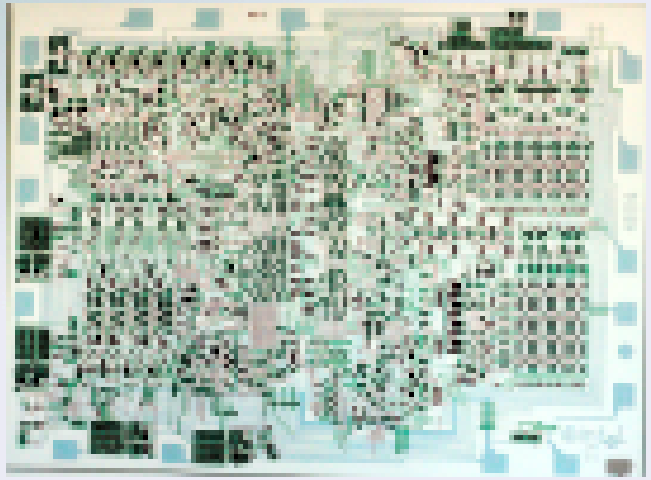


Lecture 12a: Programming in The C Language

CSIS11: **Computer Architecture and Organization**

Readings

- *Chapters 11-12* Patt and Patel: Introduction to Computing Systems...
- *The C Language* - Kernighan and Ritchie - Second Edition, strongly recommended
- Beej's Guide to C Programming (Tutorial) - Instructor Repo
- Beej's Guide to C Programming (Library Reference) - Instructor Repo

Remaining Schedule

Day	Content	Chapter(s)
4/21	C: Introduction, Variables and Operators	11/12
4/23	C: Control Structures	13
4/28	C: Functions	14
4/30	C: Testing and Debugging	15
5/5	C: Pointers, Arrays and Structs	16
5/7	C: I/O	18
5/12	Review and Work on Final Project	
5/14	Review and Work on Final Project	
5/21	**Final Exam – Wed, May 21, 2025 6-8:30 PM**	

C: A High-Level Language

- Gives symbolic names to values, don't need to know which register or memory location
- Provides abstraction of underlying hardware, operations do not depend on instruction set
  - example: can write "a = b * c", even though LC-3 doesn't have a multiply instruction
- Provides expressiveness, use meaningful symbols that convey meaning
  - simple expressions for common control patterns (if-then-else)
- Safeguards against bugs
  - can enforce rules or conditions at compile-time or run-time

Compilation vs. Interpretation

Different ways of translating high-level language:

- **Interpret -** single-step, where interpreter manages the process
- **Compile -** multi-step process to create program

Interpretation

- interpreter = program that executes program statements
- generally one line/command at a time
- limited processing
- easy to debug, make changes, view intermediate results
- languages: Python, LISP, Perl, Java, Matlab, C-shell
- Process: `python myprog.py`

Compilation

- translates statements into machine language
- does not execute, but creates executable program
- performs optimization over multiple statements
- change requires recompilation
- can be harder to debug, since executed code may be different
- languages: C, C++, Fortran, Pascal
- Process: *Compile* -> *Link* -> *Load*

Compile a C Program

Entire mechanism is usually called the "compiler"

Preprocessor

- macro substitution
- conditional compilation
- "source-level" transformations
- output is still C

Compiler

- generates object file
- machine instructions

Compiler

Source Code Analysis - *depends on language (not on target machine)*

- parses programs to identify its pieces
- variables, expressions, statements, functions, etc.

