

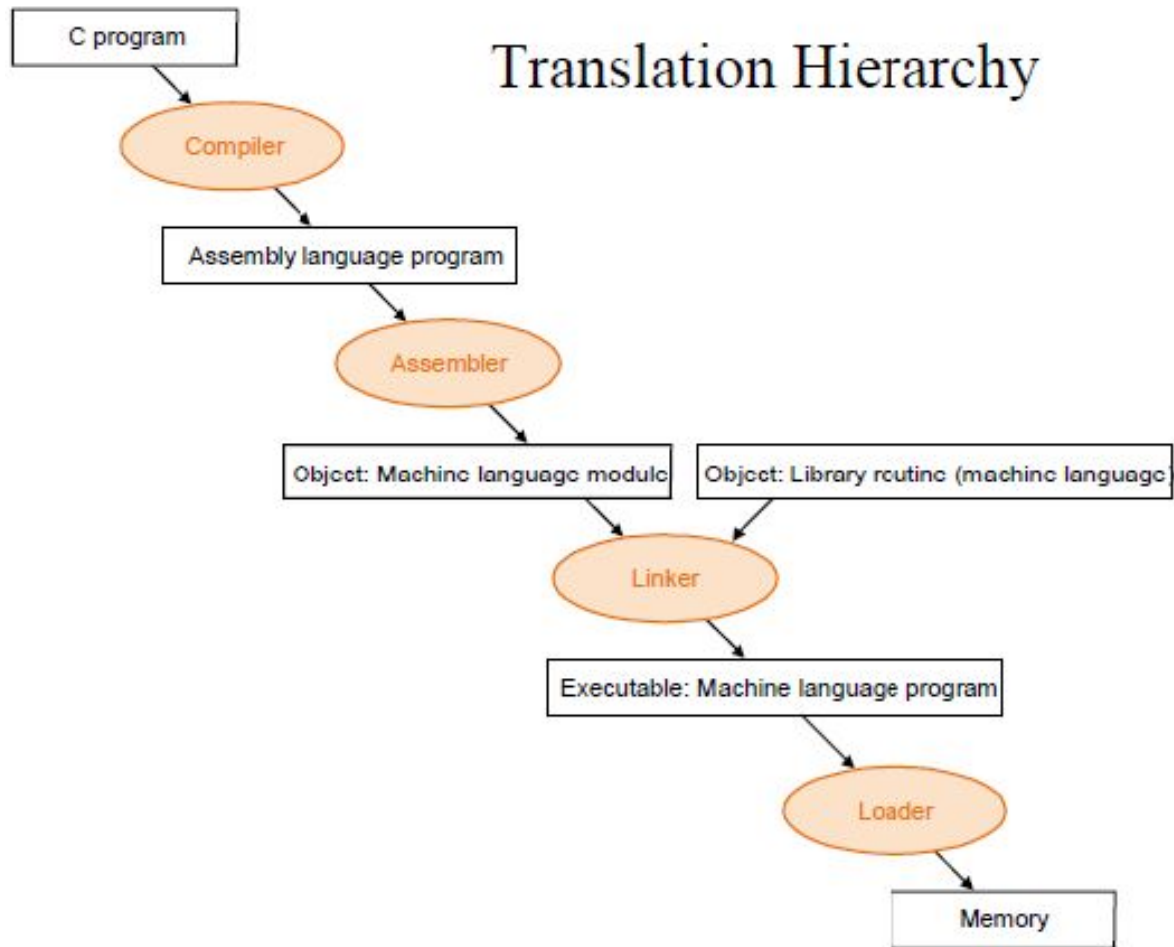
A Beginners guide to Assembly

By Tom Glint and Rishiraj
CS301 | Fall 2020

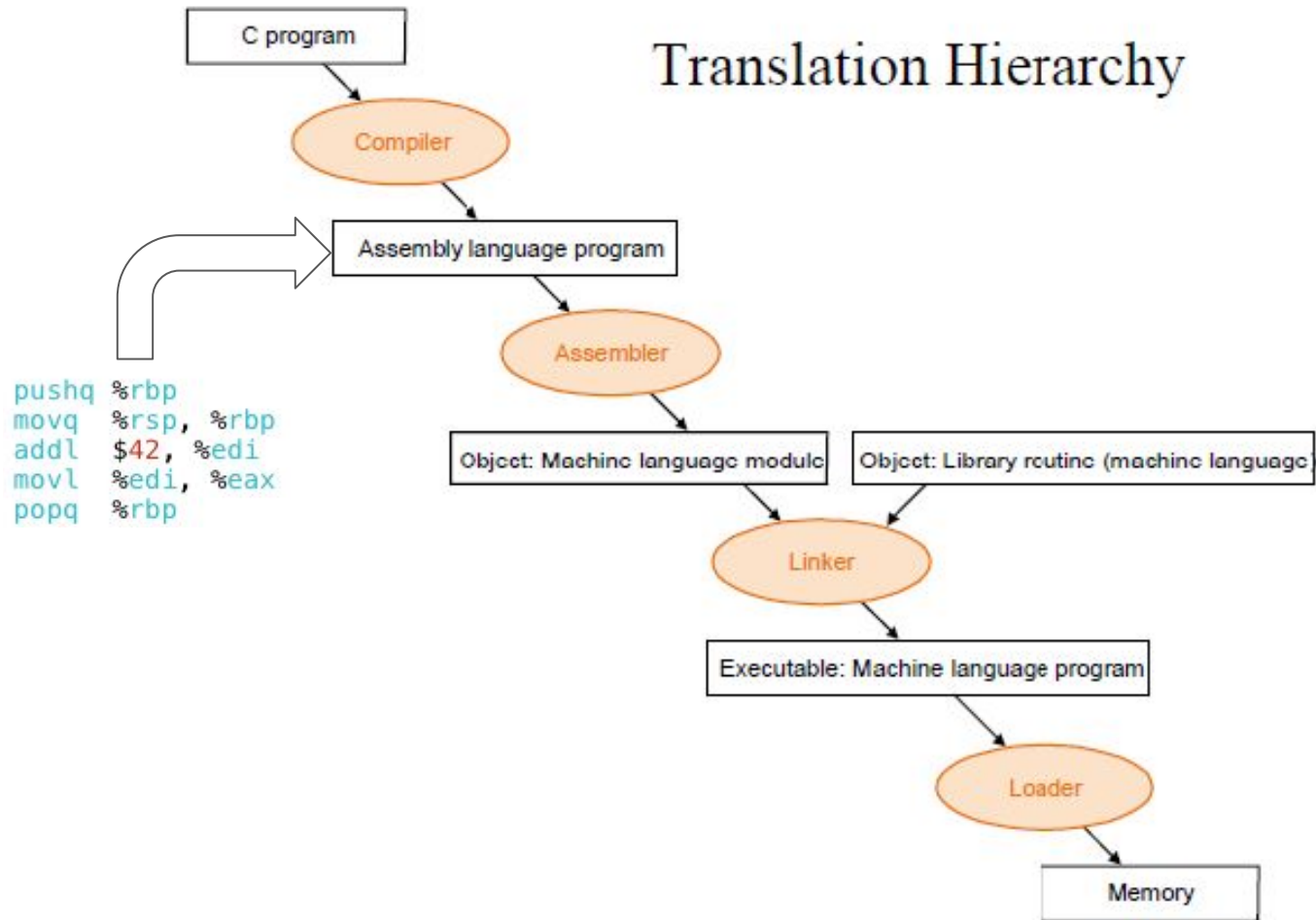
Contributors: Varun and Shreyas



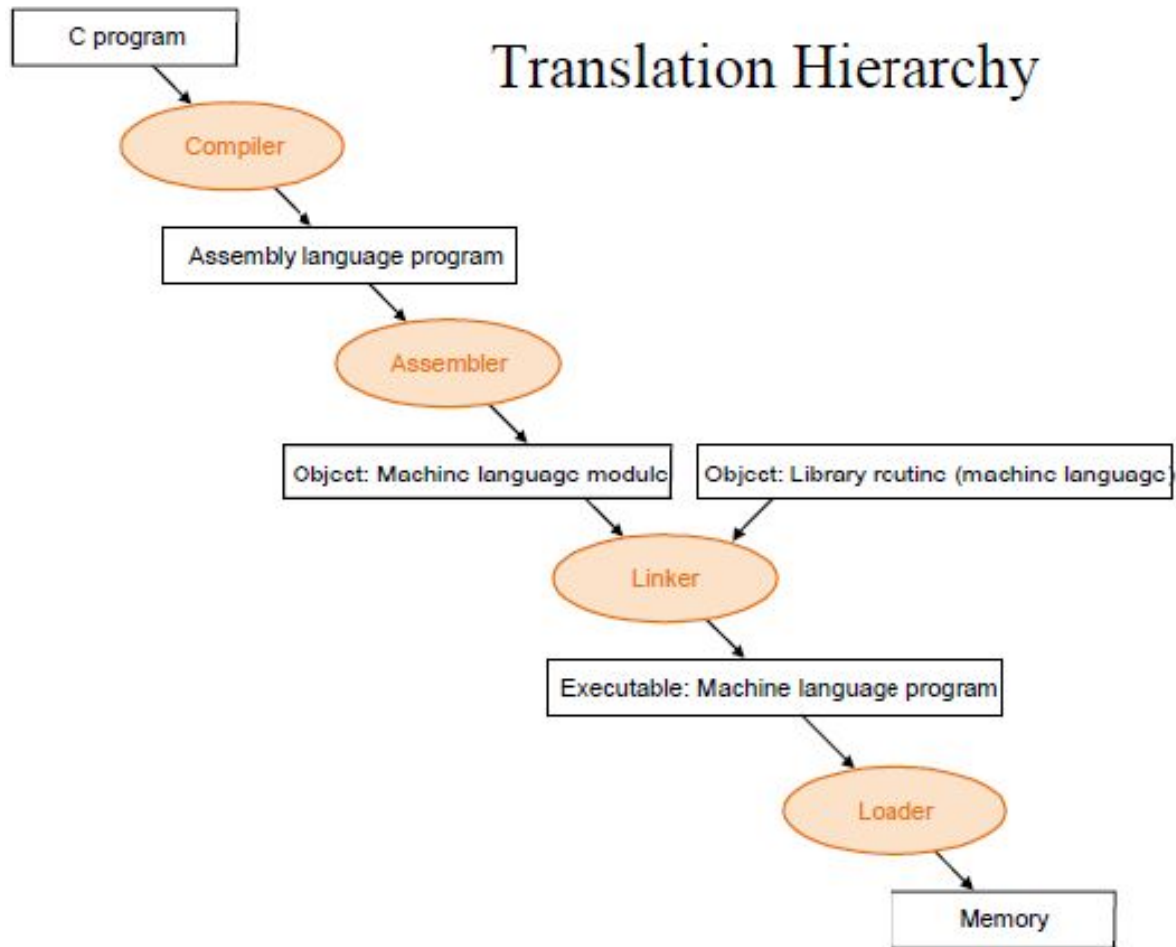
Translation Hierarchy



