ECE 220: Computer Systems & Programming

Lecture 5: Introduction to C

Thomas Moon

February 1, 2024

- MP2 due tonight.

Mock quiz	01/30 - 02/01	CBTF	Short Survey LC-3 practice (HW0)
Quiz 1	02/05 - 02/07	CBTF	LC-3 Programming (up to and including Lecture 4/Lab 1 /MP 1) See Lab1 and Lab2 programming exercise
			An LC-3 web simulator will be available during Quiz 1 - you should NOT use it for your MPs.

Exam	Date & Time	Location	Topic	Practice Questions	Conflict Exam Information
Midterm 1	Thursday 02/15 at 7.00pm - 8.20pm	See below	Lecture 1 to Lecture 06 Associated book chapters, labs,and MPs. (Programming & Concept)	Past exams Worksheets	Submit request **Deadline: 02/11**

Written in C/C++

Windows

XBOX

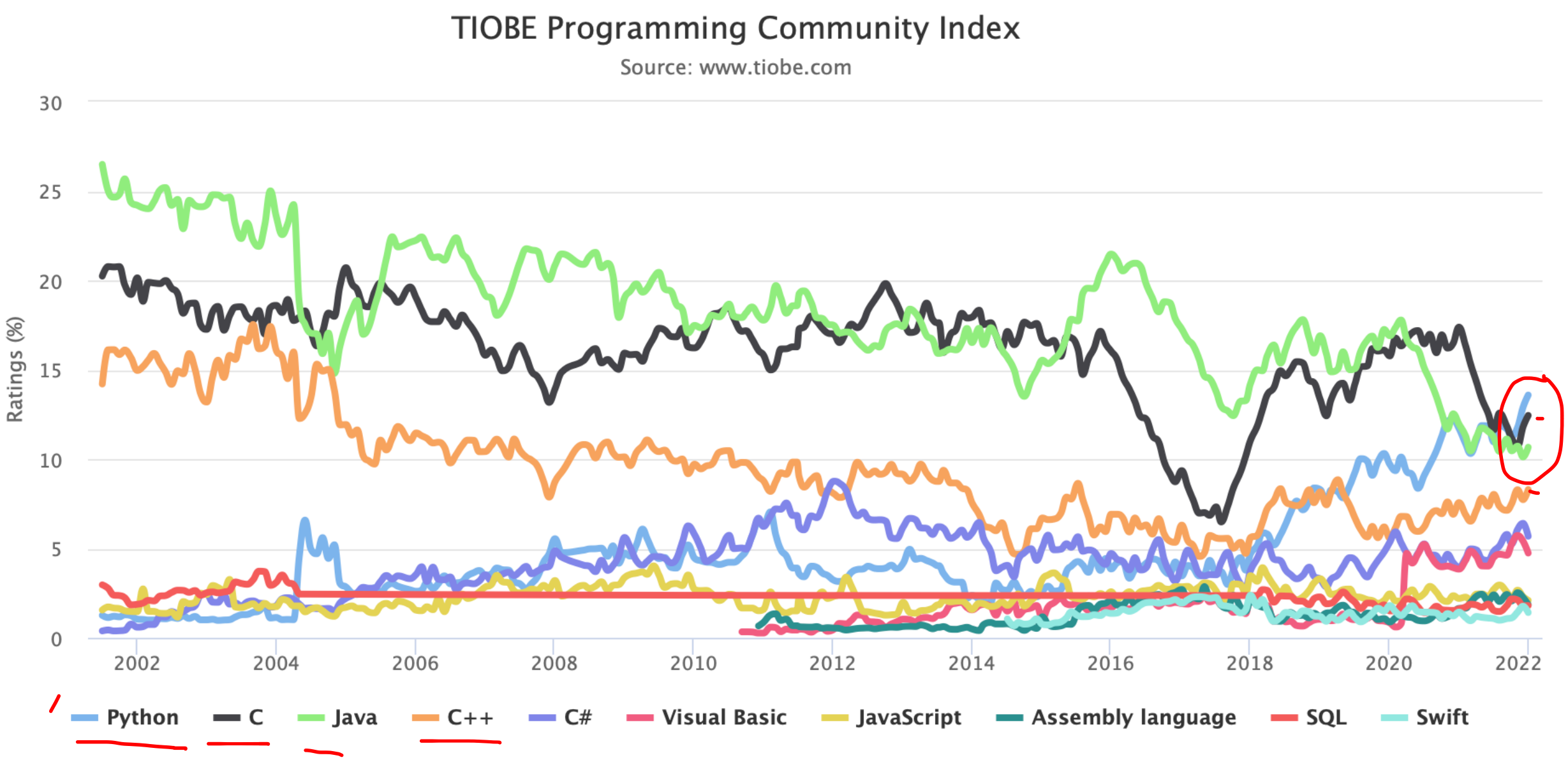
Nintendo

macOS

git

UNREAL ENGINE

Adobe


TIOBE Programming Community Index
Source: www.tiobe.com
Ratings (%)
30
25
20
15
10
5
0
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
2022
Python
C
Java
C++
C#
Visual Basic
JavaScript
Assembly language
SQL
Swift

C – High Level Language

Give symbolic names to values

- Don’t need to know which register or memory location

task: set counter to 5

```
int counter = 5;
```

```
LD      R1, COUNTER
COUNTER .FILL #5
```

`counter` instead of `R1`

C – High Level Language

Provides expressiveness

- Use meaningful symbols that convey meaning
- Simple expressions for common control patterns (if-else, for-while)

task: print 5 to 0

```c
int counter = 5;
while(counter >=0){
    printf("%d", counter);
    counter = counter - 1;
}
```

```
.ORIG   x3000
LD      R2, ASCII_0
LD      R1, COUNTER
LOOP
ADD     R0, R2, R1
OUT
ADD     R1, R1, #-1
BRzp     LOOP

COUNTER .FILL   #5
ASCII_0 .FILL   x30
.END
```

C – High Level Language

Provides abstraction of underlying hardware

- Operations do not depend on instruction set (ISA independent)

Compilation vs Interpretation

Different ways of translating high-level languages

Interpretation

- Interpreter: program that executes program statements
- Pros: Easy to debug, make changes, view intermediate results
- Cons: Programs takes longer to execute
- Languages: Python, Matlab

Compilation

- Translates statements into machine language
- Pros: Executes faster, memory efficient
- Cons: Harder to debug, change requires recompilation
- Languages: C, C++, Fortran, LC-3 assembler

Compiling C Program (CS426-Compiler construction)

