

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
#define MAXPAROLA 30
#define MAXRIGA 80
```

```
int main(int argc, char *argv[])
{
    int freq[MAXPAROLA]; /* vettore di contatori
delle frequenze delle lunghezze delle parole */
    char riga[MAXRIGA];
    int i, inizio, lunghezza;
    FILE *f;
```

```
for(i=0; i<MAXPAROLA; i++)
    freq[i]=0;
```

```
if(argc != 2)
```

```
{
    printf(stderr, "ERRORE, serve un parametro con il nome del file\n");
    exit(1);
}
```

```
f = fopen(argv[1], "r");
if(f==NULL)
```

```
{
    printf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
    exit(1);
}
```

```
while( fgets( riga, MAXRIGA, f ) != NULL )
```



L'ambiente UNIX/Linux

Strumenti per la programmazione C

Stefano Quer

Dipartimento di Automatica e Informatica

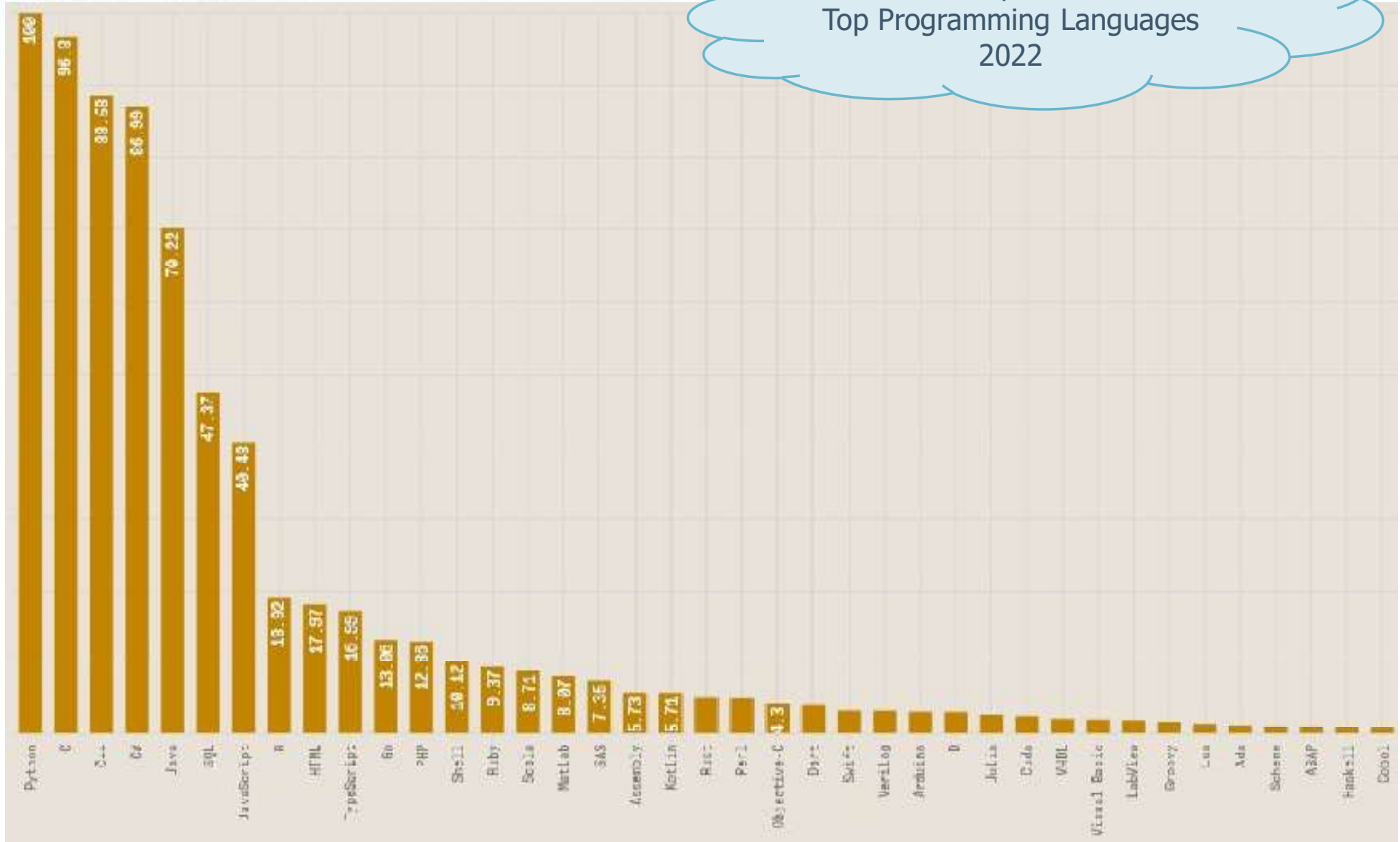
Politecnico di Torino

Linguaggi di programmazione

- ❖ Esistono più di 200 linguaggi di programmazione
 - Molti non sono praticamente utilizzati
 - Altri sono stati utilizzati soprattutto in passato
- ❖ Esistono diverse tecniche di classificazione
 - Più usato, più amato, più pagato, etc.

Linguaggi di programmazione

IEEE Spectrum
Top Programming Languages
2022



Linguaggi di programmazione

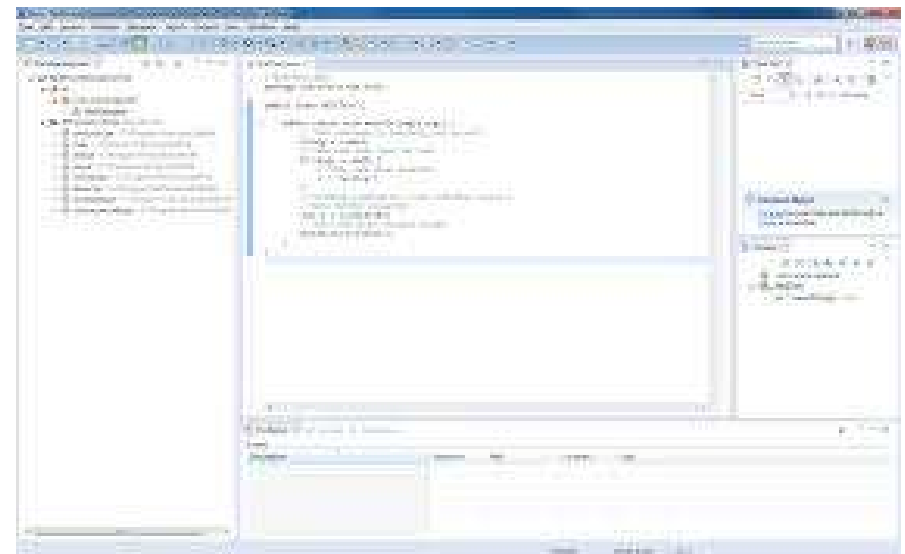
PYPL Index e Stack Overflow
Settembre 2022