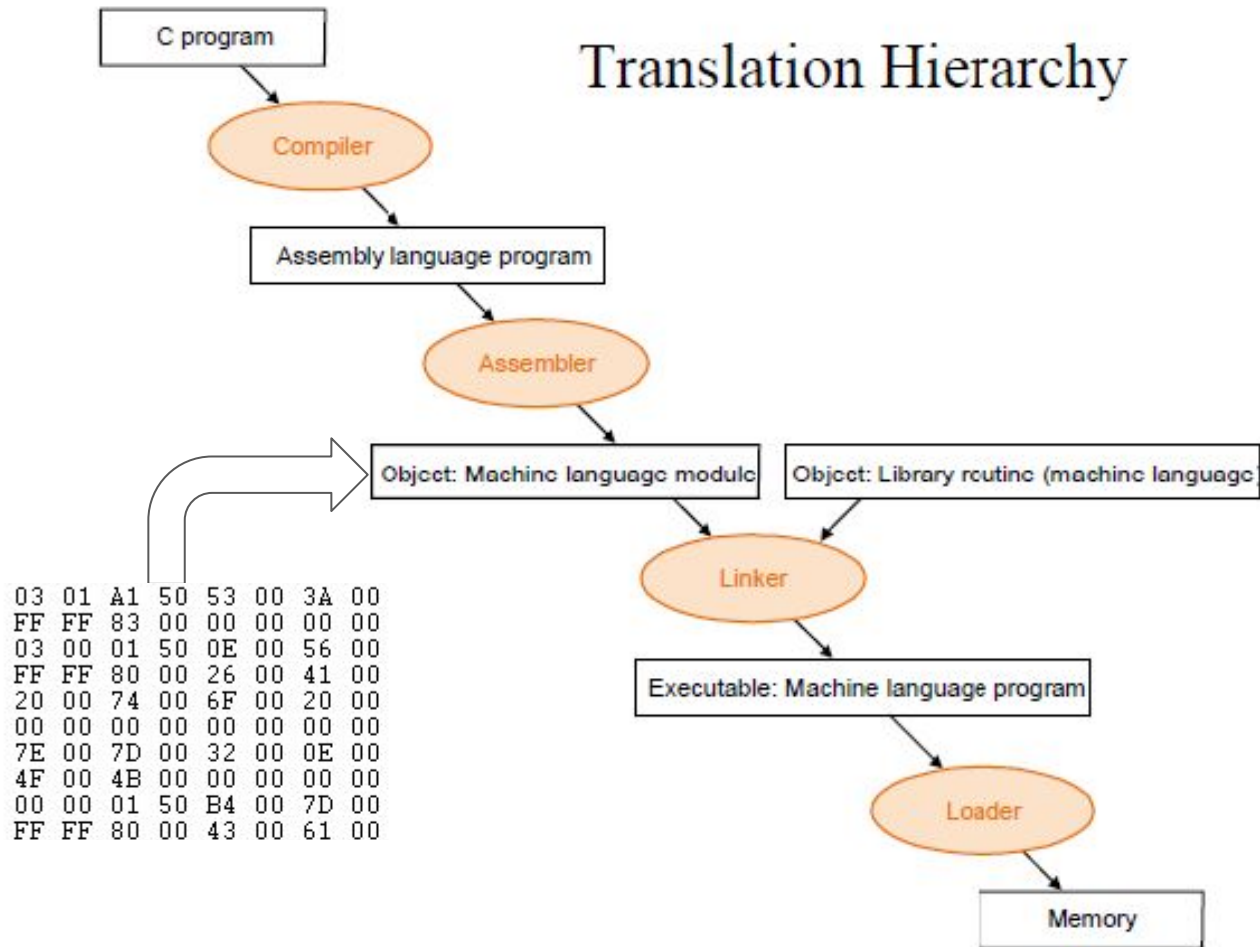
Translation Hierarchy



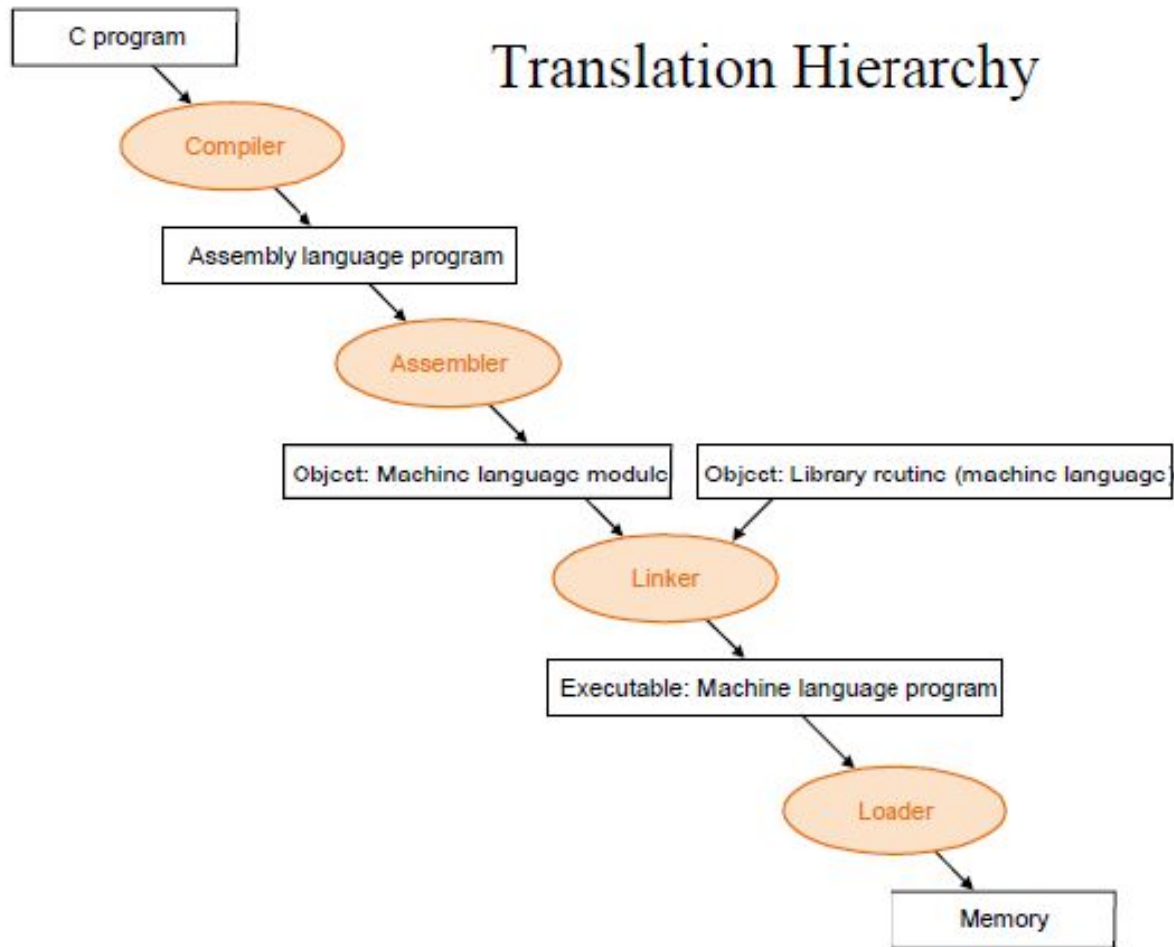
Translation Hierarchy



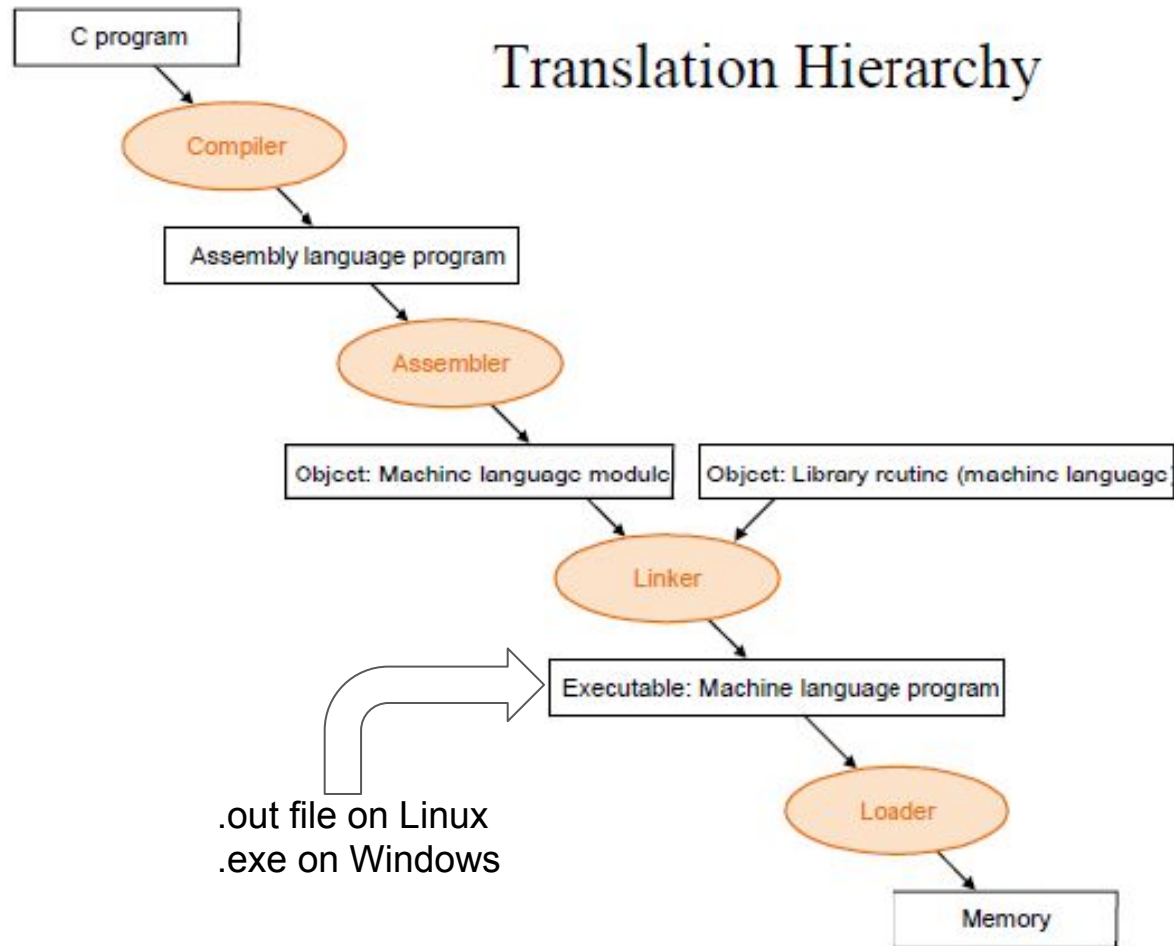
Translation Hierarchy



Translation Hierarchy

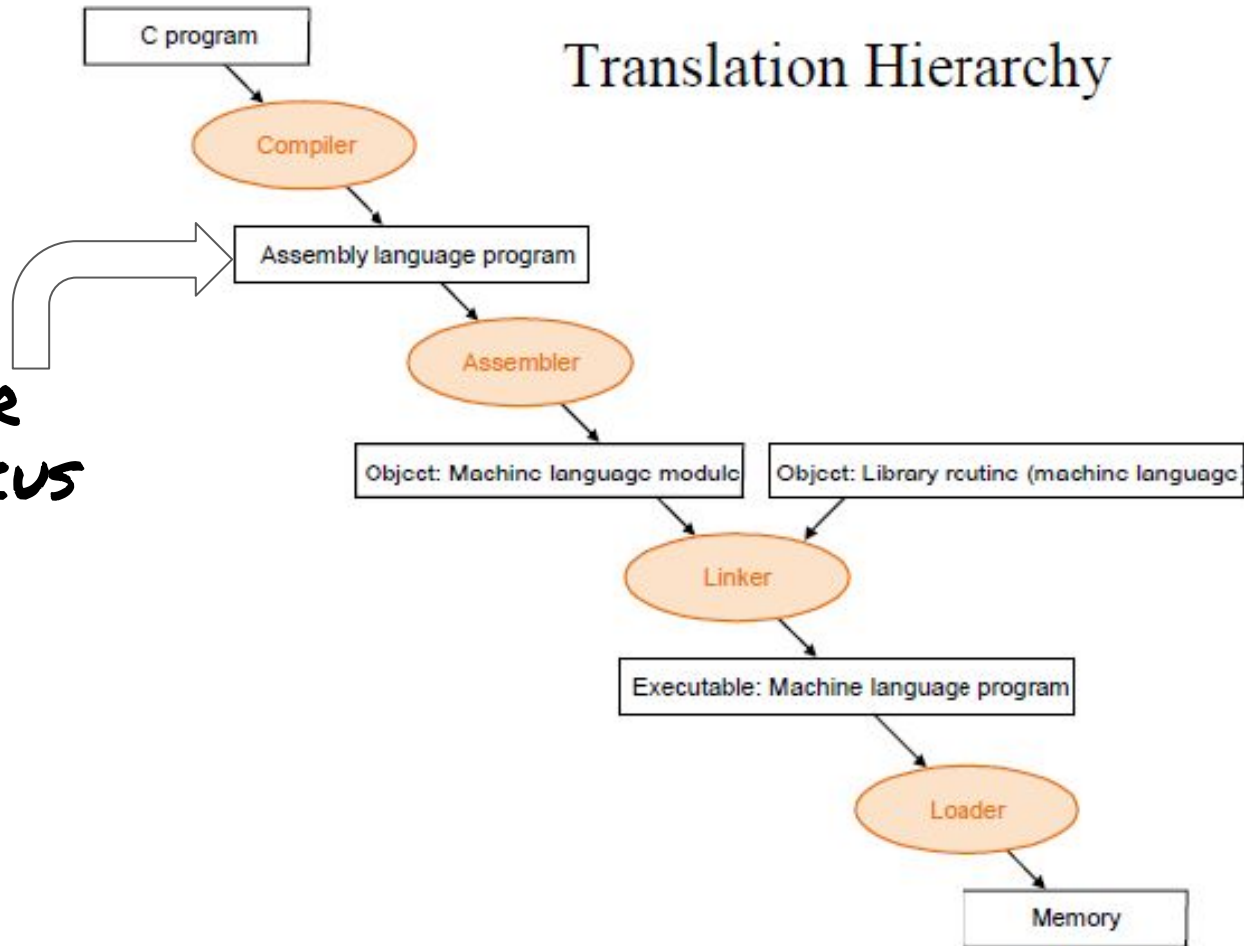


Translation Hierarchy



Translation Hierarchy

**OUR
FOCUS**



Prominent ISAs



ARM

IIT-Madras Develops 'India's First Microprocessor', Shakti

By Indo-Asian News Service | Updated: 2 November 2018 16:03 IST

[f Share on Facebook](#)

