LECTURE 4: C PROGRAMMING

Computer Systems and Networks

Dr. Pallipuram
(vpallipuramkrishnamani@pacific.edu)

Deadlines

Lab 2 September 12th, 2019

Lab 3 September 19th, 2019

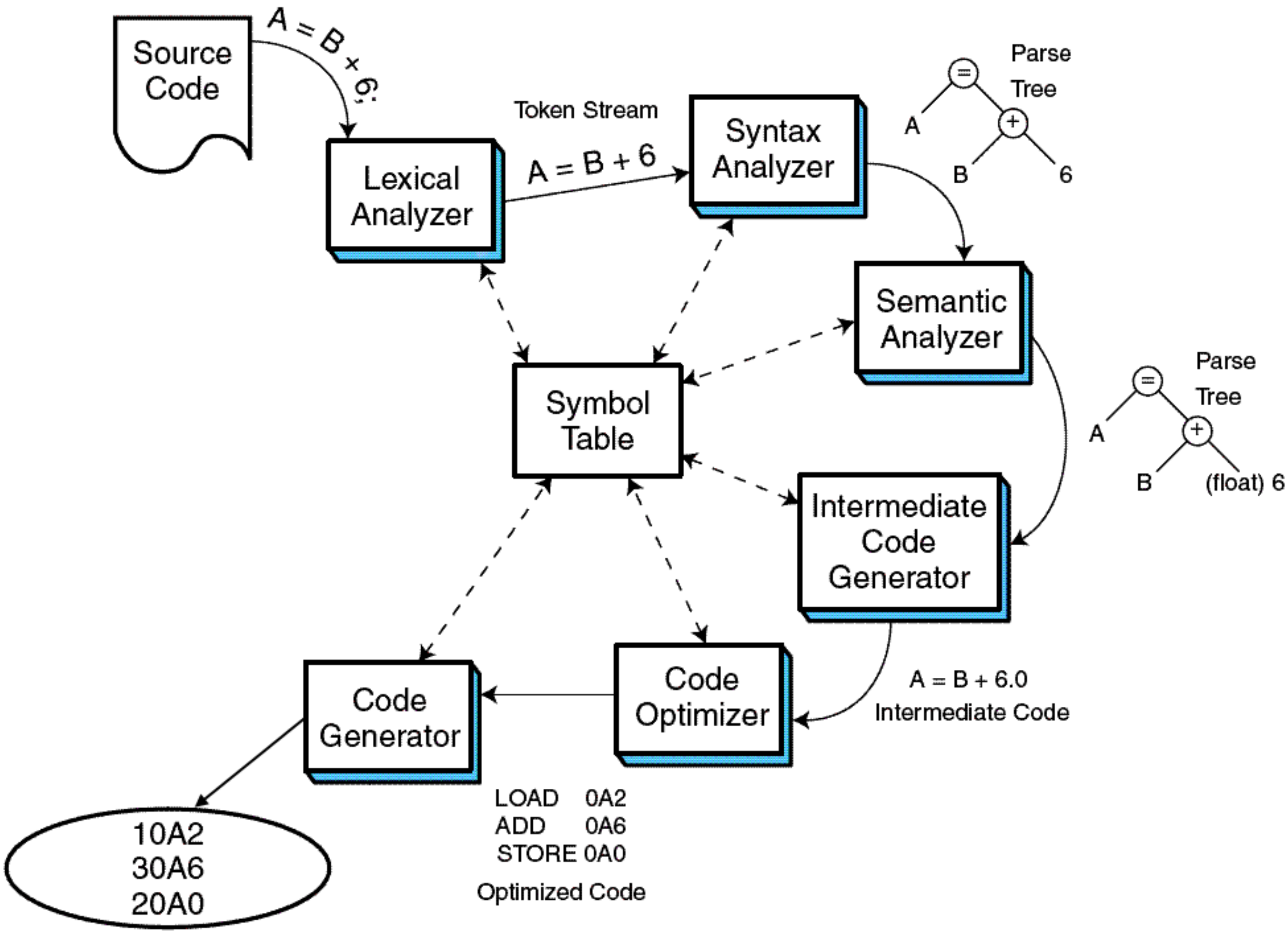
Today's Class

- The C compiler and Makefile
- `printf, scanf`, and format specifiers
- Arrays: one-dimensional and multi-dimensional
- Introduction to Pointers