Position	PYPL ranking September 2022	Stack Overflow's Developer Survey 2022
#1	Python	JavaScript
#2	Java	HTML/CSS
#3	JavaScript	SQL
#4	C#	Python
#5	C/C++	TypeScript
#6	PHP	Java
#7	R	Bash/Shell
#8	TypeScript	C#
#9	Go	C++
#10	Swift	PHP

❖ Integrated Development Environment (IDE)

- Piattaforma unica per sviluppare progetti in linguaggi diversi
- In genere forniscono
 - Text editor, syntax highlighter, customizable interfaces, compiler, code auto-save, version control, debugger, build automation, and deployment.

<https://www.geeksforgeeks.org/7-best-ides-for-c-c-plus-plus-developers-in-2022/>
Settembre 2022

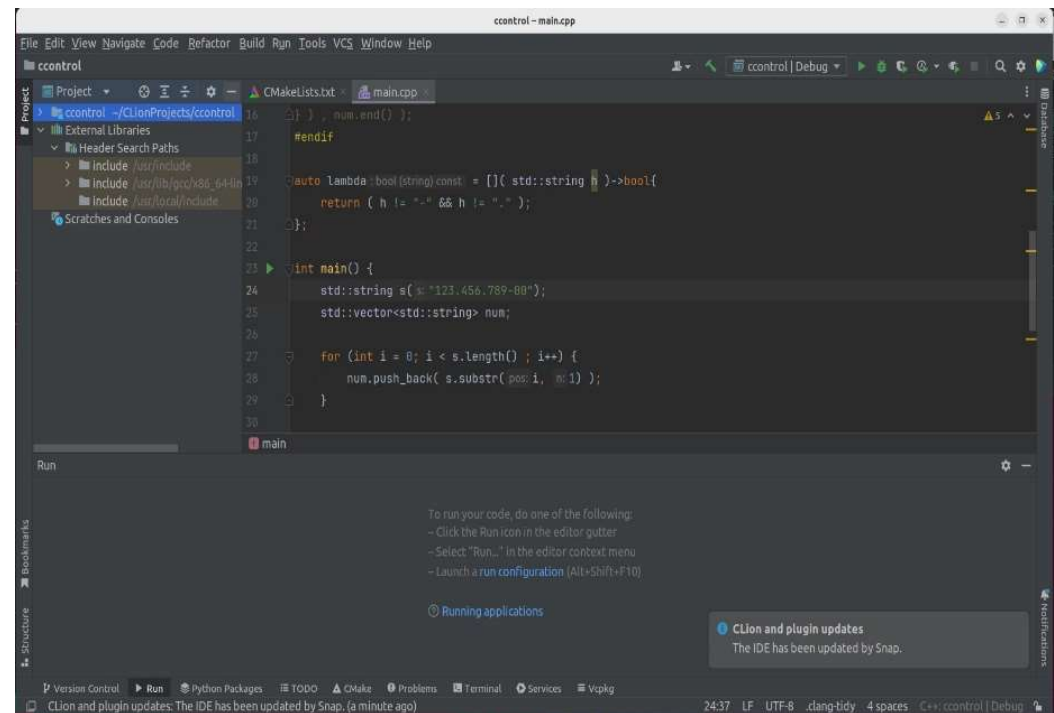


IDE

Tool	Commenti
Eclipse	Scritto in JAVA e sviluppato da IBM. Supporta C, C++, C#, Java, Javascript, COBOL; Perl, Python, etc.
Visual Studio	Scritto in C++ e sviluppato da Microsoft. Supporta C, C++, C#, CSS, Go, HTML, Java, JavaScript, Python, PHP, TypeScript e altri.
NetBeans	Free open source, sviluppato da Apache Software Foundation. Raccomandato per principianti e C/C++.
CLion	Sviluppato da JetBrains per programmatori C++, migliore piattaforma cross-platform (Mac OS, Linux, Windows integrate con Cmake); supporta Kotlin, Python, Swift, etc.
Code::Blocks	Open-source C/C++ IDE sviluppato utilizzando wxWidgets, un toolkit GUI; supporta C, C++ e Fortran.