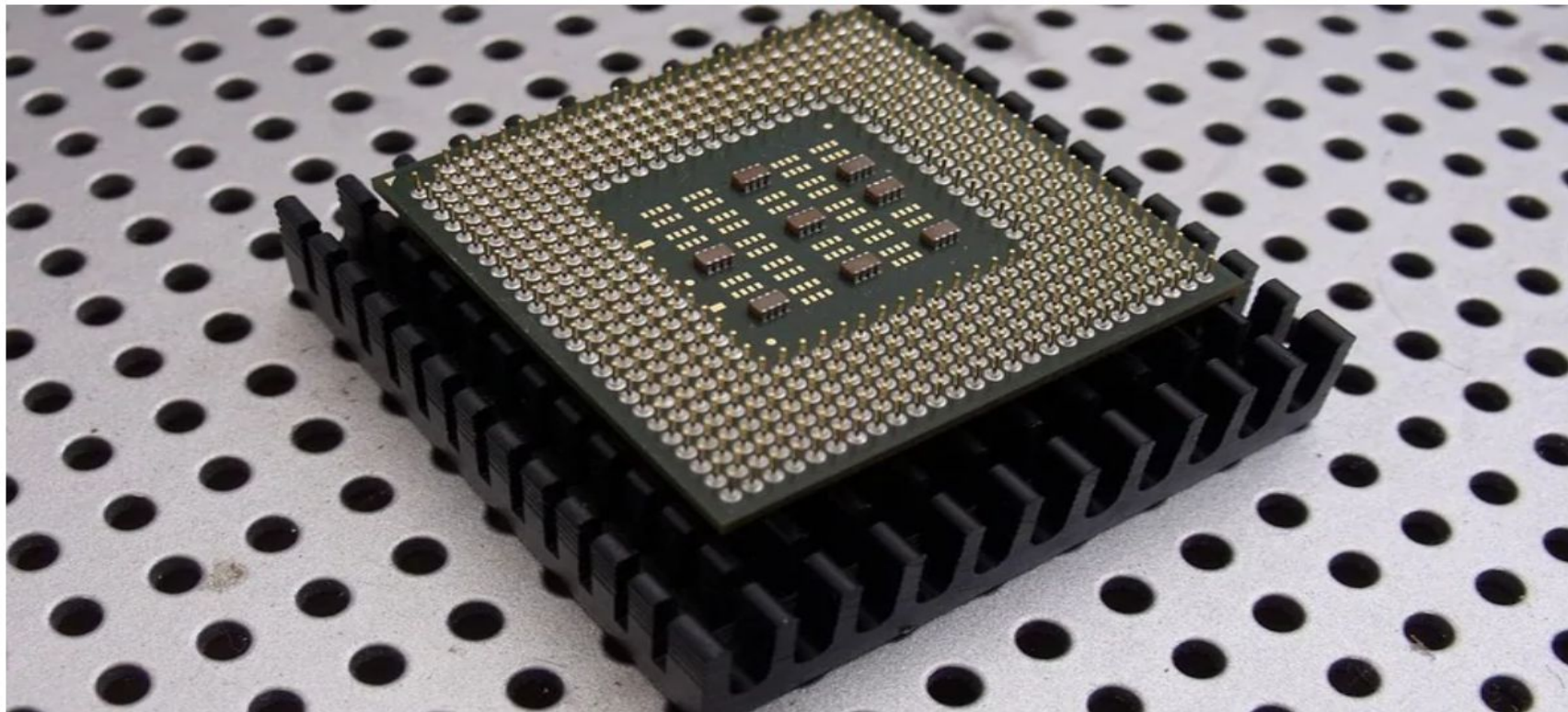
[Tweet](#)

[in Share](#)

[Email](#)

[Reddit](#)

[Comment](#)



An intriguing Example!

```
#include<stdio.h>

int main(){
    int x = 3000,y;
    x = x + 3;
    y = 100;
    return y;
}
```



```
main:
    pushq    %rbp
    movq     %rsp, %rbp
    movl     $3000, -8(%rbp)
    addl     $3, -8(%rbp)
    movl     $100, -4(%rbp)
    movl     -4(%rbp), %eax
    popq     %rbp
    ret
```

Some Basics

- % - indicates register names. Example : %rbp
- \$ - indicates constants Example : \$100
- Accessing register values:
 - %rbp : Access value stored in register rbp
 - (%rbp) : Treat value stored in register rbp as a pointer. Access the value stored at address pointed by the pointer. Basically *rbp
 - 4(%rbp) : Access value stored at address which is 4 bytes after the address stored in rbp. Basically *(rbp + 4)

An intriguing Example!

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

```
int main(){  
    int x = 3000,y;  
    x = x + 3;  
    y = 100;  
    return y;  
}
```



main:

```
pushq    %rbp  
movq     %rsp, %rbp  
movl     $3000, -8(%rbp)  
addl     $3, -8(%rbp)  
movl     $100, -4(%rbp)  
movl     -4(%rbp), %eax  
popq     %rbp  
ret
```

An intriguing Example!

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

```
int main(){  
    int x = 3000,y;  
    x = x + 3;  
    y = 100;  
    return y;  
}
```



```
main:  
    → pushq    %rbp  
      movq     %rsp, %rbp  
      movl     $3000, -8(%rbp)  
      addl     $3, -8(%rbp)  
      movl     $100, -4(%rbp)  
      movl     -4(%rbp), %eax  
      popq     %rbp  
      ret
```

For each function call, new space is created on the stack to store local variables and other data. This is known as a stack frame. To accomplish this, you will need to write some code at the beginning and end of each function to create and destroy the stack frame

An intriguing Example!

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

```
int main(){  
    int x = 3000,y;  
    x = x + 3;  
    y = 100;  
    return y;  
}
```



main:

```
pushq    %rbp  
→ movq    %rsp, %rbp  
movl     $3000, -8(%rbp)  
addl     $3, -8(%rbp)  
movl     $100, -4(%rbp)  
movl     -4(%rbp), %eax  
popq     %rbp  
ret
```

rbp is the frame pointer. In our code, it gets a snapshot of the stack pointer (**rsp**) so that when **rsp** is changed, local variables and function parameters are still accessible from a constant offset from **rbp**.

An intriguing Example!

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

```
int main(){  
    int x = 3000,y;  
    x = x + 3;  
    y = 100;  
    return y;  
}
```



```
main:  
    pushq    %rbp  
    movq     %rsp, %rbp  
    → movl    $3000, -8(%rbp)  
    addl     $3, -8(%rbp)  
    movl     $100, -4(%rbp)  
    movl     -4(%rbp), %eax  
    popq     %rbp  
    ret
```

move immediate value 3000 to (%rbp-8)

An intriguing Example!

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

```
int main(){  
    int x = 3000,y;  
    x = x + 3;  
    y = 100;  
    return y;  
}
```



main:

```
pushq    %rbp  
movq     %rsp, %rbp  
movl     $3000, -8(%rbp)  
→ addl     $3, -8(%rbp)  
movl     $100, -4(%rbp)  
movl     -4(%rbp), %eax  
popq     %rbp  
ret
```

add immediate value 3 to (%rbp-8)

An intriguing Example!

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

```
int main(){  
    int x = 3000,y;  
    x = x + 3;  
    y = 100;  
    return y;  
}
```



```
main:  
    pushq    %rbp  
    movq     %rsp, %rbp  
    movl     $3000, -8(%rbp)  
    addl     $3, -8(%rbp)  
    → movl     $100, -4(%rbp)  
    movl     -4(%rbp), %eax  
    popq     %rbp  
    ret
```

Move immediate value 100 to (%rbp-4)

An intriguing Example!

```
#include<stdio.h>

int main(){
    int x = 3000,y;
    x = x + 3;
    y = 100;
    return y;
}
```



```
main:
    pushq    %rbp
    movq     %rsp, %rbp
    movl     $3000, -8(%rbp)
    addl     $3, -8(%rbp)
    movl     $100, -4(%rbp)
    → movl    -4(%rbp), %eax
    popq     %rbp
    ret
```

Move (%rbp-4) to auxiliary register

An intriguing Example!

```
#include<stdio.h>

int main(){
    int x = 3000,y;
    x = x + 3;
    y = 100;
    return y;
}
```



```
main:
    pushq    %rbp
    movq     %rsp, %rbp
    movl     $3000, -8(%rbp)
    addl     $3, -8(%rbp)
    movl     $100, -4(%rbp)
    movl     -4(%rbp), %eax
    popq     %rbp
    ret
```

Pop the base pointer to restore state

An intriguing Example!

```
#include<stdio.h>

int main(){
    int x = 3000,y;
    x = x + 3;
    y = 100;
    return y;
}
```



```
main:
    pushq    %rbp
    movq     %rsp, %rbp
    movl     $3000, -8(%rbp)
    addl     $3, -8(%rbp)
    movl     $100, -4(%rbp)
    movl     -4(%rbp), %eax
    popq     %rbp
    ret
```

The calling convention dictates that a function's return value is stored in %eax, so the above instruction sets us up to return y at the end of our function.

