i<10;i++){
    sum += X[i];
}
```

Machine 1

Addition → 2 seconds
`int` → 4 bytes

RT = 9 * 2 = 18
SP = 10*4 + 2*4 = 48

Machine 2

Addition → 3 seconds
`int` → 8 bytes

RT = 9 * 3 = 27
SP = 10*8 + 2*8 = 96

. . .

Machine 2

Addition → 2 seconds
`int` → 8 bytes

RT = 9 * 2 = 18
SP = 10*8 + 2*8 = 96

Example

Given an array of 10 elements of type `int`

X	7	1	44	2	34	9	12	7	33	12

Complexity analysis

- What is the running time (RT) of an algorithm that sums the elements in the array?
- How much space (SP) in memory is used by that algorithm?

```
int X[10];
int i, sum = X[0];

for(i=1; i<10;i++){
     sum += X[i];
```

This is **not general**!
Performance of machines, not of algorithm!
What if array has ***n* elements**?

Machine

Addition → 2 seconds
`int`

Addition → 3 seconds

. . .

chine 2

Addition → 2 seconds
bytes

RT = 9 * 2
SP = 10*4

8
*8 = 96

We want to express complexity of algorithm in terms of

- ***n* : number of elements in array (variable)**
- c : number of seconds to execute addition (constant)
- b : number of bytes to store elements (constant)

Example

Given an array of 10 elements of type `int`

X	**7**	**1**	**44**	**2**	**34**	**9**	**12**	**7**	**33**	**12**

Complexity analysis

- What is the running time (RT) of an algorithm that sums the elements in the array?

- How much space (SP) in memory is used by that algorithm?

```
int X[10];
int i, sum = X[0];

for(i=1; i<10;i++){
    sum += X[i];
}
```

We want to express complexity of algorithm in terms of

— ***n*** **: number of elements in array (variable)**

— c : number of seconds to execute addition (constant)

— b : number of bytes to store elements (constant)

$$\mathbf{RT = c(}\boldsymbol{n}\mathbf{-1)}$$

$$\mathbf{SP = b(}\boldsymbol{n}\mathbf{+2)}$$

Big – O Notation

GOAL: estimate the order of the function **f(*n*)** that represents RT or SP in terms of ***n***

f(n) = O(g(n))

$$f(n) \underset{n\to\infty}{=} O(g(n))$$

$$\Leftrightarrow$$

$\exists C > 0$ and n_0 :

$$|f(n)| \le C|g(n)| \quad \forall n > n_0$$

Big – O Notation

GOAL: estimate the order of the function **f(*n*)** that represents RT or SP in terms of ***n***

$$\boxed{f(n) = O(\, g(n) \,)}$$

$$f(n) \underset{n \to \infty}{=} O(g(n))$$

f(n) equals oh of g(n) as *n* tends to infinity

$$\Leftrightarrow$$

if and only if

$$\exists C > 0 \text{ and } n_0 :$$

there exists a positive constant **C** and a value n_0 such that

$$|f(n)| \le C|g(n)| \quad \forall n > n_0$$

for all *n* greater than n_0, the absolute value of f(n) is smaller than **C** times the absolute value of g(n)

Big – O Notation

GOAL: estimate the order of the function **f(*n*)** that represents RT or SP in terms of ***n***

f(n) =O(g(n))

$$f(n) \underset{n\to\infty}{=} O(g(n))$$

$$\Leftrightarrow$$

$\exists C > 0$ and n_0 :

$$\boxed{|f(n)| \le C|g(n)|} \quad \forall n > n_0$$

In other words, big-O means less than some constant scaling
When analyzing complexity with big-O notation, we always consider the WORST CASE SCENARIO

Big-O notation: Examples

- $f(n) = 3n^4 + 7n^2 - 5n + 8$

$$\begin{aligned} \left|3n^4 + 7n^2 - 5n + 8\right| &\le 3n^4 + 7n^2 + \left|5n\right| + 8 \\ &\le 3n^4 + 7n^4 + 5n^4 + 8n^4 \\ &\le 23n^4 \end{aligned}$$

$|f(n)| \le C|g(n)|$

$\boxed{f(n) = O(n^4)}$

- What is the running time (RT) of an algorithm that sums *n* elements in an array?

$\boxed{C(n\text{-}1) = O(n\text{-}1) = O(n)}$

Big – O : common cases

O(1) - constant time

- The algorithm requires the same fixed number of steps regardless of the size of the task
- Example: insert an element in front of a linked list

```
int addNodeFront( int val, node *head ){
   1) node *newNode = malloc(sizeof(node));
   2) newNode->value = val;
   3) newNode->next = head->next;
   4) head->next = newNode;
}
```

No matter how long the list is, this operation always requires 4 steps
O(4) = O(1)

Big – O : common cases

O(1) - constant time

if c<<n
O(c) = O(1)

- The algorithm requires the same fixed number of steps regardless of the size of the task
- Example: insert an element in front of a linked list

```
int addNodeFront( int val, node *head ){
   1) node *newNode = malloc(sizeof(node));
   2) newNode->value = val;
   3) newNode->next = head->next;
   4) head->next = newNode;
}
```

No matter how long the list is, this operation always requires 4 steps
RT = O(4) = O(1)

Big – O : common cases

O(n) - linear time

- The algorithm requires a number of steps proportional to the size of the task
- Examples:
  - Travers a linked list or an array with ***n*** elements;
  - Find the maximum and minimum element in a list or array

```
for(i=0 ; i < n; i++){
        if(arr[i] < minVal)
                minVal = arr[i];
        if(arr[i] > maxVal)
                maxVal = arr[i];
}
```

RT = O(2n) = O(n)

SP = O(n+2) = O(n)

Big – O : common cases

$O(n^2)$ - quadratic time

- The number of operations is proportional to the size of the task squared.