Code Generation - *very dependent on target machine*

- generates machine code from analyzed source
- may optimize machine code to make it run more efficiently

Symbol Table

- map between symbolic names and items
- like assembler, but more kinds of information

Link/Load a C Program

Linker

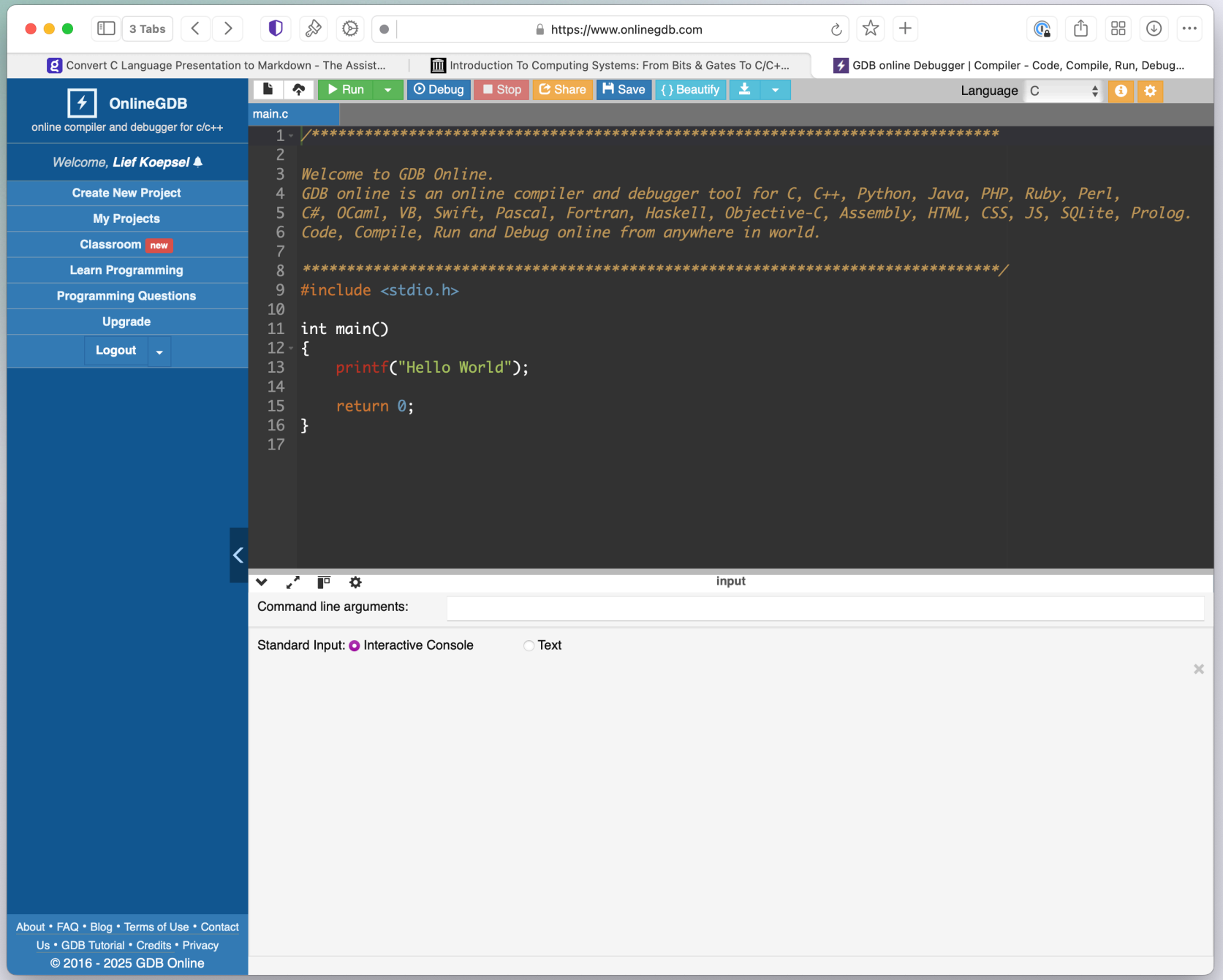
- combine object files (including libraries) into executable image

Load

- can be as simple as typing executable image name
- could also require downloading to a target computer and executing remotely