Operation Suffixes

- b = byte (8 bit)
- s = single (32-bit floating point)
- w = word (16 bit)
- l = long (32 bit integer or 64-bit floating point)
- q = quad (64 bit)
- t = ten bytes (80-bit floating point)

How to get assembly code?

Two ways:

- While Compiling
 - Use **-S flag with gcc**. Will create a .s file containing assembly
- Using Binary
 - Use **objdump**. Will show the assembly in terminal.

Understanding the output

- The output will have assembly, but there is more information!
- You will see lots of Directives like:
 - .file
 - .text
 - .global name

Understanding the output

- The output will have assembly, but there is more information also!.
- You will see lots of Directives like:
 - .file
 - .text
 - .global name

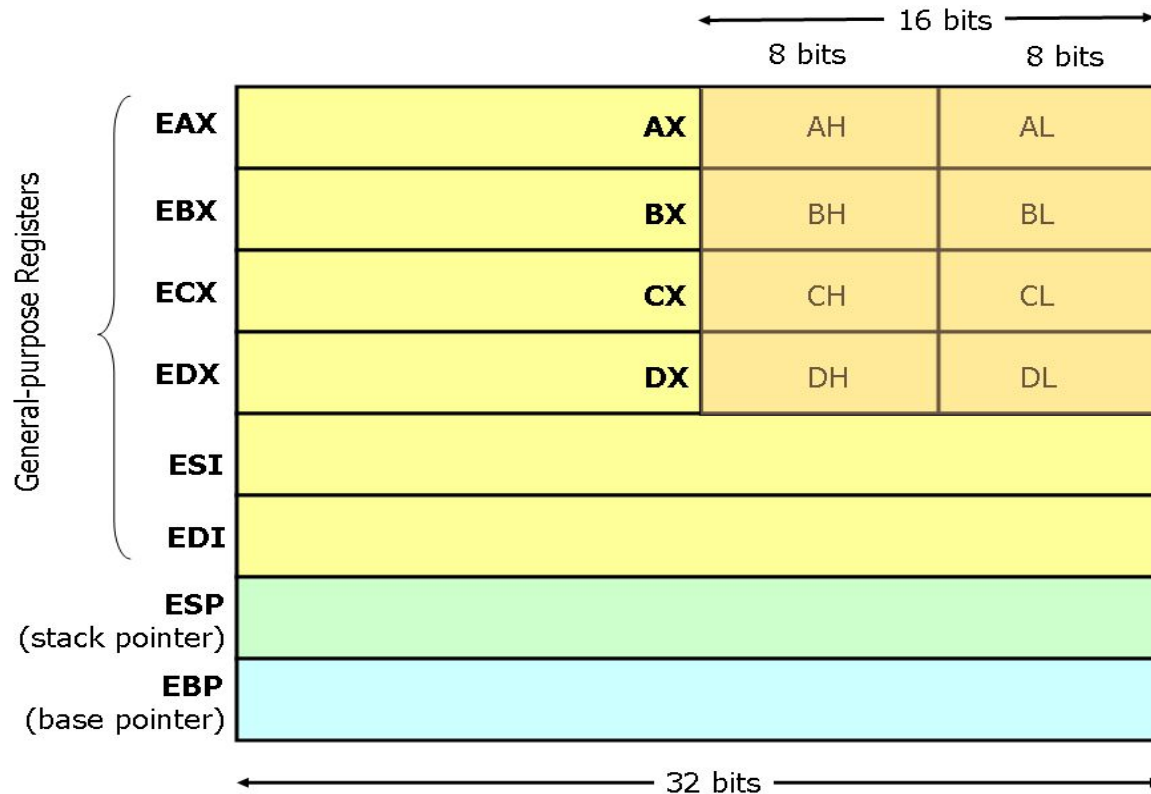
▲ To disable these, use the gcc option

137

```
-fno-asynchronous-unwind-tables
```

▼ Note, I know this is a really old thread, but this is the top result on google for `cfi_startproc`, so many people probably come here to disable that output.

x86 Register Set



x86 Register Set : A few more

- Registers starting with “r”
 - Same as “e” registers but 64 bits wide
- EIP : The Instruction Pointer or the Program Counter

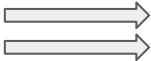
An Example with Loops!

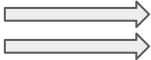
```
#include<stdio.h>
```

```
int main(){
    int x = 0;
    for(int i=0;i<10;i++){
        x = x + 1;
    }
    return x;
}
```

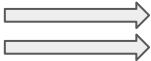
```
main:      pushq    %rbp
           movq    %rsp, %rbp
           movl    $0, -8(%rbp)
           movl    $0, -4(%rbp)
           jmp     .L2
.L3:       addl    $1, -8(%rbp)
           addl    $1, -4(%rbp)
.L2:       cmpl    $9, -4(%rbp)
           jle     .L3
           movl    -8(%rbp), %eax
           popq    %rbp
           ret
```

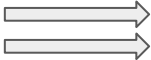

System Calls in Assembly

kernel:
 int 80h ; //Call kernel
ret

open:
push dword mode
push dword flags
push dword path
 mov eax, 5
call kernel
add esp, byte 12
ret

System Calls in Assembly

kernel:
 int 80h ; //Call kernel
ret

open:
push dword mode
push dword flags
push dword path
 mov eax, 5  Syscall Number
call kernel
add esp, byte 12
ret

A bit different!

A simple fork program



```
movl    $0, %eax  
call    printf  
call    fork
```

Embedding Assembly in C

`__asm__("instruction 1", "instruction 2", ...)`

Example:

```
__asm__(  
    "movl %edx, %eax\n\t"  
    "addl $2, %eax\n\t"  
);
```


Embedding Assembly in C

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x = 5;
    printf("x = %d\n", x);
    __asm__("add $10, %0" : "=r"(x));
    printf("x = %d\n", x);
    return 0;
}
```



```
x = 5
x = 15
```

Where will I use assembly?



Where will I use assembly?

- To write Compilers and Device Drivers
- To write viruses and for malware analysis
- Used while programming Real Time Embedded systems
- Implementing Locks for Concurrency.
We will cover this in the third module of the course!

References

- Chapter 11. x86 Assembly Language Programming, FreeBSD, https://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/x86.html
- Easy x86-64, http://ian.seyler.me/easy_x86-64/
- Introduction to the GNU/Linux assembler and linker for Intel Pentium processors, <https://www.cs.usfca.edu/~cruse/cs210s07/lesson01.ppt>
- Is there a way to insert assembly code into C?, <https://stackoverflow.com/questions/61341/is-there-a-way-to-insert-assembly-code-int-o-c>

More Topics

- GCC
- Clang
- GCC vs Clang
 - (More: <https://opensource.apple.com/source/clang/clang-23/clang/tools/clang/www/comparison.html>)
- Debugger
- How to use a debugger (the practical way)

GCC

- The GNU Compiler Collection (GCC) is a compiler system produced by the GNU Project supporting various programming languages.
- GCC is a key component of the GNU toolchain and the standard compiler for most projects related to GNU and Linux, including the Linux kernel

Clang

- Clang is a compiler front end for the C, C++, Objective-C and Objective-C++ programming languages, as well as the OpenMP, OpenCL, RenderScript, CUDA ...
- It uses the LLVM compiler infrastructure as its back end and has been part of the LLVM release cycle since LLVM 2.6

LLVM

- The LLVM compiler infrastructure project is a set of compiler and toolchain technologies, which can be used to develop a front end for any programming language and a back end for any instruction set architecture.
- LLVM is designed around a language-independent intermediate representation (IR) that serves as a portable, high-level assembly language that can be optimized with a variety of transformations over multiple passes.

Clang vs GCC [1/4]

Benefits of Using GCC

- supports languages that clang does not aim to, such as Java, Ada, FORTRAN, etc.
- front-ends are very mature and already support C++. clang's support for C++ is nowhere near what GCC supports.
- supports more targets than LLVM.
- popular and widely adopted.
- GCC does not require a C++ compiler to build it.

Clang vs GCC [1/4]

Benefits of Clang

- The Clang ASTs and design are intended to be easily understandable by anyone who is familiar with the languages involved and who has a basic understanding of how a compiler works.
- Clang is designed as an API from its inception, allowing it to be reused by source analysis tools, refactoring, IDEs (etc) as well as for code generation

Clang vs GCC [1/4]

Benefits of Clang

- Clang aims to provide extremely clear and concise diagnostics (error and warning messages), and includes support for expressive diagnostics.
- Clang inherits a number of features from its use of LLVM as a backend, including support for a bytecode representation for intermediate code, pluggable optimizers, link-time optimization support, Just-In-Time compilation, ability to link in multiple code generators, etc.

Clang vs GCC [1/4]

