Introduzione al Linguaggio C

Un corso intensivo per diventare ingegneri migliori

Alessandro Pellegrini
a.pellegrini@ing.uniroma2.it

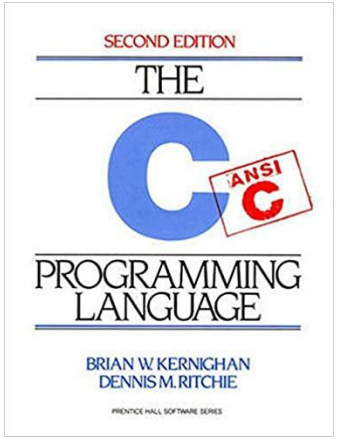
Informazioni Generali

- Informazioni sul docente
  - email: a.pellegrini@ing.uniroma2.it
  - URL: http://www.ce.uniroma2.it/~pellegrini/

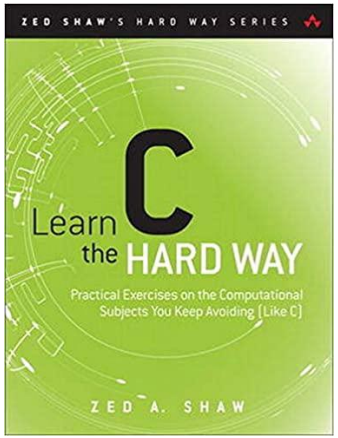
- Ricevimento:
  - Contattatemi via email

Materiale *supplementare*

- Brian W. Kernighan, Dennis M. Ritchie
  The C Programming Language (2nd edition)
  Prentice Hall
  ISBN: 978-0131103627



- Zed A. Shaw
  Learn C the Hard Way
  Addison-Wesley Professional
  ISBN: 978-0321884923

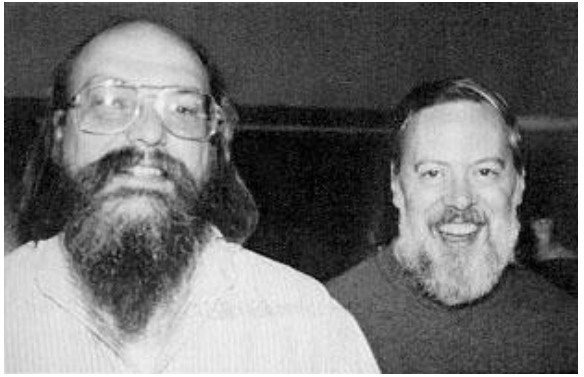


Che cos’è il C?

Che cos’è il C?

- Indovinate un po’: viene dopo il B!
- È un linguaggio procedurale progettato da Dennis Ritchie ai Laboratori Bell tra il 1972 e il 1973 per scrivere strumenti per Unix
- Fu poi usato da Ken Thompson e Dennis Ritchie per riscrivere il Kernel Unix, originariamente scritto in Assembly
- Hanno vinto il Turing Award nel 1983:

Il successo del sistema UNIX deriva dalla sua raffinata selezione di alcune idee chiave e dalla loro elegante implementazione. Il modello del sistema Unix ha portato una generazione di progettisti di software a nuovi modi di pensare alla programmazione. La genialità del sistema Unix è il suo framework, che consente ai programmatori di appoggiarsi al lavoro degli altri.



Che cos'è il C, *secondo la gente*?

- «il *C è più veloce di qualsiasi altro linguaggio (sano): forse, ma scommetto che se sommassi tutto il tempo risparmiato scrivendo in C, questo sarebbe di gran lunga minore del tempo che passato a sbattere la testa contro il muro cercando di debuggare*».
- «*con l'assembly, almeno, c'erano un mucchio di registri e nessuno ha mai cercato di fingere che fosse fantastico. L'assembly era onesto. Faceva schifo, ma almeno ti diceva: "ehi, faccio schifo"*».
- «*C, il più popolare di tutti i linguaggi per sistemi embedded, è un vero disastro, un bizzarro miscuglio pensato per dare al programmatore un controllo eccessivo sul computer*».
- «*Anche una cosa semplice come la matematica dei numeri interi può produrre risultati inaspettati. 20.000 + 20.000 può essere un numero negativo. È bello o che?!*».

Che cos'è il C, *secondo la gente*?

- «*Vi diranno che è il linguaggio migliore, vicino al metallo, e bla bla bla. Bene, una volta che avete passato 20 ore a debuggare un segfault perché qualche maledetto si è dimenticato di controllare un puntatore appeso, ditemi quanto è divertente stare vicino al metallo. Oppure provate a passare 10 giorni a scoprire quale delle milioni di funzioni nel codice non sta liberando memoria. Oh sì, ragazzi, allora amerete davvero il C*».
- «*Lasciate che vi dica come ci si sente a programmare in C: fate un pugno con la mano. Avanti, fatelo. Ora datevi un pugno davvero forte. Congratulazioni, ora siete programmatori C*».
- «*L'unica cosa che il C mi aiuta a capire è perché la mia attaccatura dei capelli si sta ritirando prematuramente*».
- «*Il C è rotto. È pieno di scelte di design che avevano senso negli anni '70 ma che non hanno alcun senso oggi*».