CodeLite	Free e open-source per C++; uno dei migliori IDE per code refactoring; supportato da Windows e Mac.
QtCreator	Richiede una licenza commerciale nella versione completa; supportato da Windows, Linux, and Mac OS.

IDE: CLion

- ❖ JetBrains sviluppa tanto CLion quanto PyCharm
 - Clion permette lo sviluppo di programmi C e C++
 - Basato sul modello di progetto
 - Supporta CMake e Makefile
 - Integragato con il debugger (GDB e LLDB)

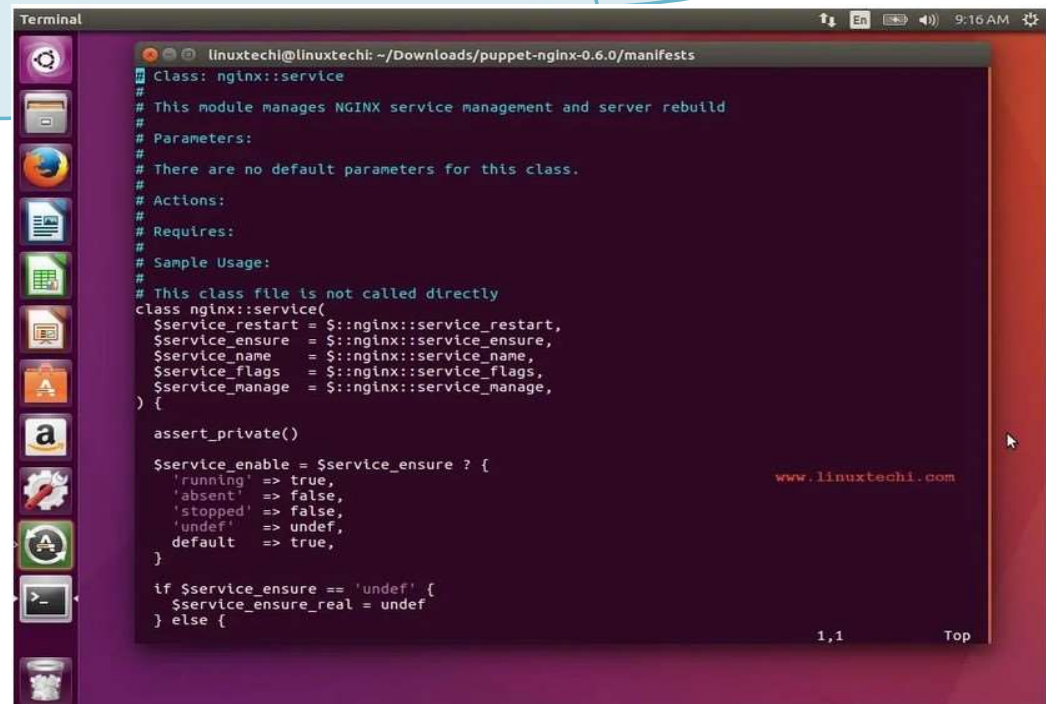


Editor

- ❖ Un editor permette di scrivere file di testo (eventualmente contenente programmi)

<https://www.linuxtechi.com/top-10-text-editors-for-linux-desktop/>

Settembre 2022



The screenshot shows a Linux desktop with a terminal window open. The terminal displays the contents of a Puppet manifest file located at `~/Downloads/puppet-nginx-0.6.0/manifests`. The file defines a class `nginx::service` for managing the NGINX service. The class includes parameters, actions, requirements, and sample usage. The class is not called directly but is used within a `class nginx::service` block. The terminal also shows the `assert_private()` function and the `$service_enable` variable. The desktop environment includes a sidebar with various application icons and a top status bar showing the time as 9:16 AM.