- Example: Finding duplicates in an unsorted list of *n* elements

```
for(i=0 ; i < n; i++){
  for(j=0 ; j < n; j++){
    if( (i!=j) && arr[i] == arr[j] )
      dup[i][j] = 1;
  }
}
```

$$RT = O(4n^2+n) = O(n^2)$$

increment `i` → n times

increment `j` → n^2 times

check `i!=j` → n^2 times

check `arr[i]==arr[j]` → n^2 times

`dup[i][j]=1` → $(n-1)*(n-1)$ times

Big – O : common cases

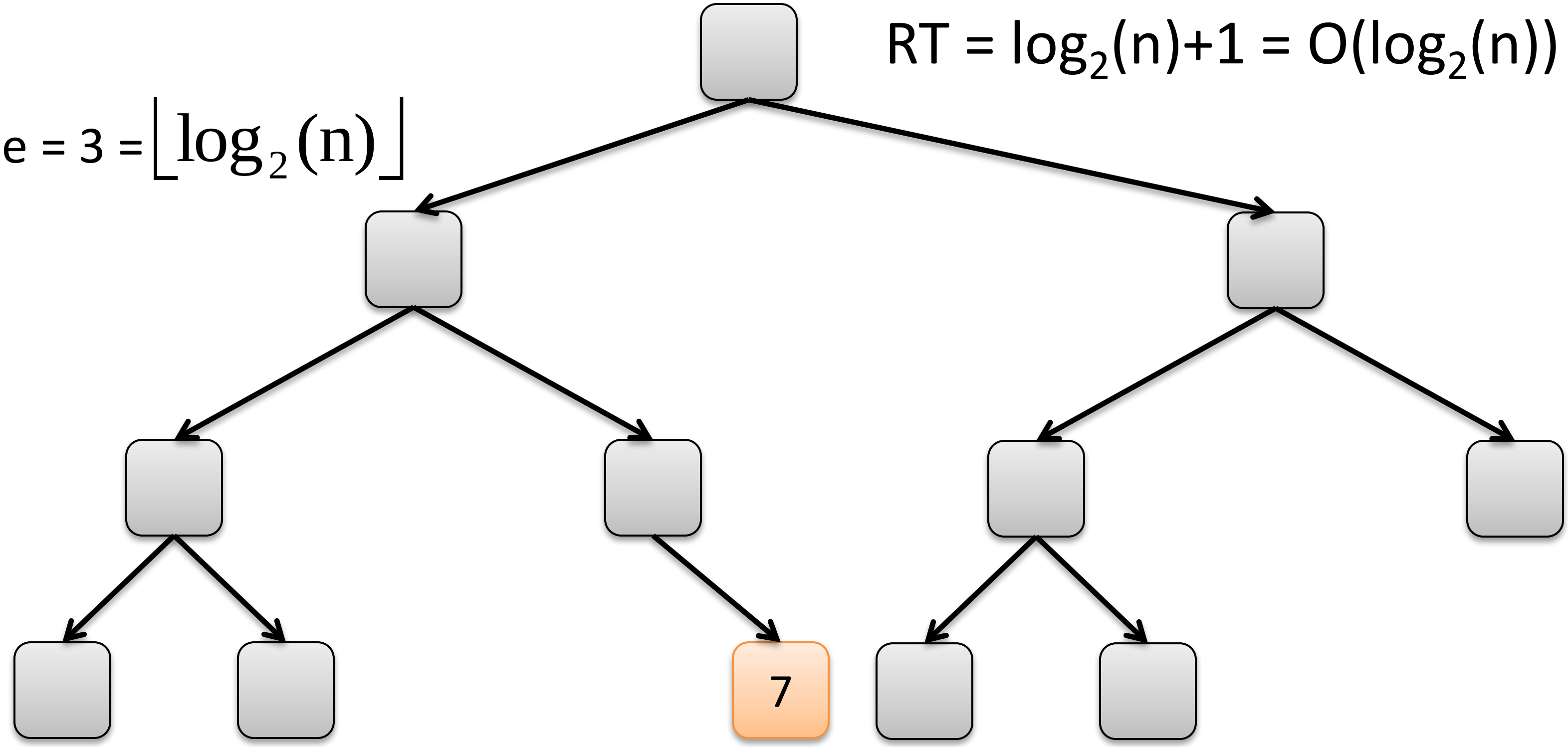
O(log(n)) - logarithmic time

- Example: Find operation in a balanced binary tree with ***n*** nodes

RT = $\log_2(n)+1 = O(\log_2(n))$

n = 15

height of tree = 3 = $\lfloor \log_2(n) \rfloor$



Big – O : common cases

O(n log(n)) – “n log(n)” time

- Examples: sorting algorithms (will see in next class)
  - quicksort
  - mergesort

Big – O : common cases

O(a^n) – exponential time

a > 1

- Example: Recursive Fibonacci implementation

```
int fib( int n ) {

 switch(n) {
  case 0:
    return(0);
  case 1:
    return(1);
  default:
    return(fib(n-1) + fib(n-2));
}
```

How many times is fib() called?