Our Test bed

OnlineGDB


3 Tabs
https://www.onlinegdb.com
Convert C Language Presentation to Markdown - The Assist...
Introduction To Computing Systems: From Bits & Gates To C/C+...
GDB online Debugger | Compiler - Code, Compile, Run, Debug...
OnlineGDB
online compiler and debugger for c/c++
Welcome, Lief Koepsel
Create New Project
My Projects
Classroom new
Learn Programming
Programming Questions
Upgrade
Logout
Run
Debug
Stop
Share
Save
{ } Beautify
Language C
main.c
/***
Welcome to GDB Online.
GDB online is an online compiler and debugger tool for C, C++, Python, Java, PHP, Ruby, Perl,
C#, OCaml, VB, Swift, Pascal, Fortran, Haskell, Objective-C, Assembly, HTML, CSS, JS, SQLite, Prolog.
Code, Compile, Run and Debug online from anywhere in world.
***/
#include <stdio.h>
int main()
{
printf("Hello World");
return 0;
}
input
Command line arguments:
Standard Input: Interactive Console
Text
About • FAQ • Blog • Terms of Use • Contact Us • GDB Tutorial • Credits • Privacy
© 2016 - 2025 GDB Online

A Simple C Program

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

/* Function: main                                    */
/* Description: counts down from user input to STOP */
main()
{
  /* variable declarations */
  int counter;  /* an integer to hold count values */
  int startPoint; /* starting point for countdown */

  /* prompt user for input */
  printf("Enter a positive number: ");
  scanf("%d", &startPoint);  /* read into startPoint */

  /* count down and print count */
  for (counter=startPoint; counter >= STOP; counter--)
    printf("%d\n", counter);
}
```

Preprocessor Directives

`#include <stdio.h>`

- Before compiling, copy contents of header file (stdio.h) into source code.
- Header files typically contain descriptions of functions and variables needed by the program.
- no restrictions -- could be any C source code

`#define STOP 0`

- Before compiling, replace all instances of the string "STOP" with the string "0"
- Called a macro
- Used for values that won't change during execution, but might change if the program is reused. (Must recompile.)

Comments

- **NO** Begin with `/*` and end with `*/`
- **NO** Can span multiple lines
- **USE** `//` and not the above
- Cannot have a comment within a comment
- As before, use comments to help reader, not to confuse or to restate the obvious
- Don't comment out code, to test, leads to debugging errors

main Function

- Every C program must have a function called main().
- Its best to provide a return type for the function (*void* is typical)
- This is the code that is executed when the program is run.
- The code for the function lives within brackets:

```
void main()
{
  // code goes here
}
```

Variable Declarations

Variables are used as names for data items.

Each variable has a type, which tells the compiler how the data is to be interpreted (and how much space it needs, etc.).

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

`int` is a predefined integer type in C.

Input and Output

Variety of I/O functions in C Standard Library.

Must include `<stdio.h>` to use them.

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

- String contains characters to print and formatting directions for variables.
- This call says to print the variable counter as a decimal integer, followed by a linefeed (`\n`).

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

- String contains formatting directions for looking at input.
- This call says to read a decimal integer and assign it to the variable startPoint. (Don't worry about the & yet.)

More About Output

Can print arbitrary expressions, not just variables

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

Print multiple expressions with a single statement

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

Different formatting options:

- `%d` - decimal integer
- `%x` - hexadecimal integer
- `%c` - ASCII character
- `%f` - floating-point number

Examples

This code:

```
printf("%d is a prime number.\n", 43);
printf("43 plus 59 in decimal is %d.\n", 43+59);
printf("43 plus 59 in hex is %x.\n", 43+59);
printf("43 plus 59 as a character is %c.\n", 43+59);
```

produces this output:

```
43 is a prime number.
43 + 59 in decimal is 102.
43 + 59 in hex is 66.
43 + 59 as a character is f.
```

Examples of Input

Many of the same formatting characters are available for user input.