A simple code compilation

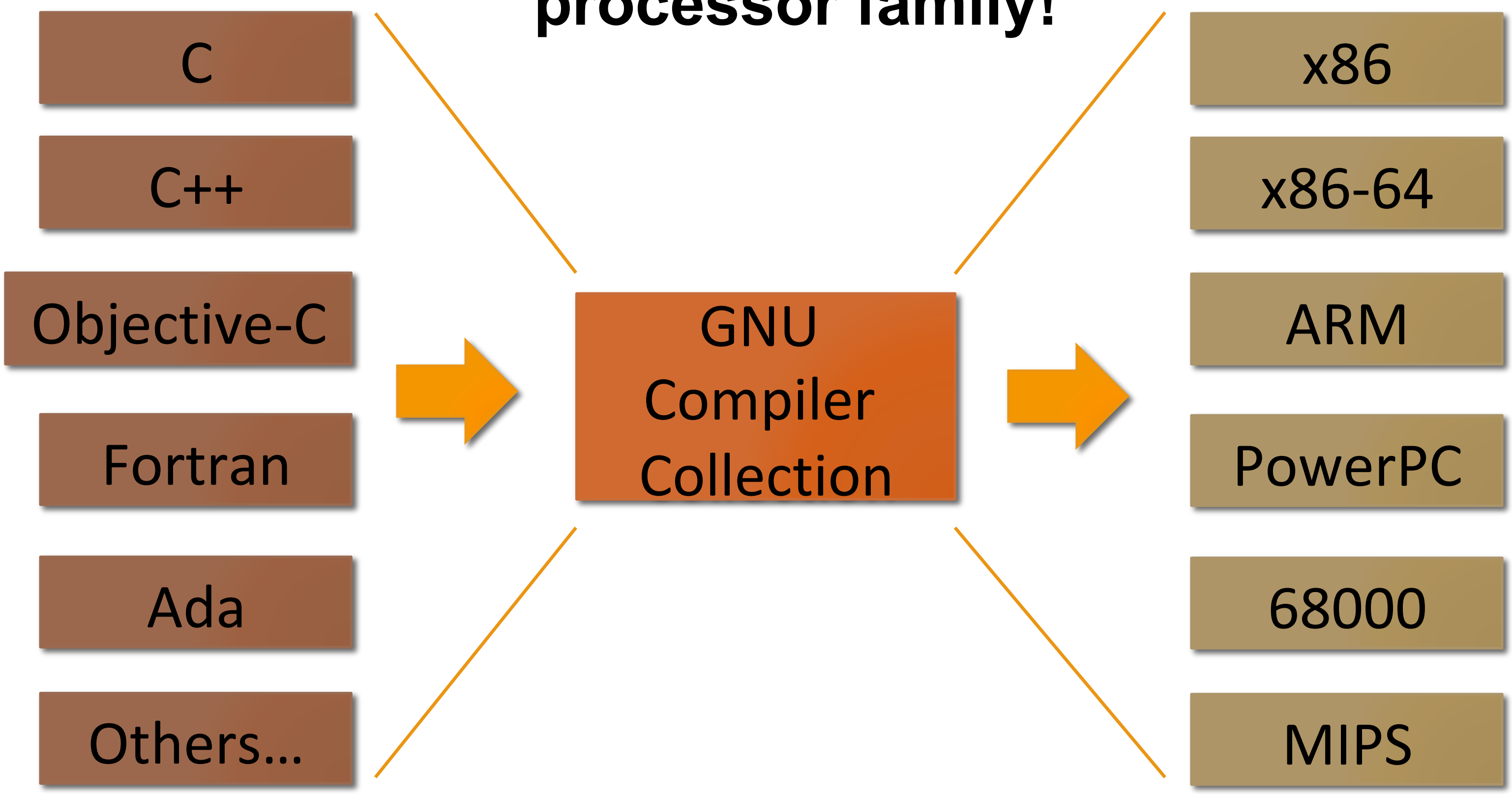
```
unix> gcc –o myprog main.c
unix> ./myprog
```

Compiler Operation



Why So Many Compilation Steps?

We don't *just* care about 1 language or 1 processor family!