Preprocessor

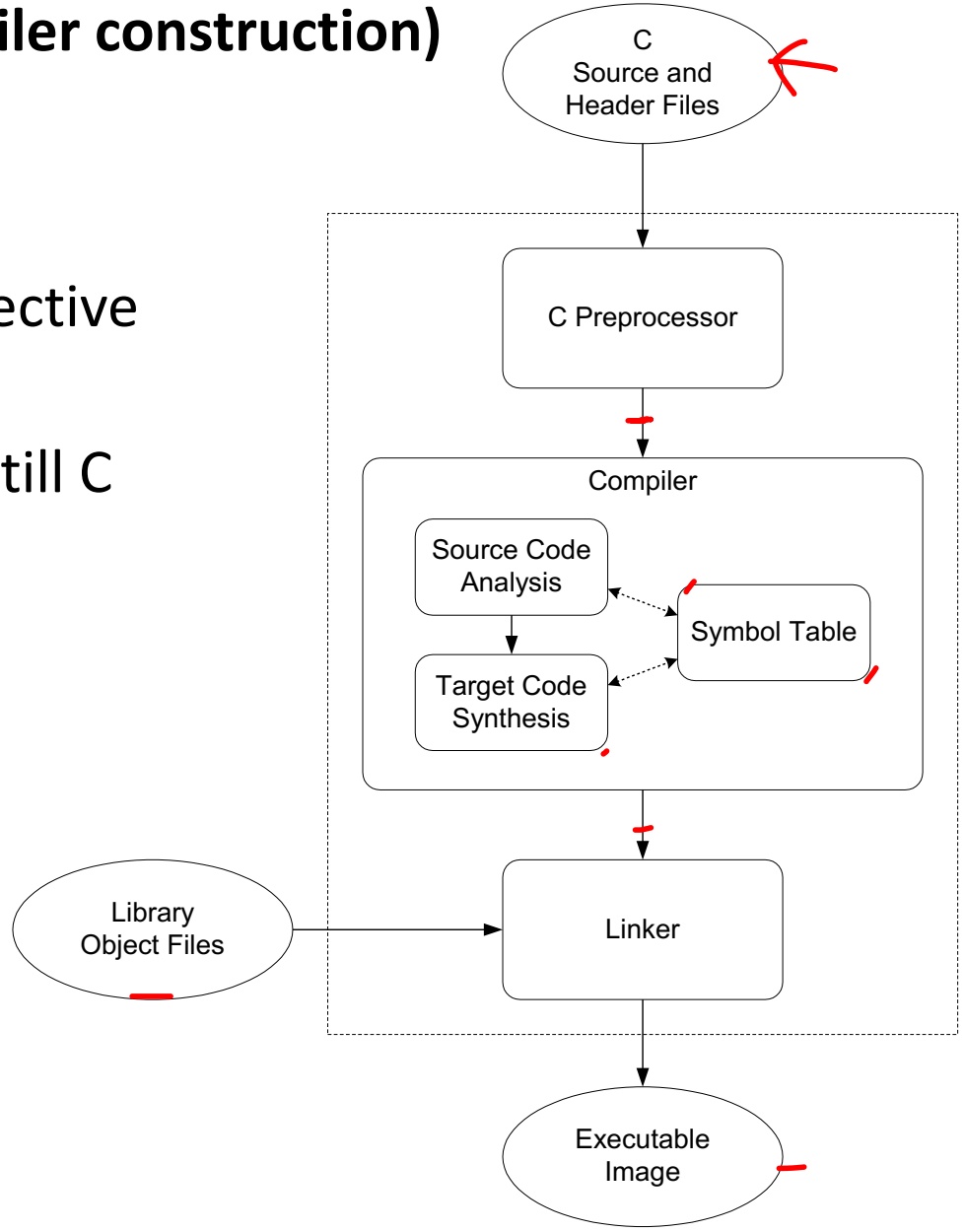
- Macro substitution by C preprocessor directive (e.g. #include, #define)
- “source-level” transformation: output is still C

Compiler

- Generate object file

Linker

- Combine object files into executable image (including libraries)



Compiler

- Source code analysis
  - Source code is broken down and parsed
- Target code synthesis
  - Generate machine code from analyzed code (optimization)
- Symbol table
  - Map between symbolic names and items

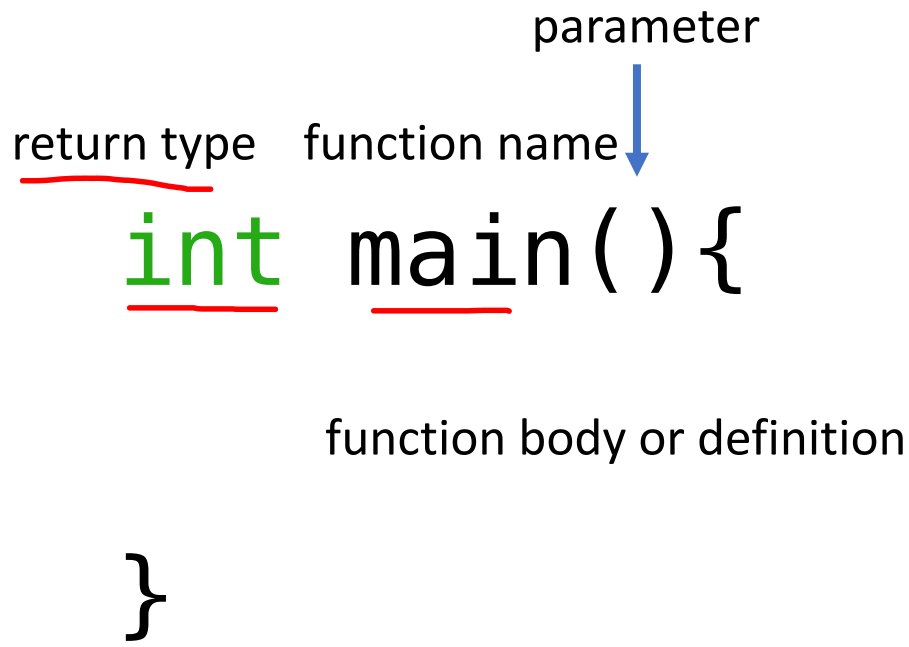
*gcc options
gcc –E: preprocessed output
gcc –S: assembly code
gcc –c: object file