```
glint@Mark4: ~  
GNU nano 4.8  
#include <bits/stdc++.h>  
  
using namespace std  
  
int main(){  
  
    cout<<"Hello World"<<endl;  
  
}
```

```
glint@Mark4:~$ gcc main.cpp  
main.cpp:3:20: error: expected ';' before 'int'  
  3 | using namespace std  
    |                      ^  
    |                      ;  
  4 |  
  5 | int main(){  
    | ~~~~  
glint@Mark4:~$ clang main.cpp  
main.cpp:3:20: error: expected ';' after namespace name  
using namespace std  
    |                      ^  
    |                      ;  
1 error generated.  
glint@Mark4:~$
```

Debugger

- A debugger or debugging tool is a computer program used to test and debug other programs (the "target" program).
- The main use of a debugger is to run the target program under controlled conditions
 - that permit the programmer to track its operations in progress
 - monitor changes in computer resources
 - most often memory areas used by the target program or the computer's operating system that may indicate malfunctioning code.

GNU GDB

- The GNU Debugger (GDB) is a portable debugger that runs on many Unix-like systems and works for many programming languages, including Ada, C, C++, Objective-C, Free Pascal, Fortran, Go ...

How to use GNU GDB (the impractical basics)

- <https://u.osu.edu/cstutorials/2018/09/28/how-to-debug-c-program-using-gdb-in-6-simple-steps/>
- Might end up as interview questions!

How to use GNU GDB (the practical basics)

- Use an IDE or an advanced text editor with support for debugging
- Something that works on almost all OS and is open source and user extensible is Visual Studio Code.
 - Requirements
 - The source code Eg. main.cpp
 - A compiler Eg gcc
 - A debugger like gdb
 - An editor with support for debugging like Visual Studio Code.

EXPLORER

> OPEN EDITORS

DEBUG [WSL: UBUNTU-20...

a.out

main.c

main.c

main.c > main()

1 #include <stdio.h>

2

3 int main()

4 {

5 int i, num, j;

6 printf ("Enter the number: ");

7 scanf ("%d", &num);

8

9 for (i=1; i<num; i++)

10 j=j*i;

11

12 printf("The factorial of %d is %d\n",num,j);

13 }

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

1: bash

glint@Mark4:/mnt/d/ws/OS/guestLect/debug\$ gcc main.c

glint@Mark4:/mnt/d/ws/OS/guestLect/debug\$./a.out

Enter the number: 4

The factorial of 4 is 196596

glint@Mark4:/mnt/d/ws/OS/guestLect/debug\$

OUTLINE

TIMELINE

WSL: Ubuntu-20.04 0 0 0

Ln 13, Col 2 Tab Size: 4 UTF-8 LF C Linux



EXPLORER

main.c

> OPEN EDITORS

DEBUG [WSL: UBUNTU-20...

a.out

main.c

```
main.c > main()
1  #include <stdio.h>
2
3  int main()
4  {
5      int i, num, j;
6      printf("Enter the number: ");
7      scanf ("%d", &num );
8
9      for (i=1; i<num; i++)
10         j=j*i;
11
12     printf("The factorial of %d is %d\n",num,j);
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

1: bash

```
glint@Mark4:/mnt/d/ws/OS/guestLect/debug$ gcc main.c
glint@Mark4:/mnt/d/ws/OS/guestLect/debug$ ./a.out
Enter the number: 4
The factorial of 4 is 196596
glint@Mark4:/mnt/d/ws/OS/guestLect/debug$
```

> OUTLINE

> TIMELINE



EXPLORER



C main.c X

> OPEN EDITORS

v DEBUG [WSL: UBUNTU-20...

a.out

C main.c

```
C main.c > main()
1  #include <stdio.h>
2
3  int main()
4  {
5      int i, num, j;
6      printf ("Enter the number: ");
7      scanf ("%d", &num );
8
9      for (i=1; i<num; i++)
10         j=j*i;
11
12     printf("The factorial of %d is %d\n",num,j);
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
glint@Mark4:/mnt/d/ws/OS/guestLect/debug$ gcc main.c
glint@Mark4:/mnt/d/ws/OS/guestLect/debug$ ./a.out
Enter the number: 4
The factorial of 4 is 196596
glint@Mark4:/mnt/d/ws/OS/guestLect/debug$
```

1: bash



> OUTLINE

> TIMELINE



EXPLORER



C main.c X

> OPEN EDITORS

v DEBUG [WSL: UBUNTU-20...