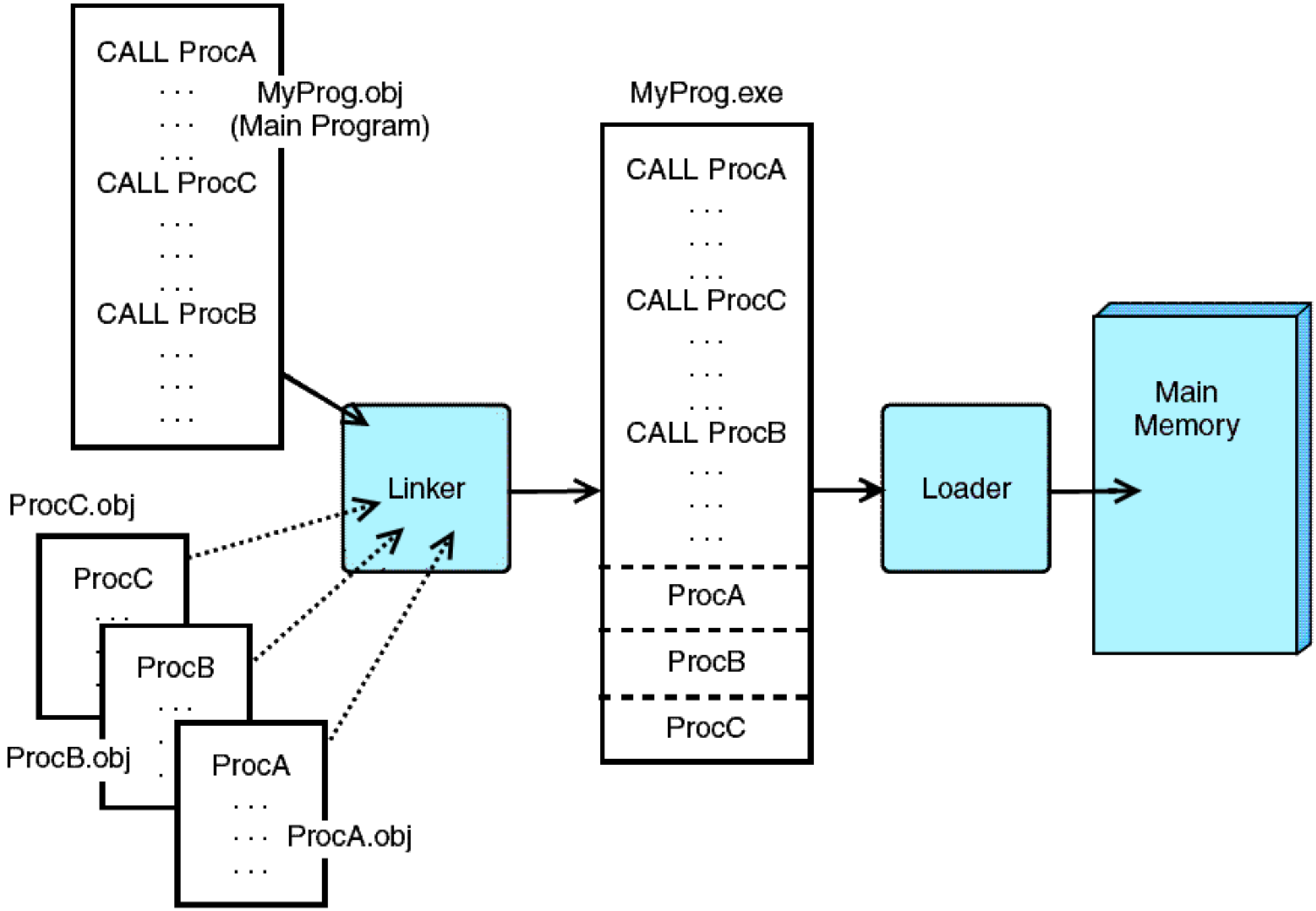


(and many more!)

When your program has multiple files

```
unix> gcc main.c file2.c -o MyProg
unix> ./MyProg
```

Linker + Loader



Result: Program binary (saved on disk)

```
11011101010000001010000001101110101000
00010100000011011101010000001010000001
10111010100000010100000011011101010000
00101000000110111010100000010100000011
01110101000000101000000110111010100000
01010000001101110101000000101000000110
11101010000001010000001101110101000000
10100000011011101010000001010000001101
11010100000010100000011011101010000001
```

Operating System Goals

Security: OK to run file?

Memory management: Find space and create new virtual memory region for this program

File system: Retrieve program binary code from disk

Loader: Place program binary code into memory

Scheduler: Find CPU time for program to run

Context switch – Program starts running

Problem 1

Without Google search, can you identify the linux command to link object files.

Makefile

Goal: Compile our program with one command:

```
unix>  make
```

A **Makefile** is a **text file** that specifies how to compile your program

- The `make` utility reads the Makefile
- You'll learn how this file works in Lab 3

An Intermediate Makefile

```
all: factorial_program
factorial_program: main.o factorial.o output.o
     gcc main.o factorial.o output.o -o
factorial_program
main.o: main.c
     gcc -c main.c
factorial.o: factorial.c
     gcc -c factorial.c
output.o: output.c
     gcc -c output.c
clean:
     rm -rf *.o factorial_program
```

An Advanced Makefile

```
# The variable CC specifies which compiler will
be used.
# (because different unix systems may use
different compilers)
CC=gcc

# The variable CFLAGS specifies compiler options
#   -c :    Only compile (don't link)
#   -Wall:  Enable all warnings about lazy /
dangerous C programming
CFLAGS=-c -Wall
# The final program to build
EXECUTABLE=factorial_program

# -------------------------------------------
```

```
all: $(EXECUTABLE)

$(EXECUTABLE): main.o factorial.o output.o
	$(CC) main.o factorial.o output.o -o $
(EXECUTABLE)

main.o: main.c
	$(CC) $(CFLAGS) main.c

factorial.o: factorial.c
	$(CC) $(CFLAGS) factorial.c

output.o: output.c
	$(CC) $(CFLAGS) output.c

clean:
	rm -rf *.o $(EXECUTABLE)
```

