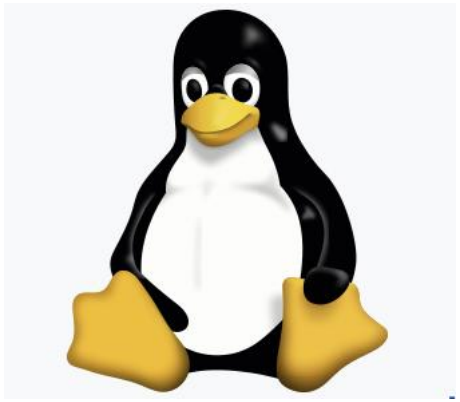
ECE 220: Computer Systems & Programming

Lecture 5: **Introduction to C**

- MP2 due this Thursday by 10:00PM
- Mock Quiz: 9/8 – 9/10
- Quiz1: 09/15 - 09/17

Written in C/C++



Windows

XBOX

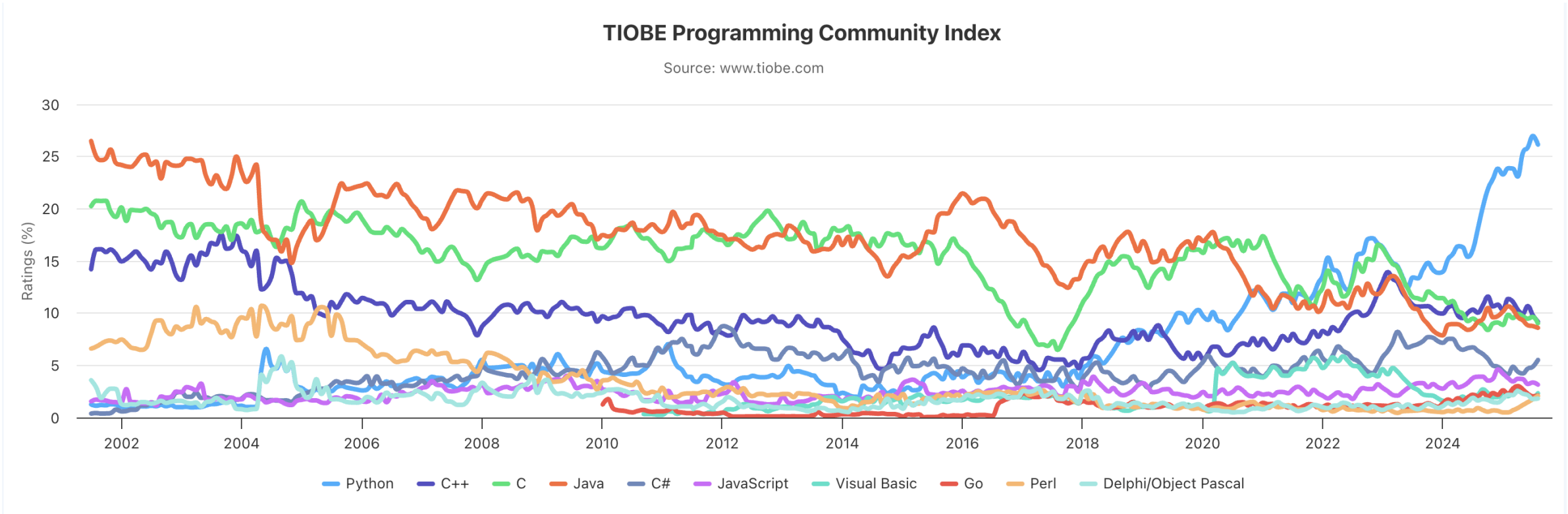
Nintendo

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
2024
Python
C++
C
Java
C#
JavaScript
Visual Basic
Go
Perl
Delphi/Object Pascal

C – High Level Language

Give symbolic names to values

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

Provides expressiveness

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

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

Compiling C Program (CS426-Compiler construction)