```
Terminal
linuxtechi@linuxtechi: ~/Downloads/puppet-nginx-0.6.0/manifests
# Class: nginx::service
# This module manages NGINX service management and server rebuild
# Parameters:
# There are no default parameters for this class.
# Actions:
# Requires:
# Sample Usage:
# This class file is not called directly
class nginx::service {
  $service_restart = $::nginx::service_restart,
  $service_ensure = $::nginx::service_ensure,
  $service_name = $::nginx::service_name,
  $service_flags = $::nginx::service_flags,
  $service_manage = $::nginx::service_manage,
} {
  assert_private()

  $service_enable = $service_ensure ? {
    'running' => true,
    'absent' => false,
    'stopped' => false,
    'undef' => undef,
    default => true,
  }

  if $service_ensure == 'undef' {
    $service_ensure_real = undef
  } else {
    1,1 Top
  }
}
```


Editor

Tool	Commenti
VIM	VI Improved. Default editor UNIX/Linux; presente ovunque. Ostico ma ampiamente configurabile.
Geany	Editor per LINUX desktop integrabile con il tool di sviluppo GTK+.
Sublime	Editor di testo e ambiente di sviluppo. Supporta diversi linguaggi (markup automatico)
Brackets	Prodotto da ADOBE nel 2014 per Linux. Sviluppato con HTML; CSS e JavaScript. Leggero.
Gedit	Default editor per il desktop GNOME. Semplice interfaccia utente. Leggero.
VS Code	Microsoft, per Windows, UNIX/Linux, Mac.
Nano	Simile all'editor Pico, rilasciato nel 2000, ma con diverse funzionalità aggiuntive. Permette solo "line interface".
Emacs	Uno degli editor per Linux più vecchi; sviluppato da Richard Stallmann, fondatore di GNU. Sviluppato interamente in LISP e C.

Editor: vi

❖ Editor di testo

- Presente in tutti i sistemi BSD e Unix
- Sviluppato a partire dal 1976
- Ultima versione (8.1) del 2018

❖ Versione base

- VI = “Visual in ex”
 - Commuta l’editor di linea ex in modalità visuale
- Non molto funzionale per operazioni estese
- Utile in caso di problemi con altri editor (e.g., editing remoto)

Editor: vi

- ❖ Nel tempo è stato ampliato e migliorato
 - VIM “VI Improved”
 - Estensione per editing di progetti complessi
 - Multi-level undo, multi-window, multi-buffer, etc.
 - On-line help, syntax hightlighting, etc.
- ❖ Insieme a emacs è uno dei protagonisti della “guerra degli editor”
- ❖ Estensioni
 - Permettono di incrementare le feature dell’editor
 - Bvi (Binary VI), Vigor (VI con Vigor Assistant), VILE (VI Like Emacs)

Editor: vi

- ❖ Si invoca con il comando
 - `vi nomeFile`
- ❖ Prevede diverse modalità operative
 - **Command Mode**
 - Cursore posizionato nel testo
 - La tastiera è utilizzata per impartire comandi
 - **Input Mode**
 - Modalità di inserzione testo
 - La tastiera è utilizzata per inserire testo
 - **Directive Mode**
 - Cursore posizionato sull'ultima riga del video
 - La tastiera è utilizzata per direttive di controllo

Editor: vi

Documentazione
man vim

<http://www.vim.org/docs.php>

<ftp://ftp.vim.org/pub/vim/doc/book/vimbook-OPL.pdf>

Command Mode	Comando
Spostamento cursore	←, ↑, →, ↓ (h, j, k, l)
Modalità inserimento (dal cursore)	i
Modalità inserimento (a inizio linea)	I
Modalità append (dal cursore)	a
Modalità append (fine linea)	A
Modalità sovrascrittura	R
Passa in modalità comandi	esc
Cancella una riga	dd
Cancella un carattere singolo	x

Anche
0-g
n-g

Anche
n-dd
n-x

Editor: vi

Command Mode (continua)	Comando
Inserisce ultima cancellazione	P
Annulla l'ultima operazione (undo)	U
Ripristina l'ultima modifica (redo)	Ctrl-r

Directive Mode	Comando
Passa in modalità direttive (ultima riga)	:
Mostra numeri di riga	:set num
Salva il file	:w!, :w filename
Esce senza salvare le ultime modifiche	:q!
Entra nell'help on-line	:help

Imparare VIM (da Google): [VIM Adventures](#)

Editor: emacs

- ❖ Editor di testo libero
 - Emacs = Editor MACroS
 - Sviluppato a partire dal 1976 da Richard Stallman
 - Ultima versione 29.1 (luglio 2023)
- ❖ Molto popolare tra i programmatori avanzati, potente, estendibile e versatile
- ❖ Ne esistono diverse versioni ma le più popolari sono la
 - GNU Emacs
 - Xemacs = next generation EMACS

Editor: emacs

❖ Disponibilile per

- GNU, GNU/Linux, FreeBSDm NetBSD, OpenBSD, Mac OS X, MS Windows, Solaris

❖ Vantaggi

- Funzionalità oltre il semplice editor di testi
- Completamente customizzabile
- Esecuzione veloce anche per operazioni complesse

❖ Svantaggi

- Curva di apprendimento lenta
- Scritto in Lisp

Editor: VS Code

- ❖ Visual Studio Code (VS Code) è un editor sviluppato da Microsoft
 - Sviluppato a partire dal 2015
 - Ultima versione 1.83.0 (ottobre 2023)
 - Disponibile gratuitamente per Windows, Linux, macOS ma con licenza proprietaria
 - Include supporto per debugging, controllo Git integrato, syntax highlighting, refactoring del codice, etc.
 - Altamente personalizzabile

Editor: VS Code

- Può essere utilizzato con vari linguaggi
 - C, C++, C#, HTML, PHP; Java, Ruby, etc.
- Molte funzionalità sono accessibili non tramite menù ma file .json

The screenshot shows the Visual Studio Code interface. The Explorer panel on the left shows a project named 'PROVA_GMAKE' with files '.vscode', 'build', 'CMakeLists.txt', and 'main.c'. The main editor window displays the content of 'main.c', which is a C program for a simple calculator. The program includes `<stdio.h>`, defines `main(void)`, and uses `scanf` to read an operation and two operands. It then performs arithmetic based on the operation and prints the result or an error message for division by zero. The Output panel at the bottom shows the build process, indicating that the project was built successfully using CMake and GCC.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     char operation;
6     float leftOperand, rightOperand, result;
7     int errorFlag=0;
8
9     printf("Ready (format: <operation> <operand1> <operand2>):\n");
10    scanf("%c", &operation);
11    scanf("%f", &leftOperand);
12    scanf("%f", &rightOperand);
13
14    if (operation == '+') {
15        result = leftOperand + rightOperand;
16    } else if (operation == '-') {
17        result = leftOperand - rightOperand;
18    } else if (operation == '*') {
19        result = leftOperand * rightOperand;
20    } else if (operation == '/') {
21        if (rightOperand == 0) {
22            printf("Error: division by zero. Aborting...\n");
23            return 1;
24        }
25        result = leftOperand / rightOperand;
26    }
27    printf("Result: %f\n", result);
28    return 0;
29 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS CMake/Build