Cost of fib() without return statement = 2 = O(1)

RT(n) = RT(n-1) + RT(n-2) + O(1)

RT = O(a^n)

$$a^n = a^{n-1} + a^{n-2}$$

$$a^2 = a + 1$$

$$a = \frac{1+\sqrt{5}}{2} \approx 1.6$$

Big–O : Relationship among common cases

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(a^n)$

Example : big-O when a function is the *sum of several statements*

```
int i=0;
for(i=0 ; i < n; i++){
    for(j=0 ; j < n; j++){
        if( (i!=j) && arr[i] == arr[j])
            dup[i][j] = 1;
    }
}
```

$RT = O(4n^2+n) = O(n^2)$

↓

increment `i`
increment `j`
check `i!=j`
check `arr[i]==arr[j]`
`dup[i][j] = 1`

Longest operation dominates (worst case)

COMSW 1003-1

Introduction to Computer Programming in C

Lecture 21

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Big–O : Relationship among common cases

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(a^n)$

Example : big-O when a function is the *sum of several statements*

```
int i=0;
for(i=0 ; i < n; i++){
   for(j=0 ; j < n; j++){
       if( (i!=j) && arr[i] == arr[j])
           dup[i][j] = 1;
   }
}
```

RT = $O(4n^2+n) = O(n^2)$

increment `i`

increment `j`

check `i!=j`

check `arr[i]==arr[j]`

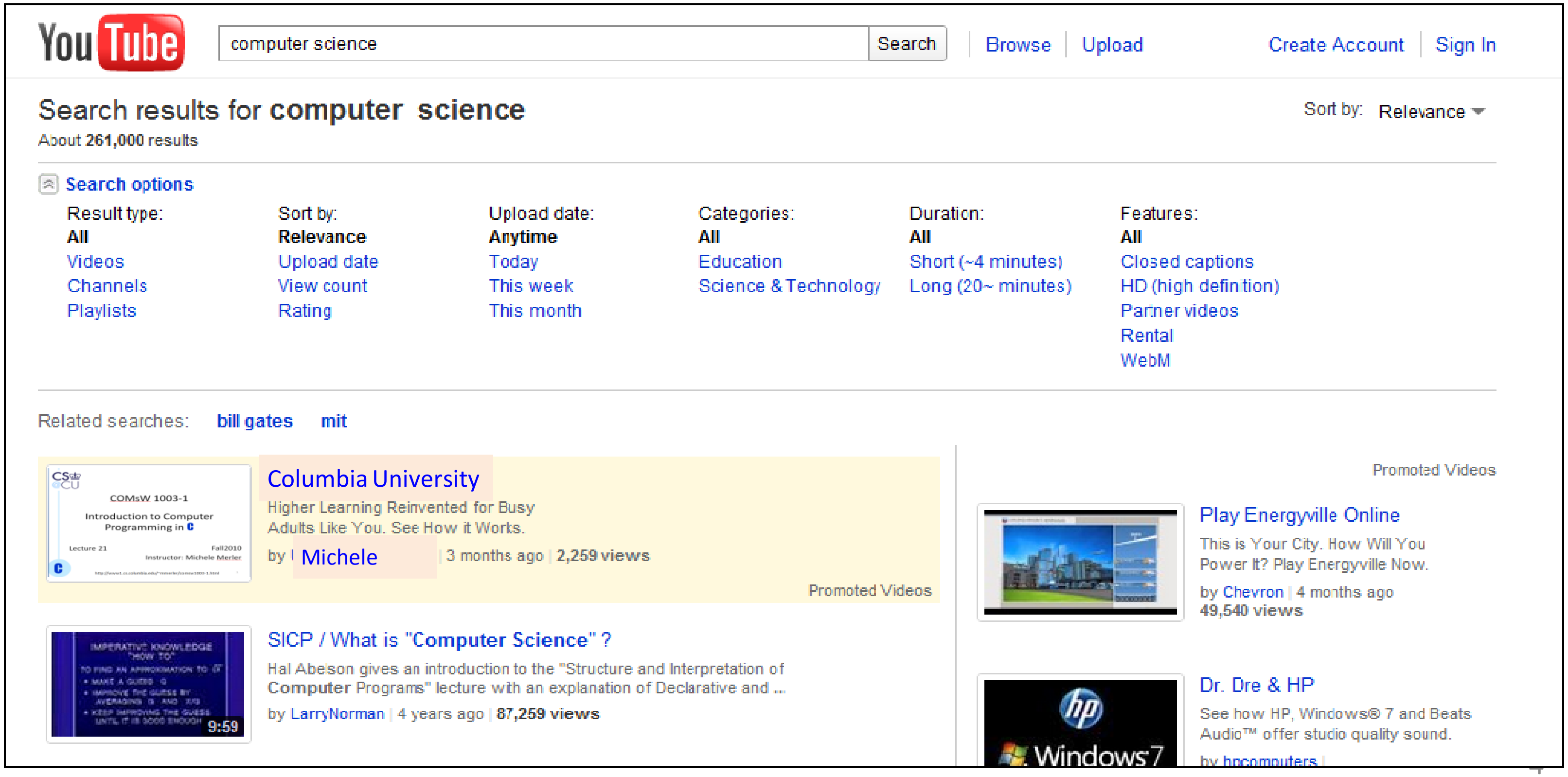
`dup[i][j] = 1`

Longest operation dominates (worst case)

Sorting

Sorting

- Given a set of N elements, put them in order according to some **criteria** (alphabetical, relevance, date, smallest to largest, etc.)
- One of the most studied problems in Computer Science
- Everybody uses it every day



Search results for **computer science**

About **261,000** results

Sort by: Relevance

Search options

Result type:	Sort by:	Upload date:	Categories:	Duration:	Features:
All	**Relevance**	**Anytime**	**All**	**All**	**All**
Videos	Upload date	Today	Education	Short (~4 minutes)	Closed captions
Channels	View count	This week	Science & Technology	Long (20~ minutes)	HD (high defin tion)
Playlists	Rating	This month			Parner videos
					Rental
					WebM

Related searches: bill gates mit

Columbia University

Higher Learning Reinvented for Busy Adults Like You. See How it Works.

by Michele | 3 months ago | 2,259 views

Promoted Videos

SICP / What is "Computer Science" ?

Hal Abeson gives an introduction to the "Structure and Interpretation of Computer Programs" lecture with an explanation of Declarative and ...

by LarryNorman | 4 years ago | **87,259 views**

Promoted Videos

Play Energyville Online

This is Your City. How Will You Power It? Play Energyville Now.

by Chevron | 4 months ago

49,540 views

Dr. Dre & HP

See how HP, Windows® 7 and Beats Audio™ offer studio quality sound.