Preprocessor

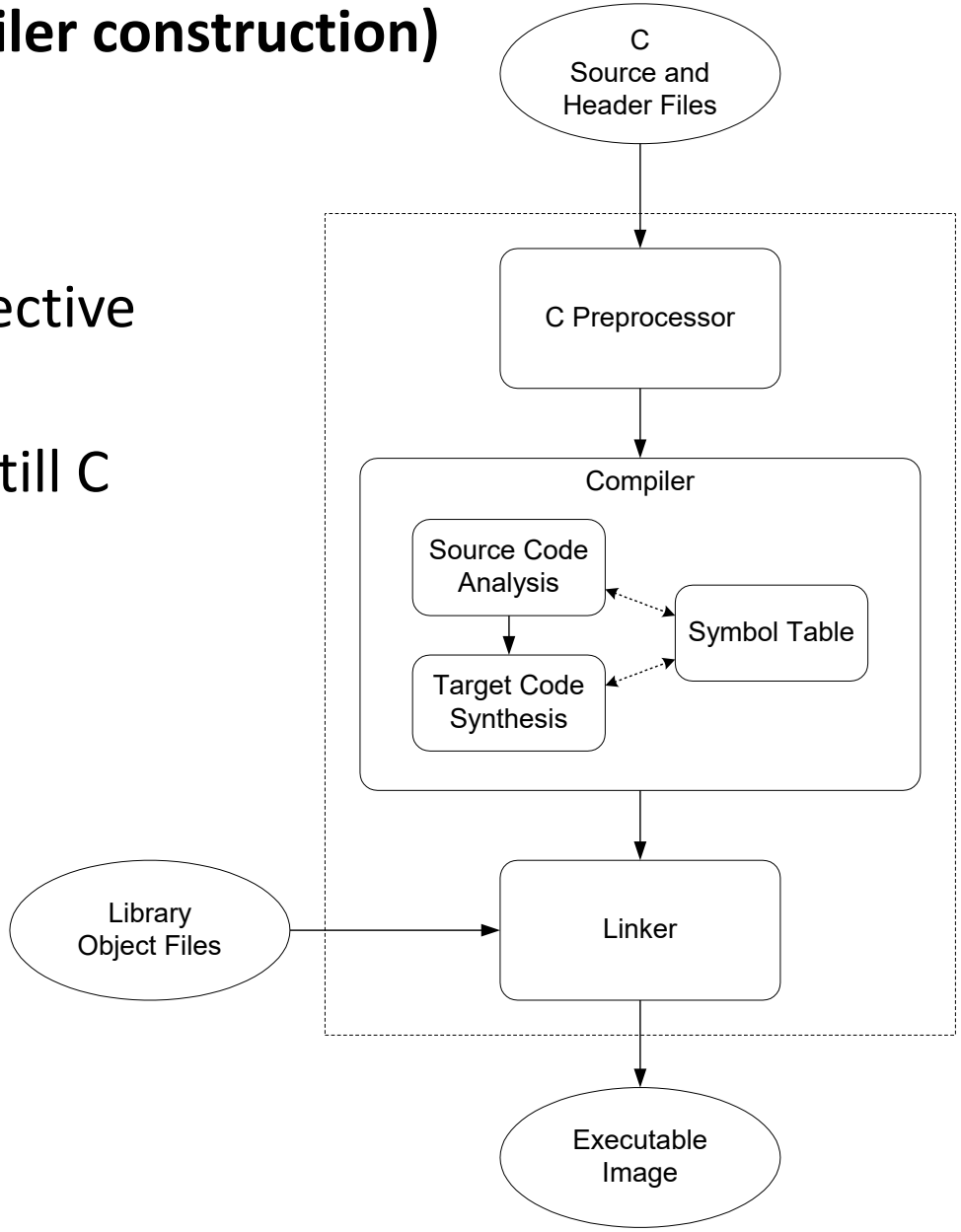
- Macro substitution by C preprocessor directive (e.g. #include, #define)
- “source-level” transformation: output is still C

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

Hello World!

hw.c

```c
#include <stdio.h>

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

    return 0;
}
```

```
$ gcc hw.c -o hw
$./hw
Hello World!
```