a.out

C main.c

```
C main.c > main()
1  #include <stdio.h>
2
3  int main()
4  {
5      int i, num, j;
6      printf ("Enter the number: ");
7      scanf ("%d", &num );
8
9      for (i=1; i<num; i++)
10         j=j*i;
11
12     printf("The factorial of %d is %d\n",num,j);
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

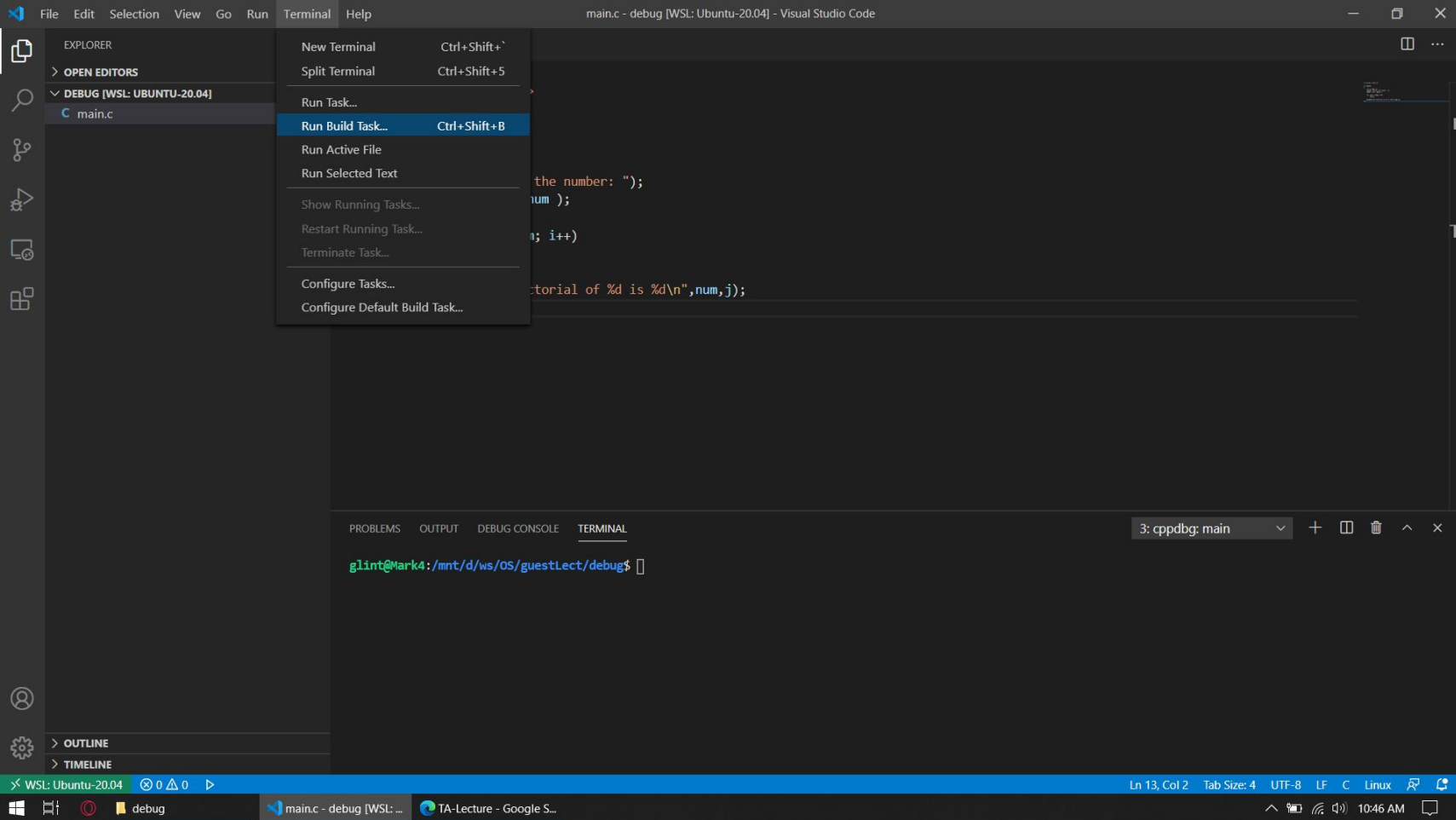
```
glint@Mark4:/mnt/d/ws/OS/guestLect/debug$ gcc main.c
glint@Mark4:/mnt/d/ws/OS/guestLect/debug$ ./a.out
Enter the number: 4
The factorial of 4 is 196596
glint@Mark4:/mnt/d/ws/OS/guestLect/debug$
```

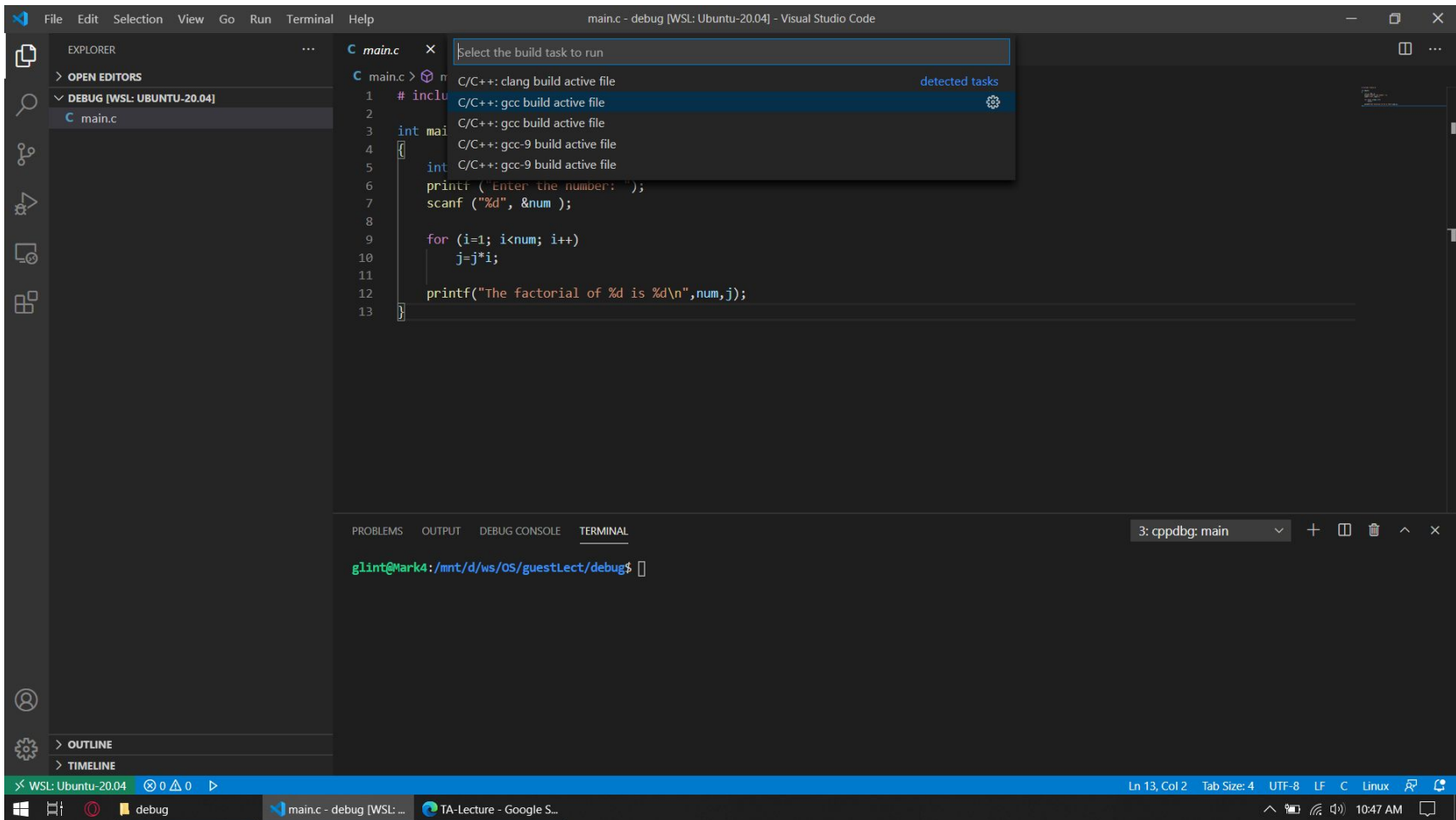
1: bash



> OUTLINE

> TIMELINE





Visual Studio Code interface showing the Explorer, Run and Debug, and Terminal panels.

Explorer: main.c

Run and Debug: Start Debugging (F5), Run Without Debugging (Ctrl+F5), Stop Debugging (Shift+F5), Restart Debugging (Ctrl+Shift+F5), Open Configurations, Add Configuration..., Step Over (F10), Step Into (F11), Step Out (Shift+F11), Continue (F5), Toggle Breakpoint (F9), New Breakpoint, Enable All Breakpoints, Disable All Breakpoints, Remove All Breakpoints, Install Additional Debuggers...

Terminal: 2: Task - gcc build activ...
Terminal will be reused by tasks, press any key to close it.
> Executing task: /usr/bin/gcc -g /mnt/d/ws/OS/guestLect/debug/main.c -o /mnt/d/ws/OS/guestLect/debug/main <
Terminal will be reused by tasks, press any key to close it.
> Executing task: /usr/bin/gcc -g /mnt/d/ws/OS/guestLect/debug/main.c -o /mnt/d/ws/OS/guestLect/debug/main <
Terminal will be reused by tasks, press any key to close it.

Code Editor: main.c - debug [WSL: Ubuntu-20.04] - Visual Studio Code

```
h>  
  
j;  
er the number: ");  
&num );  
  
num; i++)  
  
Factorial of %d is %d\n",num,j));
```

main.c - debug [WSL: Ubuntu-20.04] - Visual Studio Code

File Edit Selection View Go Run Terminal Help

EXPLORER

OPEN EDITORS

DEBUG [WSL: UBUNTU-20.04]

main

main.c

main.c

Select Environment

- C++ (GDB/LLDB)
- C++ (Windows)
- More...

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int i, num, j;
6     printf ("Enter the number: ");
7     scanf ("%d", &num );
8
9     for (i=1; i<num; i++)
10         j=j*i;
11
12     printf("The factorial of %d is %d\n",num,j);
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

2: Task - gcc build activ

Terminal will be reused by tasks, press any key to close it.

> Executing task: /usr/bin/gcc -g /mnt/d/ws/OS/guestLect/debug/main.c -o /mnt/d/ws/OS/guestLect/debug/main <

Terminal will be reused by tasks, press any key to close it.

> Executing task: /usr/bin/gcc -g /mnt/d/ws/OS/guestLect/debug/main.c -o /mnt/d/ws/OS/guestLect/debug/main <

Terminal will be reused by tasks, press any key to close it.

WSL: Ubuntu-20.04 0 0 0

debug

main.c - debug [WSL: ...]

TA-Lecture - Google S...

Ln 13, Col 2 Tab Size: 4 UTF-8 LF C Linux

10:47 AM

main.c - debug [WSL: Ubuntu-20.04] - Visual Studio Code

File Edit Selection View Go Run Terminal Help

EXPLORER

OPEN EDITORS

DEBUG [WSL: UBUNTU-20.04]

- main
- main.c

main.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int num;
6     printf("Enter a number: ");
7     scanf("%d", &num);
8
9     for (i=1; i<num; i++)
10         j=j*i;
11
12     printf("The factorial of %d is %d\n", num, j);
13 }
```

Select a configuration

- gcc - Build and debug active file
- gcc-9 - Build and debug active file
- gcc - Build and debug active file
- gcc-9 - Build and debug active file
- clang - Build and debug active file
- Default Configuration

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

2: Task - gcc build activ

Terminal will be reused by tasks, press any key to close it.