Che cos'è il C, *secondo me*?

- Un *linguaggio semplice*: è molto conciso
- Un *linguaggio versatile*: non è specializzato verso una specifica area di applicazione
- Un *linguaggio pericoloso*: se non si capisce cosa sta succedendo sotto al cofano, la probabilità che tutto scoppi è del 100%

- Semplificando molto: è un *assembly portabile*:
  - Si possono ottenere i migliori risultati, anche di performance, solo se si incomincia a "pensare come una macchina"

Quindi, di cosa parleremo in queste lezioni?

- Queste lezioni non parlano davvero del C di per sé:
  - Imparerete a scrivere programmi in C (C11 per quanto possibile)
  - Imparerete concetti legati alle architetture degli elaboratori
  - Imparerete concetti rigorosi di *programmazione difensiva*
- I programmatori possono usare Java e Python
  - Java: «*È un linguaggio aziendale; un linguaggio orientato al boilerplate, che consente a un vasto team di programmatori mediocri di creare qualcosa senza farla esplodere. Non può essere veloce, conciso o chiaro come altri linguaggi*».
  - Python: «*Python è come la Scientology dei linguaggi di programmazione. Tutto deve essere fatto nel modo in cui ha detto il Profeta. O ti guarderemo male*».
- Gli ingegneri possono scrivere in C… e in qualsiasi altro linguaggio.

Perché il C in un corso di Calcolatori Elettronici?

- L'Ingegneria Informatica tratta solo di *astrazioni*
- Le astrazioni hanno un costo:
  - Occorre padroneggiare le astrazioni per capire qual è la migliore per un certo compito od obiettivo
  - È molto facile dimenticarsi che qualsiasi astrazione, alla fine, è supportata da una CPU e dalla memoria
- Il C è un linguaggio di basso livello che fornisce un insieme molto piccolo di astrazioni
- Se siete competenti in C, probabilmente sapete cosa succede dietro le quinte di *molte* astrazioni.

La giusta mentalità

- Avete già usato altri linguaggi in passato?
- Il tipico flusso di lavoro è:
  - Lancia un IDE
  - Butta giù un po' di idee e di istruzioni
  - Lancia qualche interprete e vedi subito i risultati
  - Smanetta sul codice, finché non ottieni quello che volevi

- È una buona mentalità, se vogliamo accroccare un programma o sperimentare qualche idea
- Avrete forse già notato che con l'approccio "hack until it works" alla fine probabilmente non funziona niente

La giusta mentalità

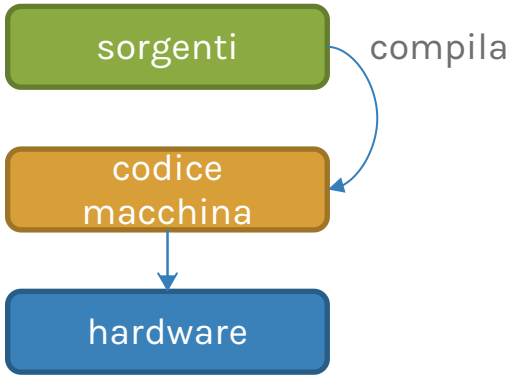
- Il C sarà più difficile all'inizio, perché richiede di pianificare in anticipo cosa si vuole creare
- È necessario progettare i componenti chiave del programma prima di iniziare ad implementarlo
  - Anche una piccola pianificazione può rendere tutto il processo più liscio
- Non si può essere sciatti nello scrivere, o non funzionerà niente
- Imparare il C rende ingegneri migliori perché:
  - siete obbligati a scontrarvi con problemi comuni molto prima e molto più frequentemente
  - il linguaggio, di per sé, è incredibilmente semplice
  - implicitamente, state imparando come funzionano le macchine
  - si diventa più bravi in abilità di programmazione di base