A Simple C Program

```c
/*
 * Program Name: countdown from a StartPoint
 */
// Preprocessor directives
#include <stdio.h>
#define STOP 0
int global; // global variable
// Main function
int main()
{
    // Variable declaration
    int counter;
    int startPoint;

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

    // Count down from the input number to 0
    for(counter = startPoint; counter >= STOP; counter--)
    {
        printf("%d\n", counter);
    }

    // Return value
    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.

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 Next 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

- 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: Static Variable

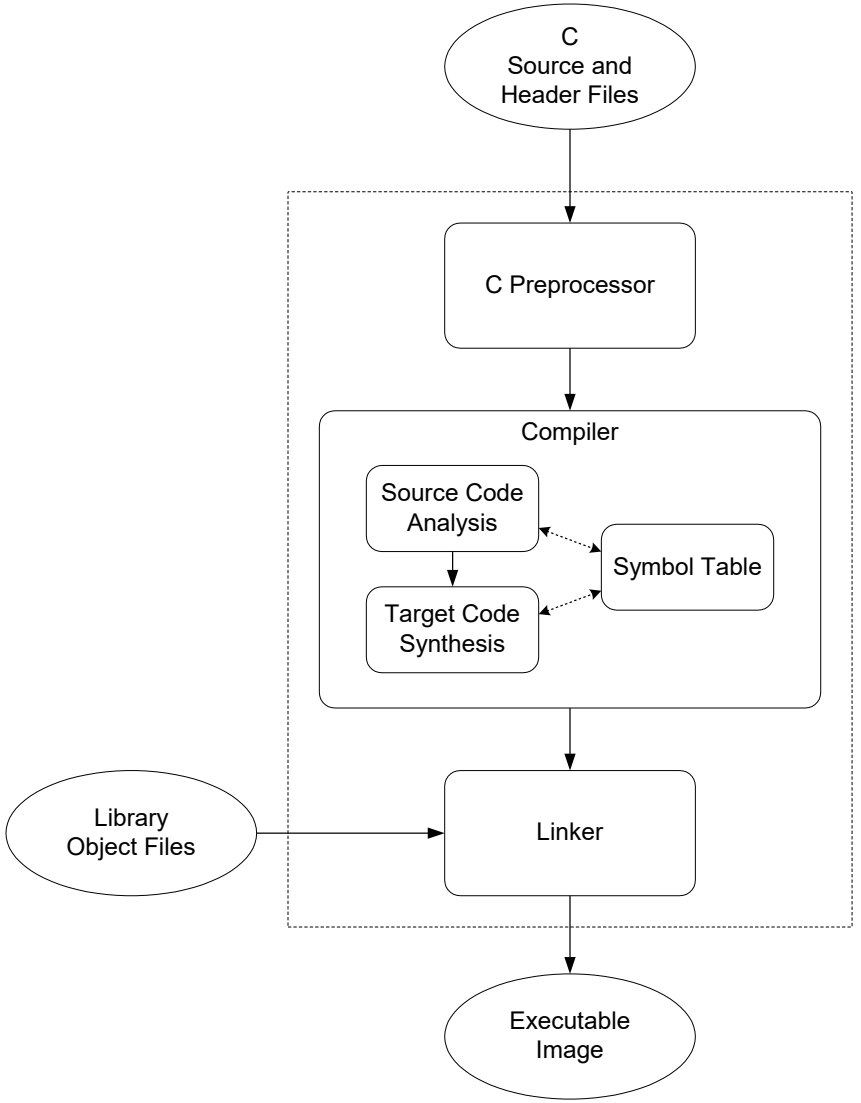
```
int A(){
    static int a = 5;
    a++;
    return a;
}
int main(){
    printf("%d\n", A());
    printf("%d\n", A());