by hpcomputers |

Sorting

- Given a set of N elements, put them in order according to some criteria

- Compare pairs of elements

- Many algorithms, some of the most famous are:
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Merge sort
  - Counting sort

- In following examples, we'll see smallest to biggest sorting

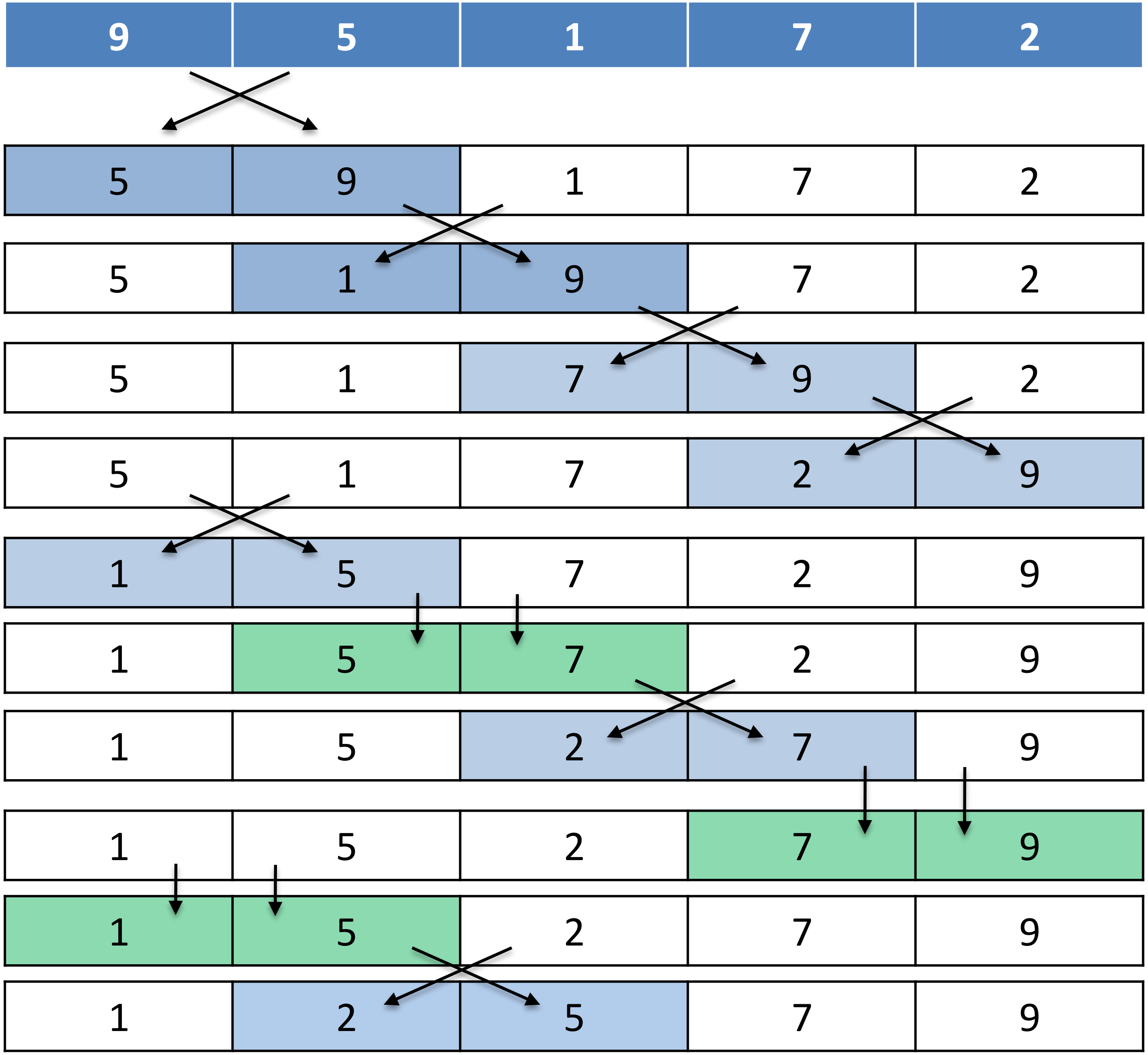
Bubble Sort

1. Start with the first two elements
2. If first element > second element
   - Swap
3. Iterate for all following pairs
4. Repeat steps 1 to 3 until no swaps are necessary

Complexity = $O(n^2)$

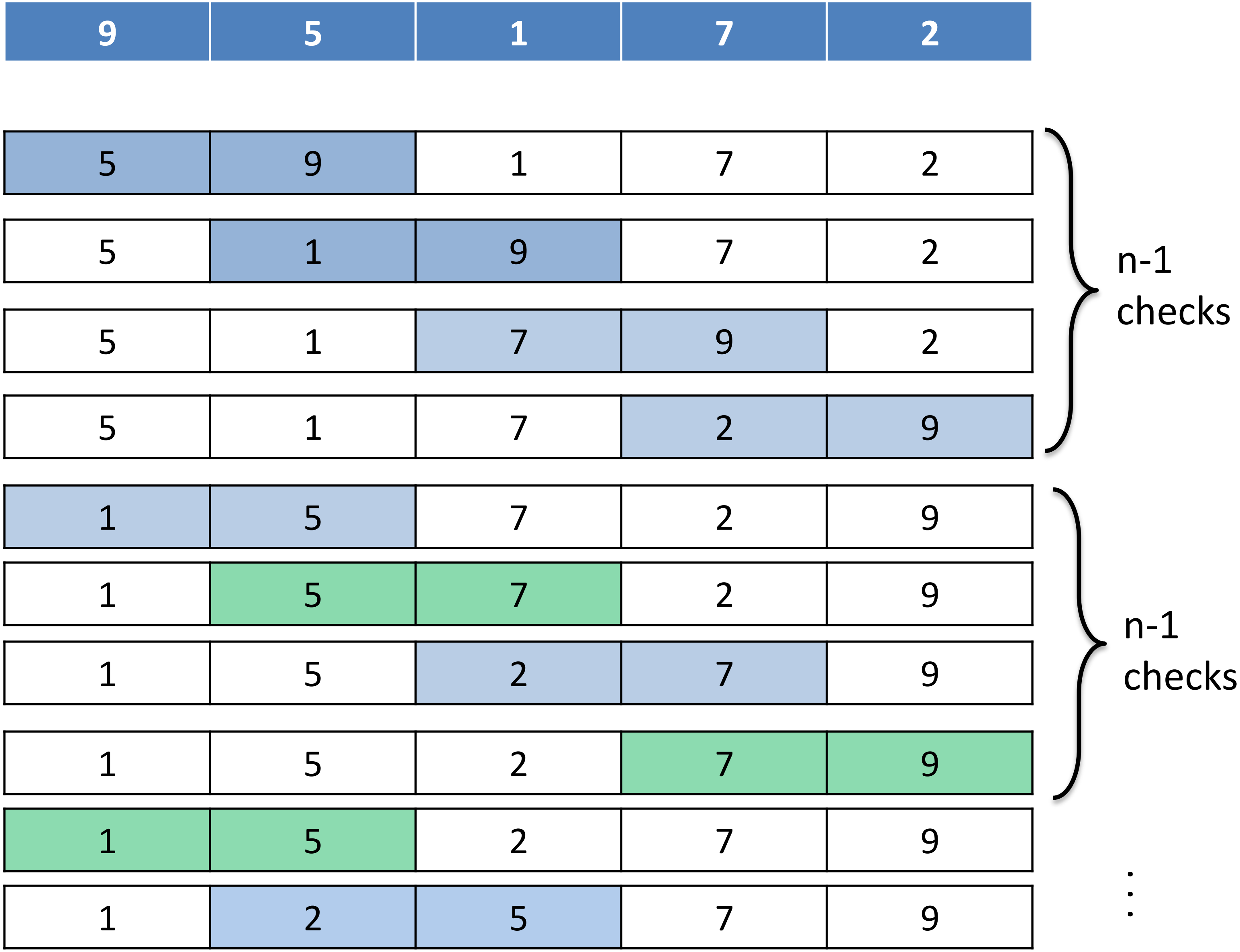
Count number of comparisons and swaps

Bubble Sort



9	5	1	7	2
5	9	1	7	2
5	1	9	7	2
5	1	7	9	2
5	1	7	2	9
1	5	7	2	9
1	5	7	2	9
1	5	2	7	9
1	5	2	7	9
1	5	2	7	9
1	2	5	7	9

Bubble Sort



9	5	1	7	2