C Tutorial

Print with printf()

```
printf("This is a string\n");

printf("The integer is %i\n", num);

printf("The floating-point values are %g
and %g\n", num1, num2);
```

Output with printf()

Format "Type" Code	Corresponding Variable Type
`d` or `i`	int (interpret as signed 2's comp)
`u`	int (interpret as unsigned)
`x`	int (print as hexadecimal)
`f` or `g`	float/double
`c`	char
`s`	string (null-terminated array of chars)
`p`	An address to which the pointer points

Prefix with `l` or `ll` (i.e. "long" or "long long" for larger 64-bit data types)

- Lots of formatting options not listed here…
  - # of digits before / after decimal point?
  - Pad with zeros?

Input with scanf()

Input from console

```
scanf("%d %c", &myint, &mychar)
```

Requires the **address** of the destination variable

- Use the `&` operator to obtain address

Problem 2 – Read the man pages for printf and scanf

Man(ual) pages exist for common programming functions too

```
unix>  man printf
unix>  man scanf
```

Arrays

Arrays

Contiguous block of memory

You can have arrays for `int, char, float, double`, structures…

```
int myarray[5]; //static declaration
```

NOTE: Name of the array is the address of the first element

address:	4	8	12	16	20
	myarray[0]	myarray[1]	myarray[2]	myarray[3]	myarray[4]

```
printf("%p",myarray); //prints what?
```

2-dimensional arrays

```
int myarray[5][5]; //static declaration
```

Memory map:

Address: 4 myarray[0][0]	Address: 8	Address: 12	Address: 16	Address: 20
Address: 24	Address: 28 myarray[1][1]	Address: 32	Address: 36	Address: 40
Address: 44	Address: 48	Address: 52	Address: 56	Address: 60
Address: 64	Address: 68	Address: 72 myarray[3][2]	Address: 76	Address: 80
Address: 84	Address: 88	Address: 92	Address: 96	Address: 100

Problem 3: Looping through an array

Consider a 3-D array, `int image[256][256][3]` for an RGB color image. The first subscript denotes the number of rows, the second subscript denotes the number of columns, and the third subscript denotes the number of color channels. For example, a pixel at row `i` and column `j` will have an R value given by `image[i][j][0]`, G value given by `image[i][j][1]`, and B value given by `image[i][j][2]`. Any pixel has a yellow color if it's R and G values are 255 and B value is 0. Write a `for` loop to search for the location of the very first yellow pixel in `image`. The search should terminate once the yellow pixel is found. Search in row-wise manner.

Pointers

Pointers are special variables that hold/store memory addresses of other variables.

When a pointer, say iptr, holds the address of an integer variable, say ivar, then we say: “iptr is an integer pointer that points to ivar.”

```
int ivar=45;
int *iptr; iptr = &ivar; //iptr points to ivar
```

ivar: 45

address: 65536

iptr: 65536

address: 65520

‘&’ is ‘address of variable’ operator. For example, `&ivar` translates to: “address of variable ivar”.

‘*’ is ‘value at address stored in pointer’ operator. For example, `*iptr` translates to: “value at address stored in pointer iptr”.

We can have a 'multiple' pointer

Example pointer declaration:

```
int *iptr; //an integer pointer that will point
to an integer

int **dptr; //A double pointer that will point
to an integer pointer


int ***tptr; //A triple pointer pointing to a
double pointer.


int ****quadptr //
```

____

Problem 4

Consider the variables below:

Variable Name: ivar	Pointer variable name: iptr
value: 5	value:
Address: 0xFFABCD	Address: OxAFABAD

```
int ivar=5;
int *iptr;
iptr = &ivar;
printf("\n %u",ivar);  prints_________
printf("\n %u",&ivar);  prints_________
printf("\n %u",&iptr);  prints_________
printf("\n %u",*iptr);  prints_________
```

Problem 5

Variable Name: ivar	Pointer variable name: iptr	Pointer variable name: dptr
value: 5	value:	value:
Address: 0xFFABCD	Address: OxAFABAD	Address: OxFFACBD

```
int ivar=5;
int *iptr;
int **dptr;
iptr = &ivar;
dptr=&iptr;
printf("\n %u",dptr);  prints________
printf("\n %u",iptr);  prints________
printf("\n %u",*dptr);  prints________
printf("\n %u",**dptr);  prints________
printf("\n %u",&dptr);  prints________
printf("\n %u",*(&(iptr)));  prints________
```

Next Class

Pointer basics

Pointers and Arrays
- Dynamic Allocation

Pointers and Structures
- Linked Lists

File I/O in C