```
[main] Building folder: PROVA_GMAKE
[build] Starting build
[proc] Executing command: /usr/bin/cmake --build /home/quer/current-B/giSquillero/SRC/PROVA_GMAKE/build --config Debug
--target all -j 12 --
[build] Consolidate compiler generated dependencies of target myapp
[build] [100%] Built target myapp
[driver] Build completed: 00:00:09.053
[build] Build finished with exit code 0
```

Compiler

❖ Compilatori

➤ GCC e G++

- Includono compiler e linker
- Supportano C e C++

❖ Utility di automatizzazione per la creazione di file eseguibili

- Makefile
- Gmake

Documentazione
man gcc

<http://www.gnu.org>

Compiler: gcc

```
gcc <opzioni> <argomenti>
```

❖ Comando di compilazione e linker generico

➤ Opzioni

- Elenco di flag che controllano il compilatore e il linker; ci sono opzioni per la sola compilazione, per il solo linker o per entrambi

➤ Argomenti

- Elenco di file che gcc legge e trasforma in maniera dipendente dalle opzioni

Compiler: gcc

- ❖ Compilazione di singoli file e poi link dei file oggetto in un unico eseguibile

```
gcc -c file1.c  
gcc -c file2.c  
gcc -c main.c
```

```
gcc -o myexe file1.o file2.o main.o
```

Run: myexe versus ./myexe
→ echo \$PATH
→ PATH=\$PATH:./
→ echo \$PATH

- ❖ Contestuale compilazione di diversi file sorgente, link e creazione dell'eseguibile

```
gcc -o myexe file1.c file2.c main.c
```

Compiler: gcc

Opzioni

Formato		Significato	Effetto
Compatto	Esteso		
-c file			Esegue la compilazione non il linker
-o file			Specifica il nome di output; in genere indica il nome dell'eseguibile finale (linkando)
-g			Indica a gcc di non ottimizzare il codice e di inserire informazioni extra per poter effettuare il debugging (i.e., vedere gdb)
-Wall			Stampa warning per tutti i possibili errori nel codice

Compiler: gcc

Opzioni			
Formato		Significato	Effetto
Compatto	Esteso		
-Idir			Specifica ulteriori direttori in cui cercare gli header file. Possibile specificare più direttori (-Idir1 -Idir2 ...). N.B. Non ci sono spazi tra -I e il nome del direttorio.
-lm			Specifica utilizzo libreria matematica
-Ldir			Specifica direttori per ricercare librerie preesistenti

Non inserire spazi

Esempio