5	9	1	7	2
5	1	9	7	2
5	1	7	9	2
5	1	7	2	9
1	5	7	2	9
1	5	7	2	9
1	5	2	7	9
1	5	2	7	9
1	5	2	7	9
1	2	5	7	9

n-1
checks

n-1
checks

⋮

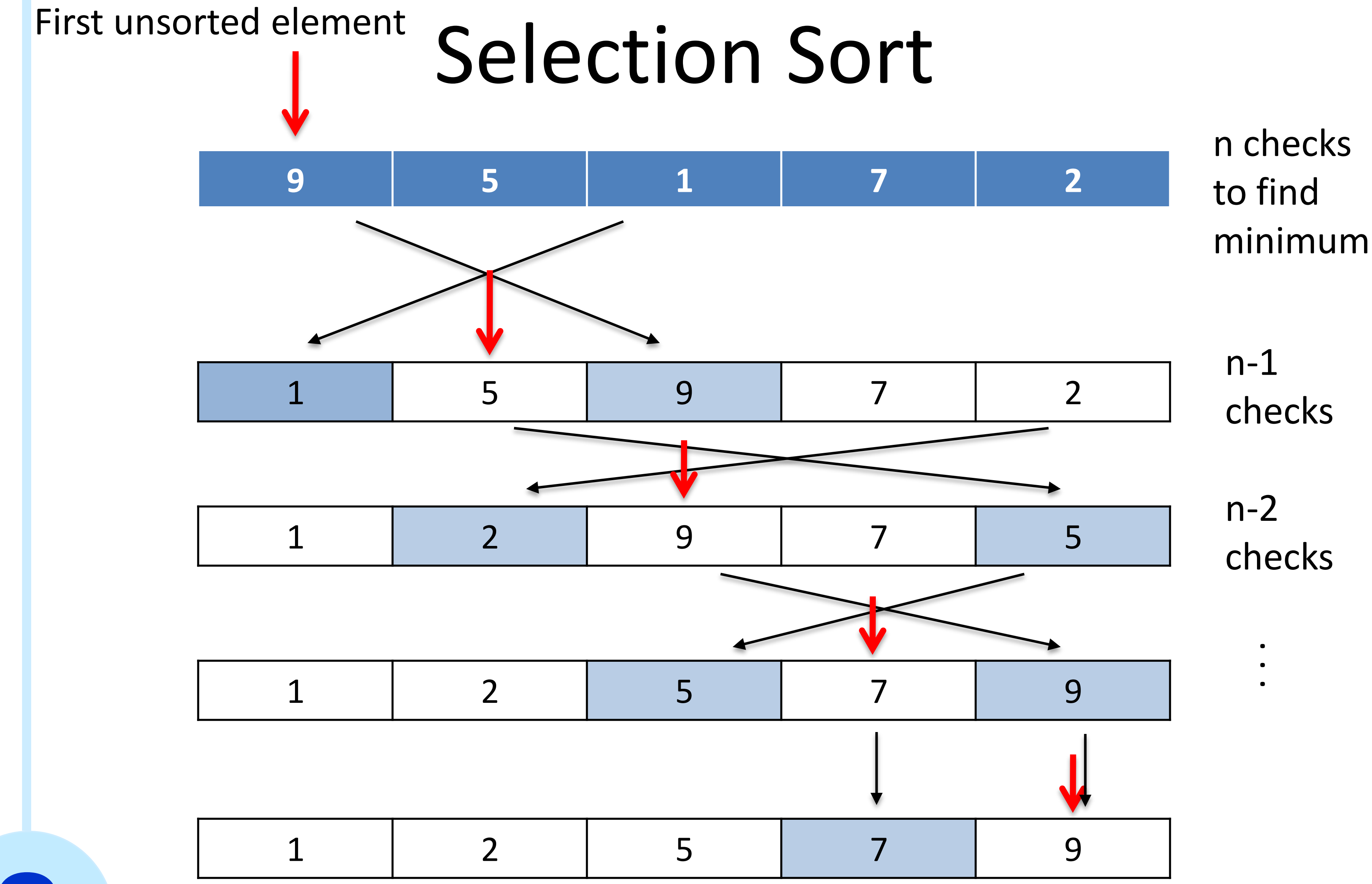
Selection Sort

- Smarter algorithm, but same complexity (worst case)

1. Find smallest unsorted element
2. Swap with first unsorted element
3. Repeat steps 1 and 2 until no more unsorted elements

Complexity = $O(n^2)$

Selection Sort



First unsorted element

9	5	1	7	2

n checks to find minimum

1	5	9	7	2

n-1 checks

1	2	9	7	5

n-2 checks

1	2	5	7	9

⋮

1	2	5	7	9

Insertion Sort

- Main idea: keep 2 separate sets (one sorted, one unsorted), and move elements from unsorted to sorted set one at a time
- Better performance in case many elements are already sorted, quadratic in worst case

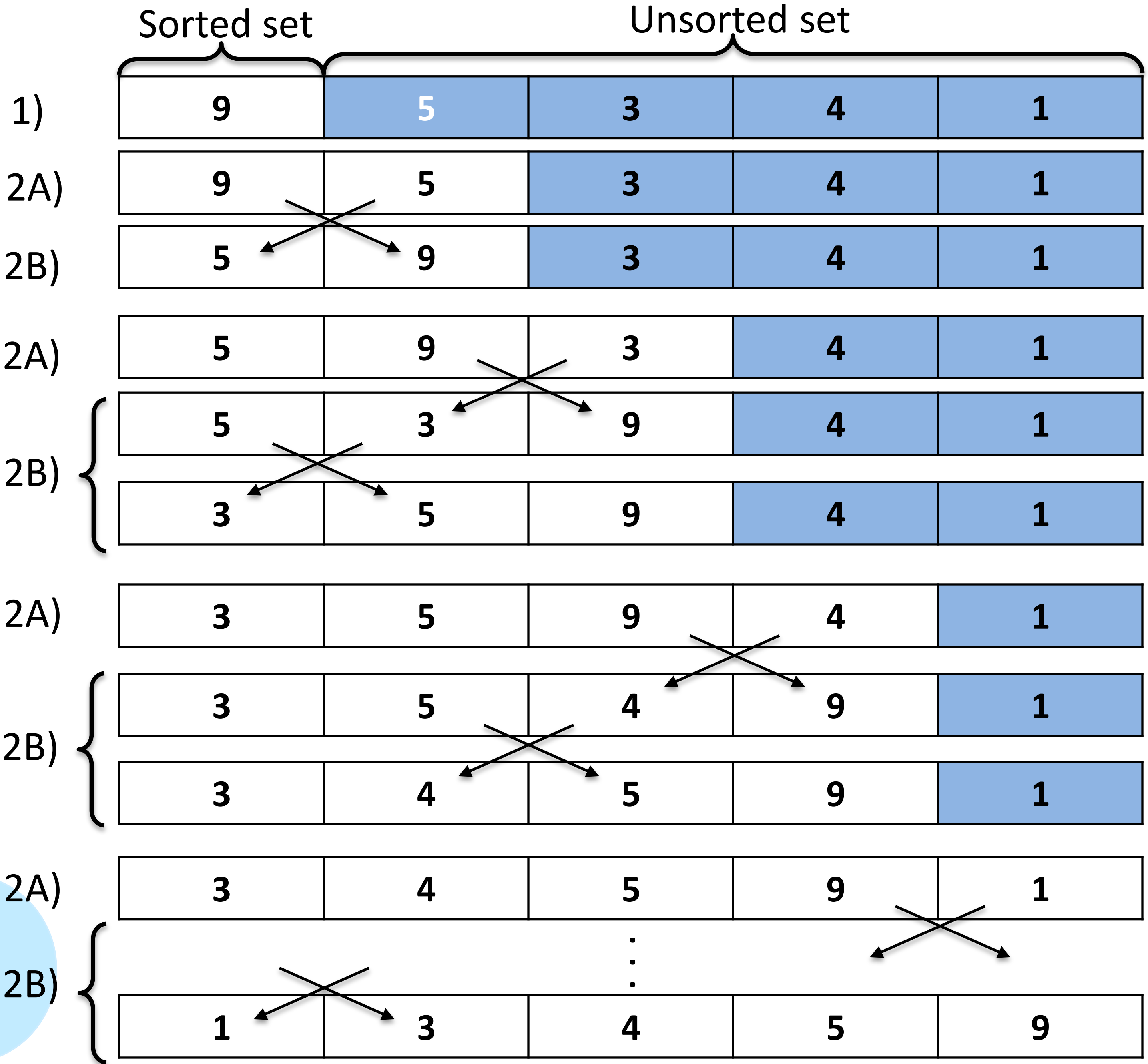