```
scanf("%c", &nextChar);
```

reads a single character and stores it in nextChar

```
scanf("%f", &radius);
```

reads a floating point number and stores it in radius

```
scanf("%d %d", &length, &width);
```

reads two decimal integers (separated by whitespace), stores the first one in length and the second in width

Must use ampersand (`&`) for variables being modified. (Explained in Chapter 16.)

Basic C Elements

- Variables
  - named, typed data items
- Operators
  - predefined actions performed on data items
  - combined with variables to form expressions, statements
- Rules and usage

Data Types

C has three basic data types:

- **int** - integer (at least 16 bits)
- **double** - floating point (at least 32 bits)
- **char** - character (at least 8 bits)

Exact size can vary, depending on processor

int is supposed to be "natural" integer size; for LC-3, that's 16 bits -- 32 bits for most modern processors

Variable Names

- Any combination of letters, numbers, and underscore (_)
- Case matters
  - "sum" is different than "Sum"
- Cannot begin with a number
  - usually, variables beginning with underscore are used only in special library routines
- Only first 31 characters are used

Literals

Integer

```
123    /* decimal */
-123
0x123  /* hexadecimal */
```

Floating point

```
6.023
5E12       /* 5.0 x 10^12 */
```

Character

```
'c'
'\n'   /* newline */
'\xA'  /* ASCII 10 (0xA) */
```

Scope: Global and Local

Where is the variable accessible?

Global: accessed anywhere in program

Local: only accessible in a particular region

Compiler infers scope from where variable is declared

- programmer doesn't have to explicitly state

Variable is local to the block in which it is declared

- block defined by open and closed braces { }
- can access variable declared in any "containing" block

Global variable is declared outside all blocks

Example

```c
#include <stdio.h>
int itsGlobal = 0;

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);
}
```

Output

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

Operators

Programmers manipulate variables using the operators provided by the high-level language.

Variables and operators combine to form expressions and statements which denote the work to be done by the program.

Each operator may correspond to many machine instructions.

Example: The multiply operator (*) typically requires multiple LC-3 ADD instructions.

Expression

Any combination of variables, constants, operators, and function calls

- every expression has a type, derived from the types of its components (according to C typing rules)

Examples:

```
counter >= STOP
x + sqrt(y)
x & z + 3 || 9 - w-- % 6
```

Statement

Expresses a complete unit of work

- executed in sequential order

Simple statement ends with semicolon

```
z = x * y; /* assign product to z */
y = y + 1; /* after multiplication */
;  /* null statement */
```

Operators

(1) Function

- what does it do?

(2) Precedence

- in which order are operators combined?
- Example: "a * b + c * d" is the same as "(a * b) + (c * d)" because multiply (*) has a higher precedence than addition (+)

(3) Associativity

- in which order are operators of the same precedence combined?
- Example: "a - b - c" is the same as "(a - b) - c" because add/sub associate left-to-right

Assignment Operator

Changes the value of a variable.

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

1. Evaluate right-hand side.
2. Set value of left-hand side variable to result.

Assignment Operator

All expressions evaluate to a value, even ones with the assignment operator.

For assignment, the result is the value assigned.

- usually (but not always) the value of the right-hand side
- type conversion might make assigned value different than computed value

Assignment associates right to left.

```
y = x = 3;
```

y gets the value 3, because (x = 3) evaluates to the value 3.

Arithmetic Operators

Symbol	Operation	Usage	Precedence	Assoc
*	multiplication	x * y	6	l-to-r
/	division	x / y	6	l-to-r
%	modulo	x % y	6	l-to-r
+	addition	x + y	7	l-to-r
-	subtraction	x - y	7	l-to-r

All associate left to right.

- / % have higher precedence than + -.

Arithmetic Expressions

If mixed types, smaller type is "promoted" to larger.

```
x + 4.3
```

if x is int, converted to double and result is double

Integer division -- fraction is dropped.

```
x / 3
```

if x is int and x=5, result is 1 (not 1.666666...)

Modulo -- result is remainder.