Hello World!

C is all about *functions*!

hw.c

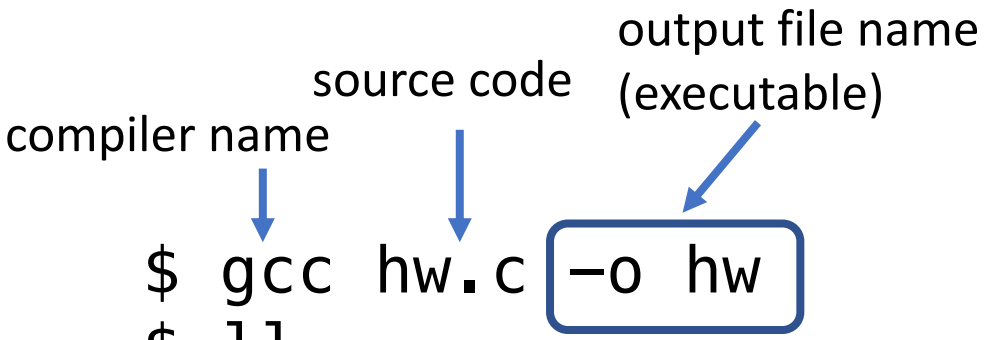
```c
#include <stdio.h>

int main(){
    printf("Hello World!\n");

    return 0;
}
```



semicolon at the end of line
(except control structures, i.e. for, if, while)

return 0 to the caller and terminate

How to compile?




```
$ gcc hw.c -o hw
$ ll
-rwxr-xr-x. 1 tmoon ews 8352 Feb  7 23:50 hw
-rw-r--r--. 1 tmoon ews   80 Feb  7 23:49 hw.c
$ ./hw
Hello World!
```

```
$ gcc hw.c
```

→ by default, the output file name is `a.out`

A Simple C Program

```c
#include <stdio.h>
#define STOP 0

int main()
{
    int counter;
    int startPoint;

    printf("Enter a positive integer: ");
    scanf("%d", &startPoint);

    for(counter = startPoint; counter >= STOP; counter--)
    {
        printf("%d\n", counter);
    }

    return 0;
}
```

Preprocessor Directives

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

- Before compiling, copy content of header file (stdio.h) into source code.
- Header files typically contain description of functions and variables needed by the program.
- <…>: header files in a predefined directory
  "…": header files in the same directory as the C source file

```
#define STOP 0
```

- Before compiling, replace all instances of the string "STOP" with the string "0".
- Used for values that won't change during execution.

main function

```
int main()
```

- Every C program must have a function called main().
- This is the code that is executed when the program is run.

```
$ ./hw
Hello World!
```

cf)

```
int main(int argc, char *argv[])
```

for command line arguments

```
$ ./hw Thomas
Hello World, Thomas!
```

Variable Declarations

```
int counter;
int startPoint;
```

- Variables are used as names for data items.
- Each variable has a type, which tells the compiler how the data to be interpreted.