1) Initialize 2 sets
  - One set of sorted elements (contains only first element in the array)
  - One set of unsorted elements (all the other elements in the array)

2) A) Take first element in unsorted set and
   B) Insert it into sorted set at proper position

3) Repeat steps 2A) and 2B) until unsorted set is empty

Complexity = O(n^2)

Insertion sort



1)	9	5	3	4	1
2A)	9	5	3	4	1
2B)	5	9	3	4	1
2A)	5	9	3	4	1
2B)	5	3	9	4	1
	3	5	9	4	1
2A)	3	5	9	4	1
2B)	3	5	4	9	1
	3	4	5	9	1
2A)	3	4	5	9	1
2B)			⋮		
	1	3	4	5	9

1 check
1 swap

2 checks
2 swaps

n-2 checks
n-3 swaps

n-1 checks
n-1 swaps

Merge Sort

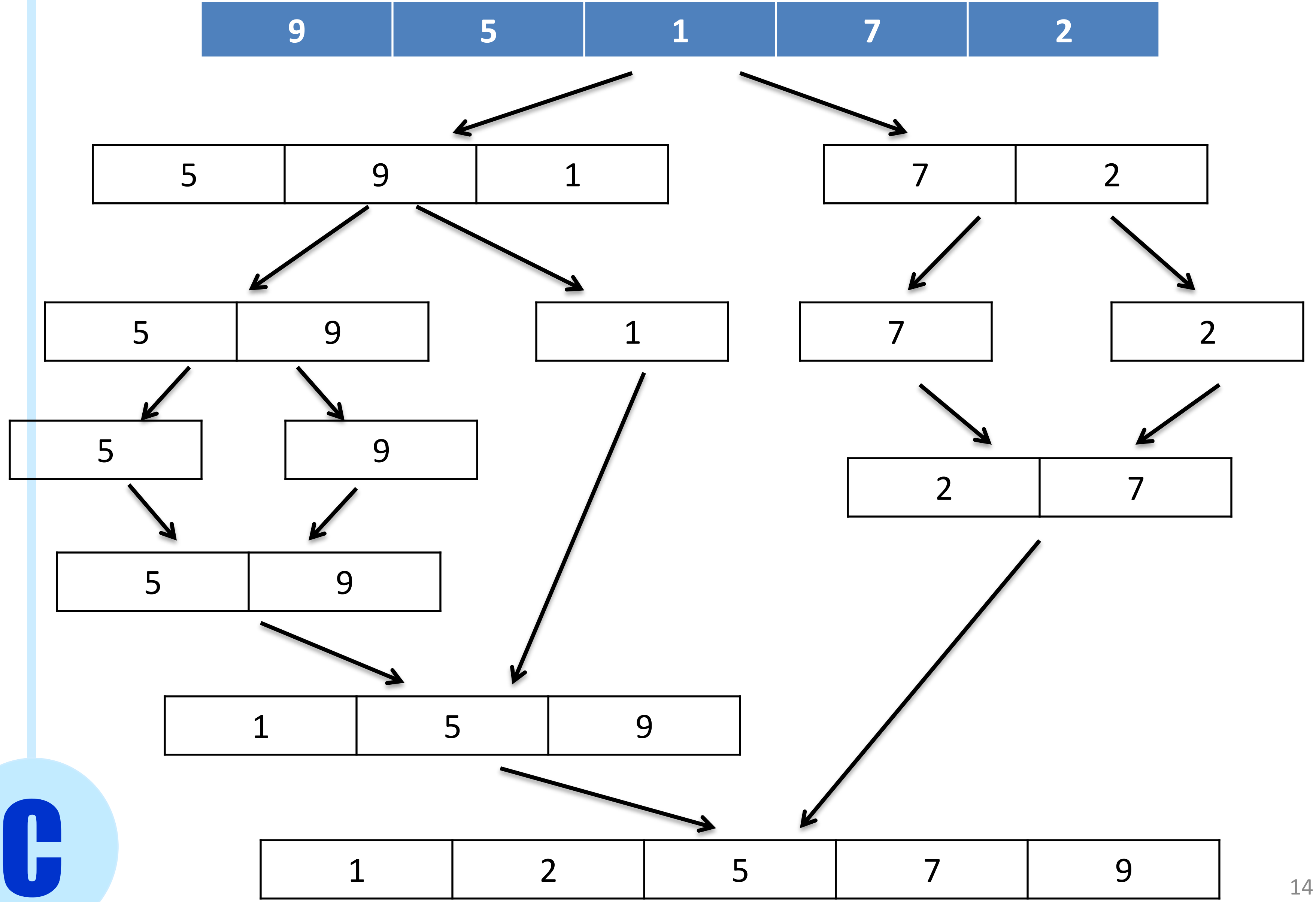
- One of the fastest algorithms, divide and conquer principle
- Uses recursion
- Sorting small sets is faster than sorting large sets