> Executing task: /usr/bin/gcc -g /mnt/d/ws/OS/guestLect/debug/main.c -o /mnt/d/ws/OS/guestLect/debug/main <

Terminal will be reused by tasks, press any key to close it.

> Executing task: /usr/bin/gcc -g /mnt/d/ws/OS/guestLect/debug/main.c -o /mnt/d/ws/OS/guestLect/debug/main <

Terminal will be reused by tasks, press any key to close it.

WSL: Ubuntu-20.04 0 0 0

main.c - debug [WSL: ...] TA-Lecture - Google S...

Ln 13, Col 2 Tab Size: 4 UTF-8 LF C Linux 10:47 AM

File Edit Selection View Go Run Terminal Help launch.json - debug [WSL: Ubuntu-20.04] - Visual Studio Code

EXPLORER

OPEN EDITORS

DEBUG [WSL: UBUNTU-20.04]

- .vscode
- launch.json
- main
- main.c

main.c

```
2
3
4
5
6
7
8      "name": "gcc - Build and debug active file",
9      "type": "cppdbg",
10     "request": "launch",
11     "program": "${fileDirname}/${fileBasenameNoExtension}",
12     "args": [],
13     "stopAtEntry": false,
14     "cwd": "${workspaceFolder}",
15     "environment": [],
16     "externalConsole": false,
17     "MIMode": "gdb",
18     "setupCommands": [
19         {
20             "description": "Enable pretty-printing for gdb",
21             "text": "-enable-pretty-printing",
22             "ignoreFailures": true
23         }
24     ],
25     "preLaunchTask": "C/C++: gcc build active file",
26     "miDebuggerPath": "/usr/bin/gdb"
27 }
28
29 }
```

Add Configuration...

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

2: Task - gcc build activ

Terminal will be reused by tasks, press any key to close it.

> Executing task: /usr/bin/gcc -g /mnt/d/ws/OS/guestLect/debug/main.c -o /mnt/d/ws/OS/guestLect/debug/main <

Terminal will be reused by tasks, press any key to close it.

> Executing task: /usr/bin/gcc -g /mnt/d/ws/OS/guestLect/debug/main.c -o /mnt/d/ws/OS/guestLect/debug/main <

Terminal will be reused by tasks, press any key to close it.

OUTLINE

TIMELINE

WSL: Ubuntu-20.04 0 0 0

Ln 22, Col 43 Spaces: 4 UTF-8 LF JSON with Comments

launch.json - debug [WSL: Ubuntu-20.04] TA-Lecture - Google S...

10:47 AM

main.c - debug [WSL: Ubuntu-20.04] - Visual Studio Code

EXPLORER

OPEN EDITORS

DEBUG [WSL: UBUNTU-20.04]

- .vscode
- launch.json
- main
- main.c**

main.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int i, num, j;
6     printf ("Enter the number: ");
7     scanf ("%d", &num );
8
9     for (i=1; i<num; i++)
10         j=j*i;
11
12     printf("The factorial of %d is %d\n",num,j);
13 }
```

Breakpoint

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

2: Task - gcc build activ

Terminal will be reused by tasks, press any key to close it.

> Executing task: /usr/bin/gcc -g /mnt/d/ws/OS/guestLect/debug/main.c -o /mnt/d/ws/OS/guestLect/debug/main <

Terminal will be reused by tasks, press any key to close it.

> Executing task: /usr/bin/gcc -g /mnt/d/ws/OS/guestLect/debug/main.c -o /mnt/d/ws/OS/guestLect/debug/main <

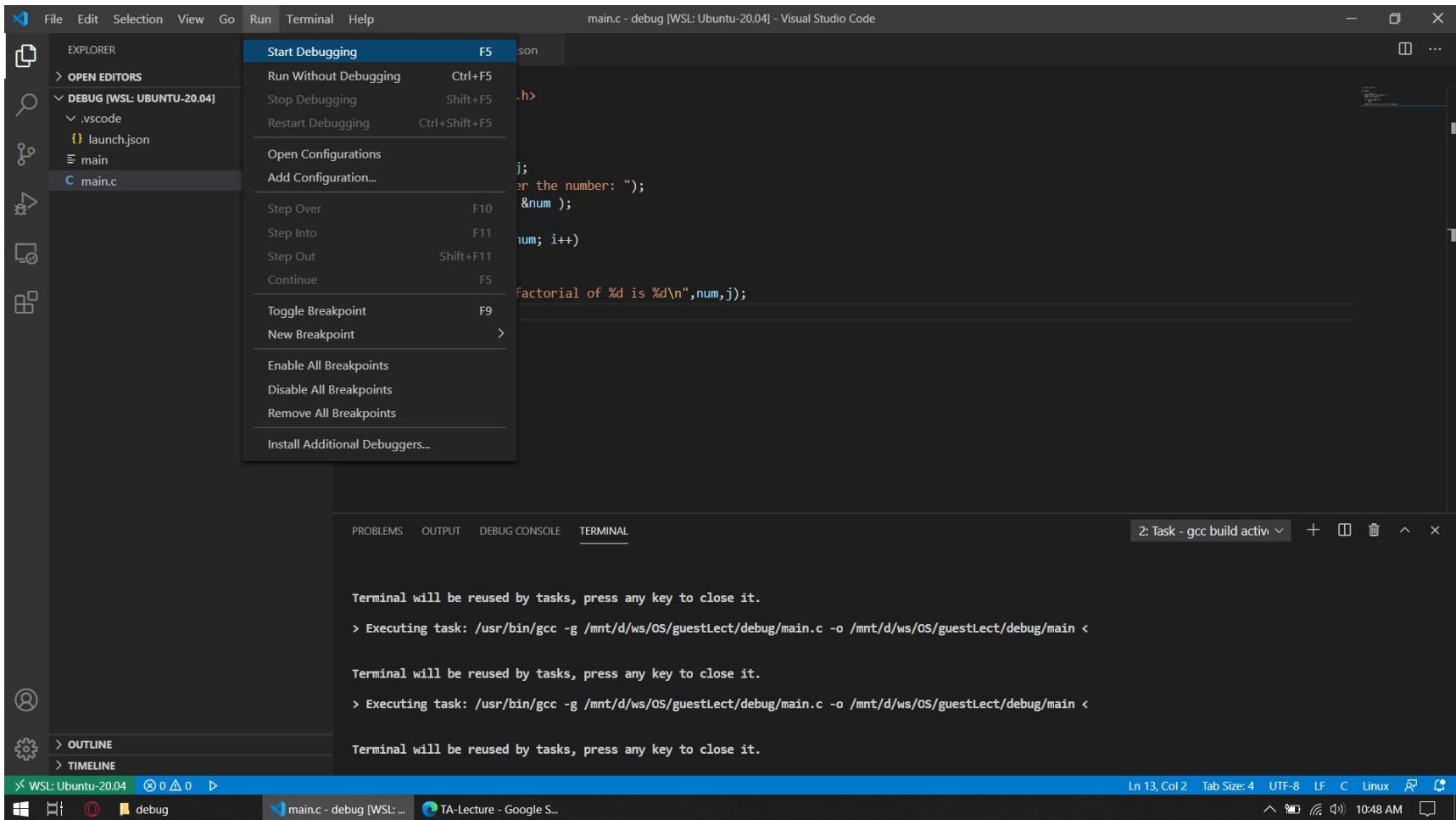
Terminal will be reused by tasks, press any key to close it.

WSL: Ubuntu-20.04 0 0 0

main.c - debug [WSL: ...] TA-Lecture - Google S...

Ln 13, Col 2 Tab Size: 4 UTF-8 LF C Linux

10:48 AM



Visual Studio Code interface showing a C program being debugged in WSL (Ubuntu-20.04).

File Explorer (Left):

- VARIABLES
 - Locals
 - i: -8032
 - num: 21845
 - j: 32767
- WATCH
- CALL STACK
 - main() (main.c:6:1) - PAUSED ON BREAKPOINT
- BREAKPOINTS
 - main.c:6

Editor (Center):

```
main.c > main()
1  #include <stdio.h>
2
3  int main()
4  {
5      int i, num, j;
6      printf ("Enter the number: ");
7      scanf ("%d", &num );
8
9      for (i=1; i<num; i++)
10         j=j*i;
11
12     printf("The factorial of %d is %d\n",num,j);
13 }
```

Terminal (Bottom):

3: cppdbg: main

Status Bar (Bottom):

WSL: Ubuntu-20.04 | gcc - Build and debug active file (debug) | Ln 6, Col 1 | Tab Size: 4 | UTF-8 | LF | C | Linux | 10:48 AM