Input and Output (More details in upcoming lectures)

- Must include `<stdio.h>` to use I/O functions.

```
printf("%d\n", counter);
```

- This call says to print the variable counter as a decimal integer, followed by a linefeed (\n).

```
scanf("%d", &startPoint);
```

- This call says to read a decimal integer and assign it to the variable startPoint.
- Must use ampersand (&) for variables being modified. (Explained in later lecture)

More About Output

- Different formatting options:
  - %d: decimal integer
  - %x: hexadecimal integer
  - %c: ASCII character
  - %f: floating-point number

```
int number = 65;
printf("in decimal: %d, in hex: %x, in character: %c\n"
        , number, number, number);
```

```
in decimal: 65, in hex: 41, in character: A
```

Variables (type + identifier + scope)

Type - 3 Basic data types

- `int` 32-bit 2's complement integer (machine dependent)
- `char` 8-bit character (ASCII)
- `double` 64-bit floating-point
  - `float` 32-bit floating-point

Identifier – variable name

- Any combination of letters, numbers, and underscore(_)
- Case matters
- Cannot begin with a number

Variables (type + identifier + scope)

Scope - the region of the program in which the variable is “alive”

- Local variables
  - Accessible within a block
  - Block defined by open and close braces { }
- Global variables
  - Accessible throughout the program

- Storage class
  - Automatic – Lose value once block is completed (local variables are automatic by default)
  - Static – Retain value throughout program

Example: Global Variable

```
int itsGlobal = 0;

int main()
{
    int itsLocal = 1;    /* local to main */
    printf("Global %d Local %d\n", itsGlobal, itsLocal);
    {
        int itsLocal = 2;    /* local to this block */
        itsGlobal = 4;       /* change global variable */
        printf("Global %d Local %d\n", itsGlobal, itsLocal);
    }
    printf("Global %d Local %d\n", itsGlobal, itsLocal);

    return 0;
}
```

Output:

```
Global 0 Local 1
Global 4 Local 2
Global 4 Local 1
```

Example: Global Variable

```
int itsGlobal = 0;

int main()
{
    int itsLocal = 1;    /* local to main */
    printf("Global %d Local %d\n", itsGlobal, itsLocal);
    {
        //int itsLocal = 2;    /* local to this block */
        itsGlobal = 4;        /* change global variable */
        printf("Global %d Local %d\n", itsGlobal, itsLocal);
    }
    printf("Global %d Local %d\n", itsGlobal, itsLocal);

    return 0;
}
```

← comment this line out

← itsLocal here comes from the only and first itsLocal

Output:

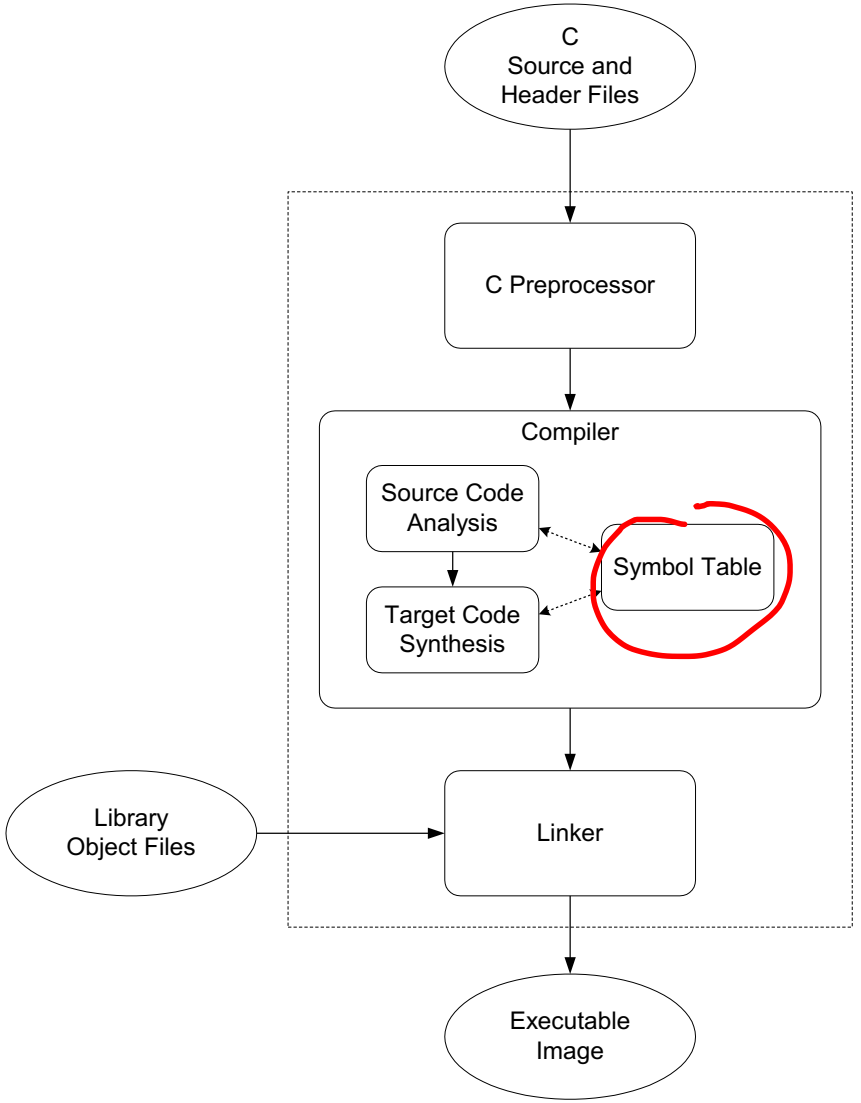
```
Global 0 Local 1
Global 4 Local 1
Global 4 Local 1
```

This is just an example for global vs local. No one wants to use two identical names for their local variables. Use different names!

Memory Allocation for Variables

- When C-compiler compiles a program, it keeps track of variables in a program using a symbol table.

- Symbol table contains
  - variable's name
  - variable's type
  - variable's location (as an offset)
  - variable's scope