- Merging 2 sets into a sorted union is faster if the sets are already sorted

1. If set H has 1 element, stop
2. else
   - Split set into 2 halves H1 and H2 of (approximately) same size
   - Sort H1 and H2 with merge sort
   - Merge the sorted H1 and H2 into a sorted set

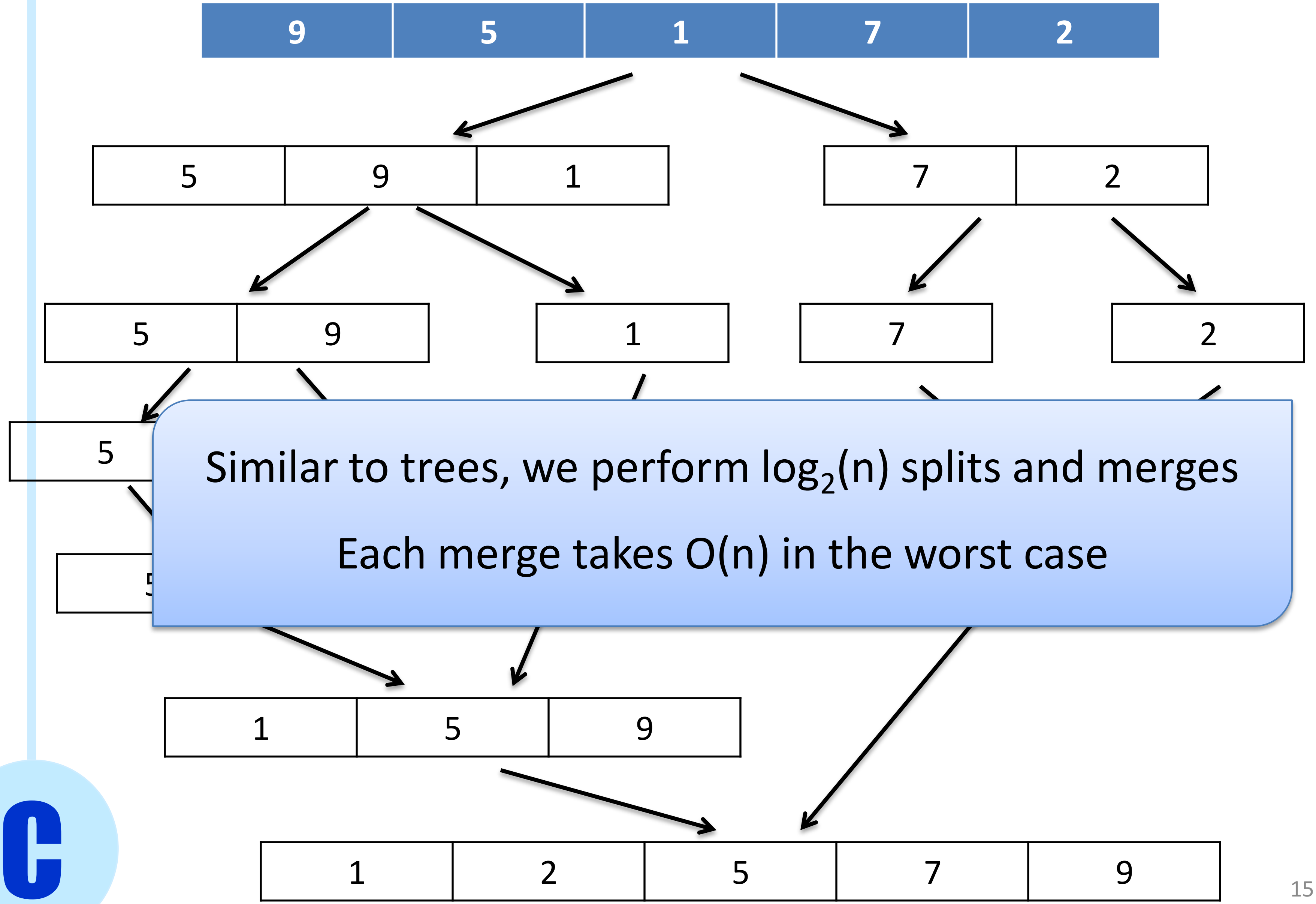
recursion

Complexity = O(n log(n))

Merge Sort



Merge Sort



Similar to trees, we perform $\log_2(n)$ splits and merges

Each merge takes O(n) in the worst case

Merge Sort

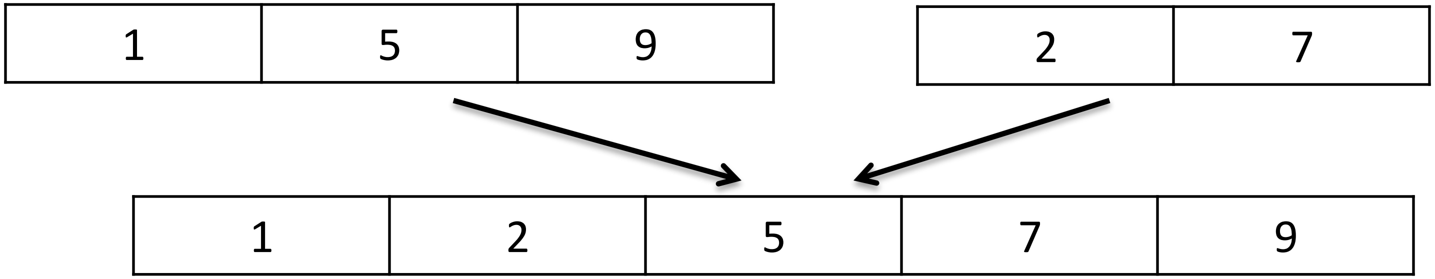
Similar to trees, we perform $\log_2(n)$ splits and merges

Each merge takes O(n) in the worst case

Merge routine:

Given H1 and H2 of size n1 and n2 respectively, create H of length n = n1 + n2

```
int c1=0, c2=0;
for (i=0; i<n; i++){
    if( (c1<n1) && ((H1[c1] < H2[c2]) || (c2==n2)) ){
            H[i] = H1[c1];