```
x % 3
```

if x is int and x=5, result is 2.

Bitwise Operators

Symbol	Operation	Usage	Precedence	Assoc
~	bitwise NOT	~x	4	r-to-l
<<	left shift	x << y	8	l-to-r
>>	right shift	x >> y	8	l-to-r
&	bitwise AND	x & y	11	l-to-r
^	bitwise XOR	x ^ y	12	l-to-r
|	bitwise OR	x | y	13	l-to-r

- Operate on variables bit-by-bit.
- Like LC-3 AND and NOT instructions.
- Shift operations are logical (not arithmetic).
- Operate on values -- neither operand is changed.

Logical Operators

Symbol	Operation	Usage	Precedence	Assoc
!	logical NOT	!x	4	r-to-l
&&	logical AND	x && y	14	l-to-r
||	logical OR	x || y	15	l-to-r

Treats entire variable (or value) as TRUE (non-zero) or FALSE (zero).

Result is 1 (TRUE) or 0 (FALSE).

Relational Operators

Symbol	Operation	Usage	Precedence	Assoc
>	greater than	x > y	9	l-to-r
>=	greater than or equal	x >= y	9	l-to-r
<	less than	x < y	9	l-to-r
<=	less than or equal	x <= y	9	l-to-r
==	equal	x == y	10	l-to-r
!=	not equal	x != y	10	l-to-r

Result is 1 (TRUE) or 0 (FALSE).

Note: Don't confuse equality (==) with assignment (=).

Special Operators: ++ and --

Changes value of variable before (or after) its value is used in an expression.

Symbol	Operation	Usage	Precedence	Assoc
++	postincrement	x++	2	r-to-l
--	postdecrement	x--	2	r-to-l
++	preincrement	++x	3	r-to-l
--	predecrement	--x	3	r-to-l

Pre: Increment/decrement variable before using its value.

Post: Increment/decrement variable after using its value.

Using ++ and --

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

Results: x = 5, y = 4

(because x is incremented after assignment)

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

Results: x = 5, y = 5

(because x is incremented before assignment)

Practice with Precedence

Assume a=1, b=2, c=3, d=4.

```
x = a * b + c * d / 2;  /* x = 8 */
```

same as:

```
x = (a * b) + ((c * d) / 2);
```

For long or confusing expressions, use parentheses, because reader might not have memorized precedence table.

Note: Assignment operator has lowest precedence, so all the arithmetic operations on the right-hand side are evaluated first.

Symbol Table

Like assembler, compiler needs to know information associated with identifiers

- in assembler, all identifiers were labels and information is address

Compiler keeps more information

- Name (identifier)
- Type
- Location in memory
- Scope

Name	Type	Offset	Scope
amount	int	0	main
hours	int	-3	main
minutes	int	-4	main
rate	int	-1	main
seconds	int	-5	main
time	int	-2	main

Local Variable Storage

Local variables are stored in an activation record, also known as a stack frame.

Symbol table "offset" gives the distance from the base of the frame.

R5 is the frame pointer – holds address of the base of the current frame.

A new frame is pushed on the run-time stack each time a block is entered.

Because stack grows downward, base is the highest address of the frame, and variable offsets are <= 0.

Allocating Space for Variables

Global data section

- All global variables stored here (actually all static variables)
- R4 points to beginning

Run-time stack

- Used for local variables
- R6 points to top of stack
- R5 points to top frame on stack
- New frame for each block (goes away when block exited)

Variables and Memory Locations

In our examples, a variable is always stored in memory.

When assigning to a variable, must store to memory location.

A real compiler would perform code optimizations that try to keep variables allocated in registers.

Why?

Example: Symbol Table

Name	Type	Offset	Scope
inGlobal	int	0	global
inLocal	int	0	main
outLocalA	int	-1	main
outLocalB	int	-2	main

Statement	Equivalent assignment
x += y;	x = x + y;
x -= y;	x = x - y;
x *= y;	x = x * y;
x /= y;	x = x / y;
x %= y;	x = x % y;
x &= y;	x = x & y;
x |= y;	x = x | y;
x ^= y;	x = x ^ y;
x <<= y;	x = x << y;
x >>= y;	x = x >> y;

Credit:

Introduction to C and Variables and Operators

University of Texas at Austin

CS310H - Computer Organization

Spring 2010

Don Fussell