```
gcc -Wall -g -I. -I/myDir/subDir -o myexe \  
myMain.c \  
fileLib1.c fileLib2.c file1.c \  
file2.c file3.c -lm
```

- ❖ Contestuale compilazione di diversi file sorgente, link e creazione dell'eseguibile
 - Comando su più righe
 - Fornisce "All Warnings"
 - Non ottimizzare il codice (debug)
 - Preleva gli header in due direttori
 - Inserisce la libreria matematica

Makefile

- ❖ Tool di supporto allo sviluppo di progetti complessi
- ❖ Sviluppato a partire dal 1998
- ❖ Costituito dalle utility
 - Makefile
 - Make
- ❖ Fornisce un mezzo conveniente per automatizzare le fasi di compilazione e linker
- ❖ Help
 - `man make`

Primo linguaggio di scripting che analizziamo

Strumento estremamente duttile, ma punto di forza principale è la verifica delle dipendenze

Makefile

- ❖ Makefile ha due scopi principali
 - Effettuare operazioni ripetitive
 - Evitare di (ri)fare operazioni inutili
 - Mediante la verifica di **dipendenze** e l'istante dell'ultima **modifica** evita di gestire ripetutamente operazioni inutili (e.g., ricompilare file non modificati)
- ❖ Si procede in due fasi
 - Si scrive un file Makefile
 - File di testo simile a uno script (di shell o altro)
 - Si interpreta il file Makefile con l'utility make
 - In questo modo si effettuano compilazione e link

Makefile

Opzioni			
Formato		Significato	Effetto
Compatto	Esteso		
-n			Non esegue i comandi ma li stampa solo
-i	--ignore-errors		Ignora gli eventuali errori e va avanti
-d			Stampa informazioni di debug durante l'esecuzione
	-- debug=[options]		Opzioni: a = print all info, b = basic info, v = verbose = basic + altro, i = implicit = verbose + altro

Makefile: Opzioni

- Il comando make può eseguire sorgenti (Makefile) diversi dal file standard ovvero
 - Il comando make esegue, di default
 - Il file makefile se esiste
 - Oppure il file Makefile se makefile non esiste
 - -f <nomeFile> (oppure --file <nomeFile>)
 - Permette di eseguire il Makefile di nome specificato
 - make --file <nomeFile>
 - make --file=<nomeFile>
 - make -f <nomeFile>

Makefile: Formato

Carattere di
tabulazione

```
target: dependency  
    <tab>command
```

❖ Ogni Makefile include

➤ Righe bianche

- Esse sono ignorate

➤ Righe che iniziano per '#'

- Esse sono commenti e sono ignorate

➤ Righe che specificano regole

- Ogni regola specifica un obiettivo, delle dipendenze e delle azioni e occupa una o più righe
- Righe molto lunghe possono essere spezzate inserendo il carattere "\" a fine riga

Makefile: Formato

```
target: dependency  
    <tab>command
```

- ❖ Quando si esegue un Makefile (con il comando `make`)
 - Il default è eseguire la prima regola
 - Ovvero quella che compare prima nel file
 - Nel caso esistano più regole può però esserne eseguita una specifica
 - `make <nomeTarget>`
 - `make -f <myMakefile> <nomeTarget>`

Makefile: Formato

```
target: dependency  
    <tab>command
```

❖ Ogni regola è costituita da

➤ Nome del target

- Spesso il nome di un file
- Talvolta il nome di un'azione

Si definisce "phony"
target

➤ Lista delle dipendenze da verificare prima di eseguire i comandi relativi alla regola

➤ Comando o elenco di comandi

- Ogni comando è preceduto da un carattere di **tabulazione**, invisibile ma **necessario**

Esempio 1: Target singolo

target:

```
<tab>gcc -Wall -o myExe main.c -lm
```

❖ Specifica

- Un'unica regola all'interno del file Makefile di nome **target**
- Il target non ha dipendenze

❖ Eseguendo il Makefile

- Viene eseguito il target
- Il target non ha dipendenze, quindi l'esecuzione del target corrisponde all'esecuzione del comando di compilazione

Esempio 2: Target multiplo

```
project1:
<tab>gcc -Wall -o project1 myFile1.c

project2:
<tab>gcc -Wall -o project2 myFile2.c
```

❖ Specifica più regole

- Occorre scegliere quale target eseguire
- Il default consiste nell'eseguire il primo target

❖ Eseguendo

- **make**
 - Viene eseguito il target project1
- **make project2**
 - Viene eseguito il target project2

Esempio 3: Target e azioni multiple

target:

```
<tab>gcc -Wall -o my \  
<tab>  main.c \  
<tab>  bst.c list.c queue.c stack.c  
<tab>cp my /home/myuser/bin
```

Comando su più
righe

clean:

```
<tab>rm -rf *.o *.txt
```

❖ Specifica più regole

- Le regole non hanno dipendenze
- Il primo target esegue due comandi (gcc e cp)
 - Esso viene eseguito con i comandi
 - make
 - make target

Esempio 3: Target e azioni multiple

target:

```
<tab>gcc -Wall -o my \  
<tab>  main.c \  
<tab>  bst.c list.c queue.c stack.c  
<tab>cp my /home/myuser/bin
```

Comando su più
righe

clean:

```
<tab>rm -rf *.o *.txt
```

- Il secondo target rimuove tutti i file di estensione .o e tutti quelli di estensione .txt
 - Esso viene eseguito con
 - make clean

Esempio 4: Dipendenze

```
target: file1.o file2.o
<tab>gcc -Wall -o myExe file1.o file2.o

file1.o: file1.c myLib1.h
<tab>gcc -Wall -g -I./dirI -c file1.c

file2.o: file2.c myLib1.h myLib2.h
<tab>gcc -Wall -g -I./dirI -c file2.c
```

- ❖ Esecuzione di target multipli in presenza di dipendenze
 - Si verifica se le dipendenze del target sono più recenti del target stesso
 - In tale caso si eseguono le dipendenze prima dei comandi procedendo in maniera ricorsiva

Esempio 4: Dipendenze