            c1++;
    }
    else{
       H[i] = H2[c2];
       c2++;
    }
}
```



1 | 5 | 9

2 | 7

1 | 2 | 5 | 7 | 9

Counting sort

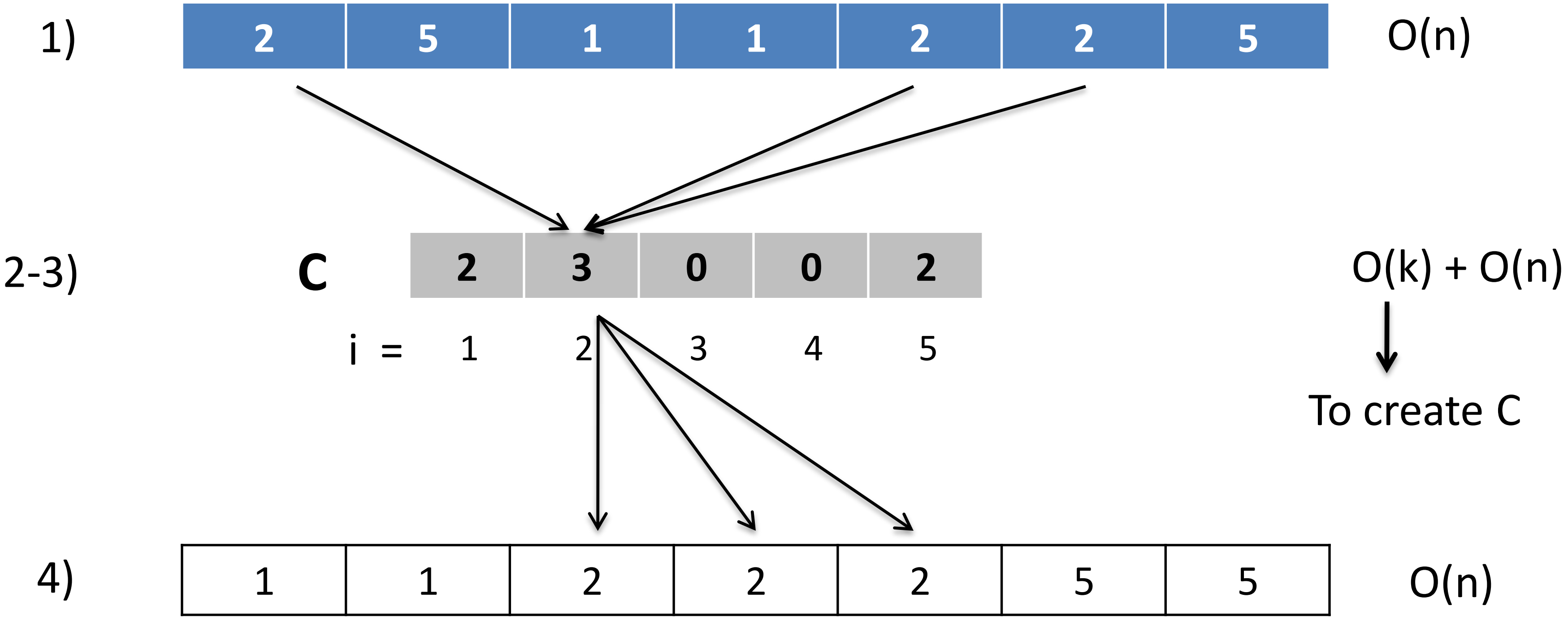
- Intuition: exploit range *k* of values in set
- Efficient if *k* is not much larger than *n*

1. Find biggest and smallest values in the set (k = maxVal – minVal+1)
2. Create an array C of k elements
3. Count occurrences *C(i)* of each value *i* in the set
4. Fill ordered set by inserting *C(i)* elements of value *i*, for each value in range *k*

Complexity = O(n + k)

Counting sort

<u>Example:</u> range of values in set is [1, 5], k = 5



1)

2	**5**	**1**	**1**	**2**	**2**	**5**

O(n)

2-3) **C**

	2	**3**	**0**	**0**	**2**
i =	1	2	3	4	5

O(k) + O(n)

↓

To create C

4)

1	1	2	2	2	5	5

O(n)

Homework 4 Solution