    a = 10; //compile error

    return 0;
}
```

Output:
6
7

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

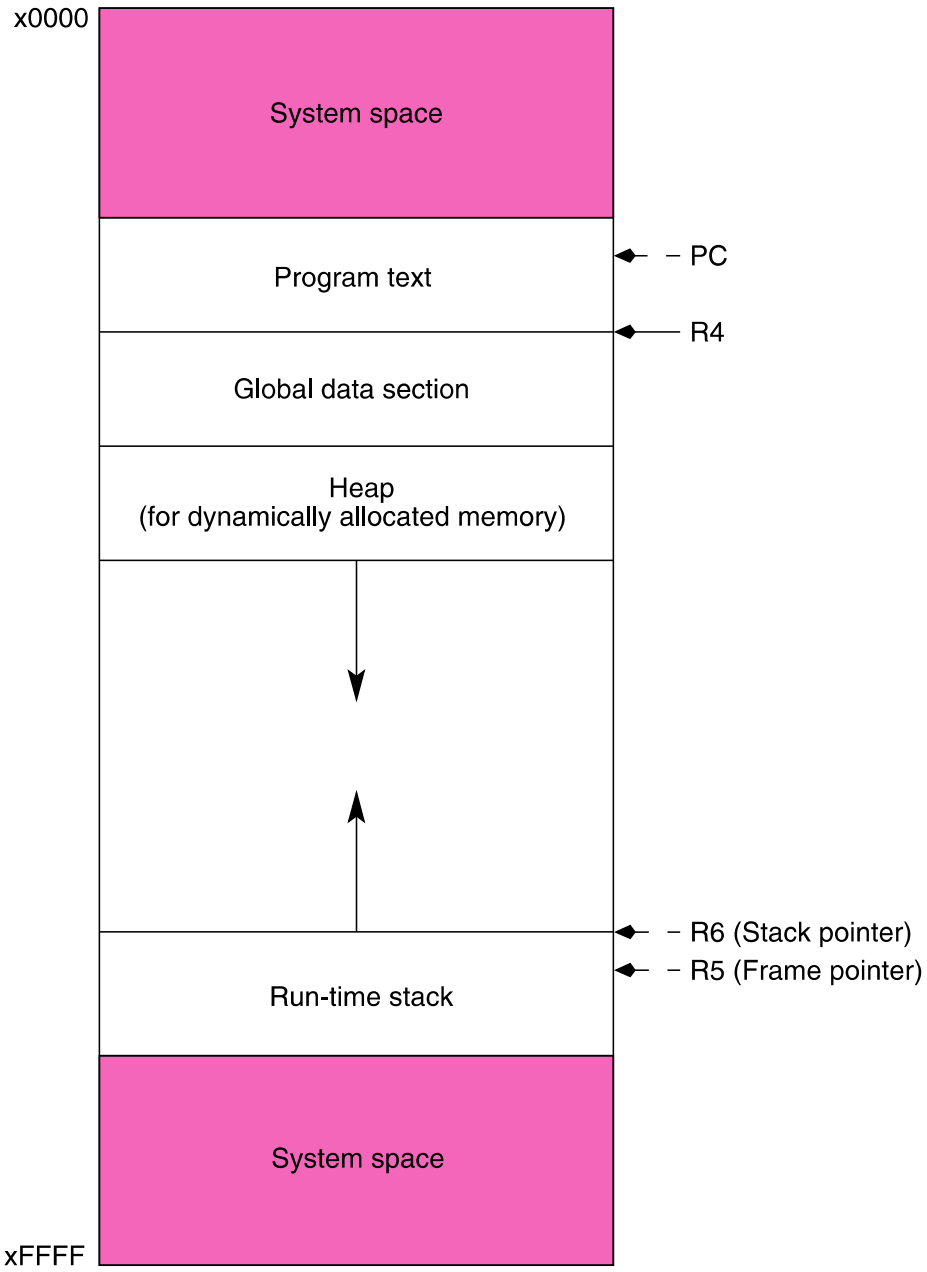
```
int inGlobal;
int outGlobal;

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.

- `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

Ex)
8%3

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`

- 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`

Operators (continued)

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

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

Ex)
7 & 8 = 0b0111 & 0b1000 = 0
7 && 8 = true && true = 1

- 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;

result
x=5
y=5
```

Example

/*
 * Write a C program to calculate the function

$$f(x) = \frac{1}{5}\cos(\omega x) \text{ on the interval } x \in [0, \pi].$$

Your program should ask user to enter all the relevant values (n, ω)
and print
A table of n pairs $(x_i, f(x_i))$, where $i = 0,1,2, \ldots (n-1) and\ x_i = \frac{i\pi}{n}$

Implementation should include a loop construct and must call standard functions, scanf, printf, and cos

*/

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.

cf)

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

Example code (git): main_test.c
compile with – gcc main_test.c –o main_test
Execute with - ./main_test 1 9