Symbol Table

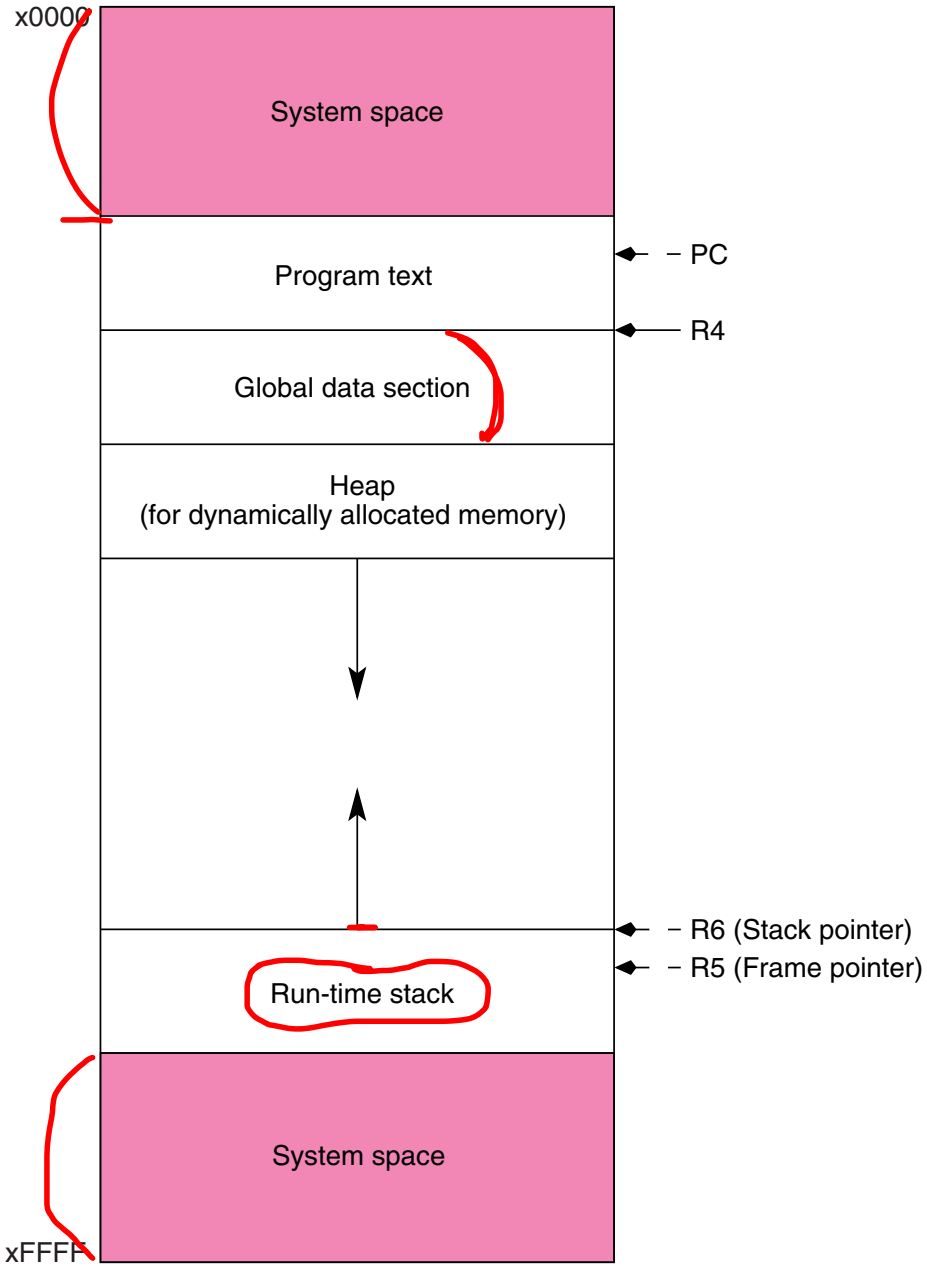
```
int inGlobal;
int outGlobal;

int dummy(int in1, int in2);

int main()
{
    int x,y,z;
    …

}

int dummy(int in1, int in2)
{
    int a,b,c;
    …
}
```

Name	Type	Location (as an offset)	Scope
inGlobal	int	0	global
outGlobal	int	1	global
x	int	0	main
y	int	-1	main
z	int	-2	main
a	int	0	dummy
b	int	-1	dummy
c	int	-2	dummy

Space for Variables (More details in Lecture 10…)

1. Global data section (global variables)
2. Run-time stack (local variables)

- R4 (global pointer) points the first global variable
- R5 (frame pointer) points first local variable
- R6 (stack pointer) points the top of run-time stack



Type Qualifiers

- The basic types (int, char, float/double) can be modified by a qualifier.

- `signed, unsigned`

```
unsigned int d;
```

- `long, short`
  - change its default size
  - No strict definition on the change (depends on the machine)

```
sizeof(char) < sizeof(short int) < sizeof(int) < sizeof(long int)
```

Operators

Expression: combination of variables and literals with operators (e.g. x*y+4)

Statement: expresses a complete unit of work, includes assignment operator (e.g. z = x*y;)

- Assignment operator (=)
- Arithmetic operators

Symbol	Operation	Usage
*	multiply	x * y
/	divide	x / y
%	modulo	x % y
+	addition	x + y
-	subtraction	x - y

Arithmetic operators

```c
int num1 = 11, num2 = 3;
printf("%d + %d=%d \n", num1, num2, num1+num2);
printf("%d - %d=%d \n", num1, num2, num1-num2);
printf("%d * %d=%d \n", num1, num2, num1*num2);
printf("%d / %d=%d \n", num1, num2, num1/num2);
printf("%d %% %d=%d \n", num1, num2, num1%num2);
```

```
11 + 3=14
11 - 3=8
11 * 3=33
11 / 3=3
11 % 3=2
```

Operators (continued)