```
target: file1.o file2.o
<tab>gcc -Wall -o myExe file1.o file2.o

file1.o: file1.c myLib1.h
<tab>gcc -Wall -g -I./dirI -c file1.c

file2.o: file2.c myLib1.h myLib2.h
<tab>gcc -Wall -g -I./dirI -c file2.c
```

- ❖ Target ha come dipendenze file1.o e file2.o
 - Si verifica la regola file1.o
 - Se file1.c (oppure myLib1.h) è più recente di file1.o si esegue tale regola, ovvero il comando gcc
 - Altrimenti non si esegue tale regola
 - Si procede analogamente per la regola file2.o
 - Al termine si esegue il target **se** è necessario

Esempio 4: Dipendenze

Nome azione
(phony target)

```
target: file1.o file2.o  
    <tab>gcc -Wall -o myExe file1.o file2.o
```

...

Nome file

```
file2.o: file2.c myLib1.h myLib2.h  
    <tab>gcc -Wall -g -I./dirI -c file2.c
```

- ❖ Se il target non è il nome di un file è un "phony" target che dovrebbe sempre essere eseguito
- ❖ Per essere sicuri venga sempre eseguito
 - **.PHONY : target**

Indipendentemente dall'esistenza di un file con lo stesso nome più recente delle dipendenze

Regole implicite e modularità

- ❖ Esistono regole molto potenti per modularizzare e rendere più efficiente la scrittura dei makefile
 - Utilizzo di macro
 - Utilizzo di regole implicite
 - La dipendenza tra .o e .c è automatica
 - La dipendenza tra .c e .h è automatica
 - Le dipendenze ricorsive sono analizzate automaticamente
 - etc.

Esempio 5: Macro

```
CC=gcc
FLAGCS=-Wall -g
SRC=main.c bst.c list.c util.c
```

```
project: $(SRC)
<tab>$(CC) $(FLAGS) -o project $(SRC) -lm
```

Definizione macro:
macro=nome
(con o senza spazi)

Utilizzo della macro:
\$(macro)

❖ Le macro permettono di definire

➤ Simboli

- Compilatore, flag di compilazione, etc.

➤ Elenchi

- File oggetto, eseguiti, direttori, etc.

Esempio 6: Multi-folder

```
CC=gcc
FLAGCS=-Wall -g
SDIR=source
HDIR=header
ODIR=obj
```

La macro \$@
copia il "target name"
corrente

La macro \$^
copia l'elenco dei file nella
lista delle dipendenze

```
project: $(ODIR)/main.o $(ODIR)/bst.o
<tab>$(CC) $(FLAGS) -o $@ $^
```

```
$(ODIR)/main.o: $(SDIR)/main.c $(HDIR)/main.h
<tab>$(CC) $(FLAGS) -c $^
```

```
$(ODIR)/bst.o: $(SDIR)/bst.c $(HDIR)/bst.h
<tab>$(CC) $(FLAGS) -c $^
```

La macro \$< copierebbe il primo file dell'elenco
nella lista delle dipendenze

Gmake in VS Code

- ❖ In alcuni ambienti lo GNU Gmake viene utilizzato per creare direttamente il Makefile
- ❖ Procedura in VS Code
 1. Si apre il direttorio contenente il progetto
 2. Si inserisce un file di nome **CMakeLists.txt** che specifica le principali dipendenze

```
cmake_minimum_required(VERSION 3.22)

project(myapp)