- Bitwise operators

Symbol	Operation	Usage
~	bitwise NOT	~x
<<	left shift	x << y
>>	right shift	x >> y
&	bitwise AND	x & y
^	bitwise XOR	x ^ y
|	bitwise OR	x | y

```
int bit = 1;
printf("%x << 1 = %x\n", bit, bit<<1);
printf("%x << 2 = %x\n", bit, bit<<2);

int bit1 = 0x01, bit2 = 0x10;
printf("%x | %x  = %x\n", bit1, bit2, bit1|bit2);
printf("%x & %x  = %x\n", bit1, bit2, bit1&bit2);
printf("%x ^ %x  = %x\n", bit1, bit2, bit1^bit2);
```

```
1 << 1 = 2
1 << 2 = 4
1 | 10  = 11
1 & 10  = 0
1 ^ 10  = 11
```

Operators (continued)

- Rational operators (Result is 1 or 0)

Symbol	Operation	Usage
>	greater than	`x > y`
>=	greater than or equal	`x >= y`
<	less than	`x < y`
<=	less than or equal	`x <= y`
→ **==**	**equal**	**`x == y`**
!=	**not equal**	**`x != y`**

- Logical operators (1, if logically true or non-zero)

Symbol	Operation	Usage
!	logical NOT	`!x`
&&	logical AND	`x && y`
||	logical OR	`x || y`

Operators (continued)

```c
int num1 = 10, num2 = 12;
int result1, result2, result3;

result1 = (num1==10 && num2==12);
result2 = (num1<12 || num2>12);
result3 = (!num1);

printf("result1: %d\n", result1);
printf("result2: %d\n", result2);
printf("result3: %d\n", result3);
```

non-zero is TRUE in c

```
result1: 1
result2: 1
result3: 0
```

Operators (continued)

- Increment/Decrement operators: ++,--
- Special operator (conditional)
  - variable = condition ? value_if_true : value_if_false;
  - example: x = (y<z) ? 5 : 7
- Compound Assignment Operators
  - a+=b; <-->a=a+b;
  - a*=b; <-->a=a*b;

```
x = 4;
y = x++;

result
x=5
y=4

x = 4;
y = x;
x++;
```

```
x = 4;
y = ++x;

result
x=5
y=5
```

Example

```c
/*
 * Write a C program to calculate the total amount to pay including tip.
 */


// preprocessor directives


int main()
{
    // declare floating-point variables (total, bill)


    // prompt user to enter the amount of bill


    // calculate the total amount


    // print the result


    // return
}
```

float vs double

- `float` 32-bit floating-point
- `double` 64-bit floating-point

`1.52` (which is `double`) is rounded to `float`

```
float num1 = 1.52;
double num2 = 1.52;

printf("%f\n", num1);
printf("%lf\n", num2);

printf("%d\n", num1 == 1.52);
printf("%d\n", num2 == 1.52);
```

```
1.520000
1.520000
0
1
```

By default, a floating-point literal is `double` type

float vs double

Use suffix `f`
`1.52f` is now `float`

```
    float num1 = 1.52f;
    double num2 = 1.52;

    printf("%f\n", num1);
    printf("%lf\n", num2);

    printf("%d\n", num1 == 1.52f);
    printf("%d\n", num2 == 1.52);
```

```
1.520000
1.520000
1
1
```

Implicit Type Cast

- Automatic type conversion in c

```c
double num1 = 100;
int num2 = 1.52;
char ch = 0xAA61;

printf("num1 = %lf\n", num1);
printf("num2 = %d\n", num2);
printf("ch = %c\n", ch);
```

```
num1 = 100.000000
num2 = 1
ch = a
```

Warning: int / int

```c
// result = 20*30/100
int x = 20, y = 30;
int z = 100;

int result1 = x*y/z;
int result2 = x/z*y;

printf("(x*y)/z = %d\n", result1);
printf("(x/z)*y = %d\n", result2);
```

```
(x*y)/z = 6
(x/z)*y = 0
```