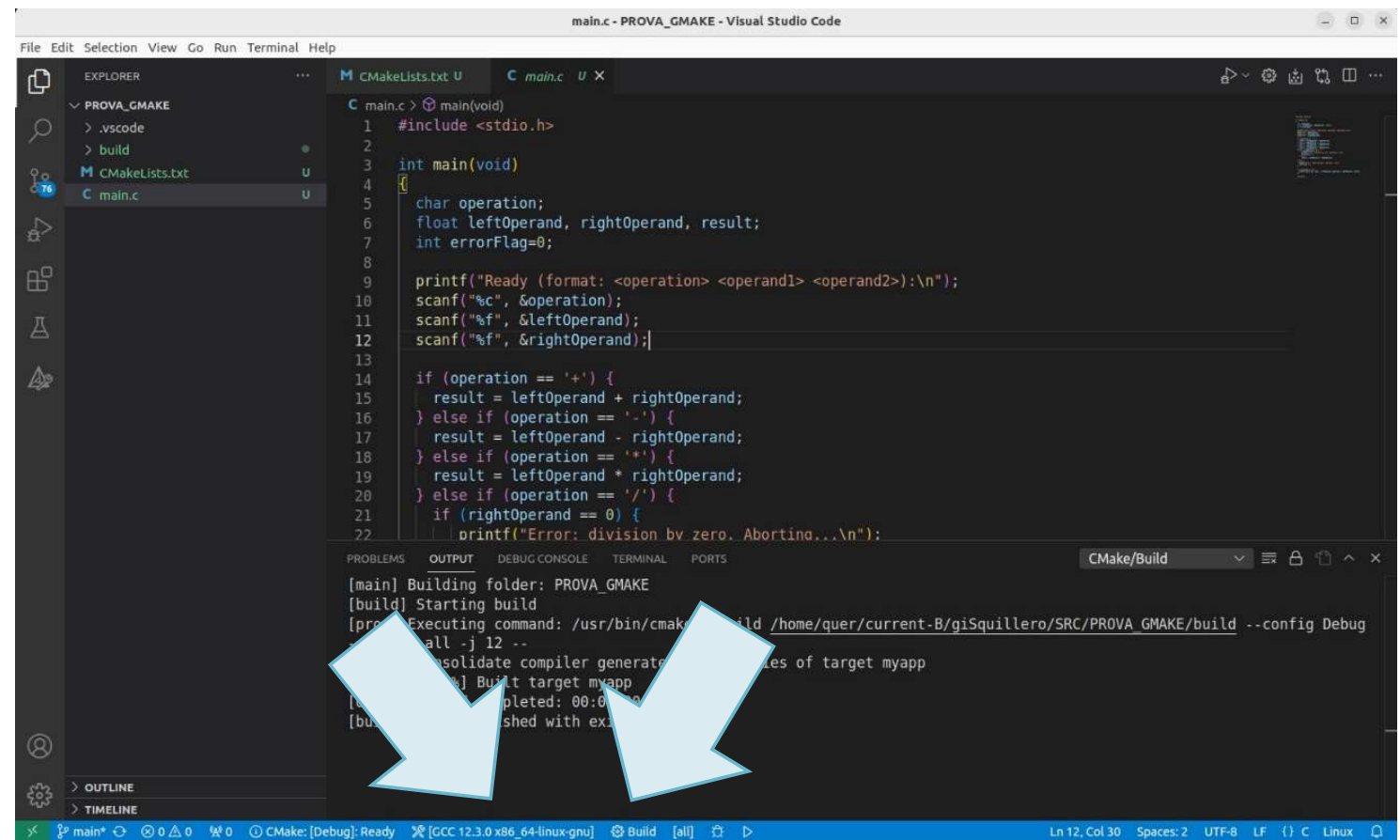
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pthread")

add_executable(myapp main.c)
```

Gmake in VS Code

3. Si seleziona **build** e il compilatore più recente

- Se il procedimento va a buon fine viene creata una cartella build con all'interno il Makefile



The screenshot shows the Visual Studio Code interface with a C program named `main.c` open. The program is a simple calculator that takes an operation and two operands as input and outputs the result. The code is as follows:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     char operation;
6     float leftOperand, rightOperand, result;
7     int errorFlag=0;
8
9     printf("Ready (format: <operation> <operand1> <operand2>):\n");
10    scanf("%c", &operation);
11    scanf("%f", &leftOperand);
12    scanf("%f", &rightOperand);
13
14    if (operation == '+') {
15        result = leftOperand + rightOperand;
16    } else if (operation == '-') {
17        result = leftOperand - rightOperand;
18    } else if (operation == '*') {
19        result = leftOperand * rightOperand;
20    } else if (operation == '/') {
21        if (rightOperand == 0) {
22            printf("Error: division by zero. Aborting...\n");
```

The bottom panel shows the output of the build process, which is as follows:

```
[main] Building folder: PROVA_GMAKE
[build] Starting build
[proc] Executing command: /usr/bin/cmake --build /home/quer/current-B/giSquillero/SRC/PROVA_GMAKE/build --config Debug
[proc] all -j 12 --
[proc] Consolidate compiler generated dependencies of target myapp
[proc] Built target myapp
[proc] Completed: 00:00:00
[build] Finished with exit code 0
```

Two large blue arrows point from the text "Se il procedimento va a buon fine viene creata una cartella build con all'interno il Makefile" to the `build` directory in the Explorer and the build output in the Output panel.

Gmake in VS Code

4. Build successivi inseriscono nella cartella build l'eseguibile dell'applicazione

Debugger

- ❖ Pacchetto software utilizzato per analizzare il comportamento di un altro programma allo scopo di individuare e eliminare eventuali errori (bug)
- ❖ Disponibile per quasi tutti i sistemi operativi
 - Spesso è integrato in IDE grafiche
- ❖ Può essere utilizzato
 - Come tool "stand-alone"
 - Utilizzo particolarmente scomodo
 - Completamente integrato con molti editor (e.g., emacs)
- ❖ I vari comandi possono essere forniti in maniera completa o abbreviata

Debugger: gdb

Azione	Comandi
Comandi esecuzione step-by-step	<code>run (r)</code> <code>next (n)</code> <code>next <numeroStep></code> <code>step (s)</code> <code>step <numeroStep></code> <code>stepi (si)</code> <code>finish (f)</code>
Comandi per breakpoint	<code>continue (c)</code> <code>info break</code> <code>break (b), ctrl-x-blank</code> <code>break numeroLinea</code> <code>break nomeFunzione</code> <code>fileName:numeroLinea</code> <code>disable numeroBreak</code> <code>enable numeroBreak</code>

Debugger: gdb

Azione	Comandi
Comandi di stampa	print (p) print espressione display espressione
Operazioni sullo stack	down (d) up (u) info args info locals
Comandi di listing del codice	list (l) list numeroLinea list primaLinea, ultimaLinea
Comandi per operazioni varie	file fileName exec fileName kill