Il nostro modello di riferimento

Modelli di esecuzione e linguaggi

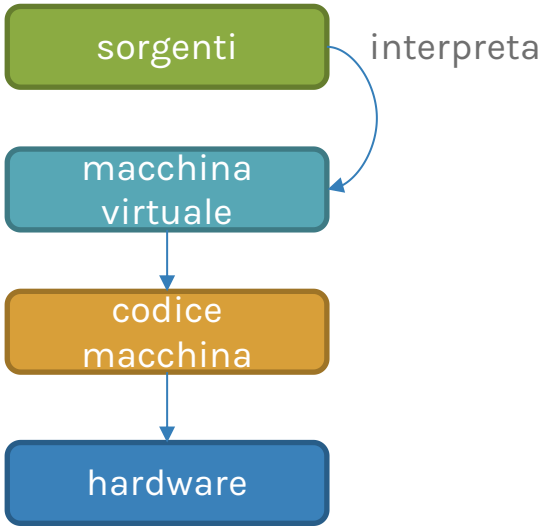
compilati

C, C++, Go, Fortran, Pascal



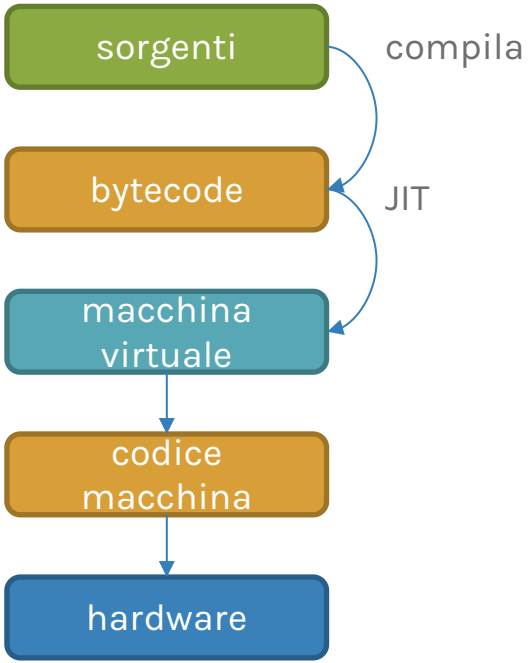
interpretati

Python, PHP, Ruby, Javascript

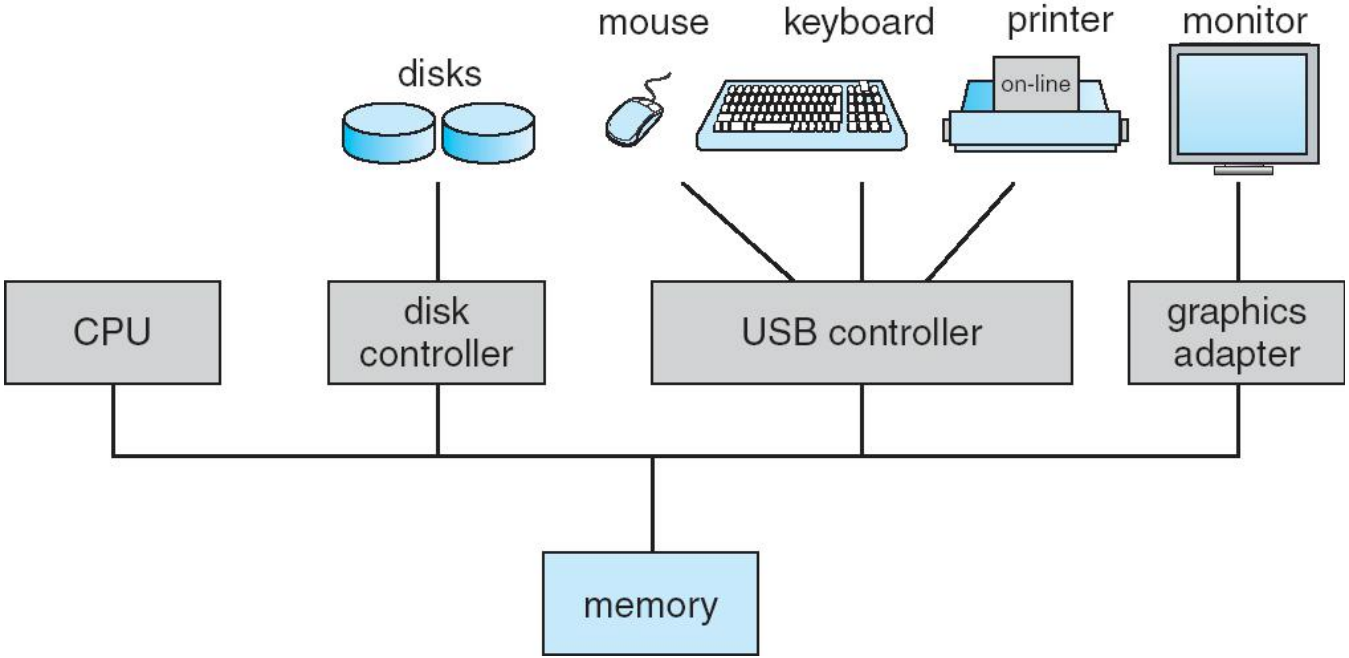


basati su bytecode

Java, Kotlin

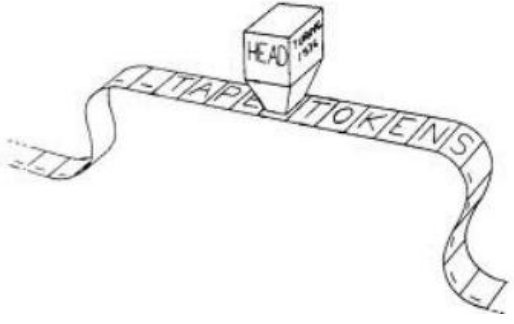


L’hardware è sempre alla base!



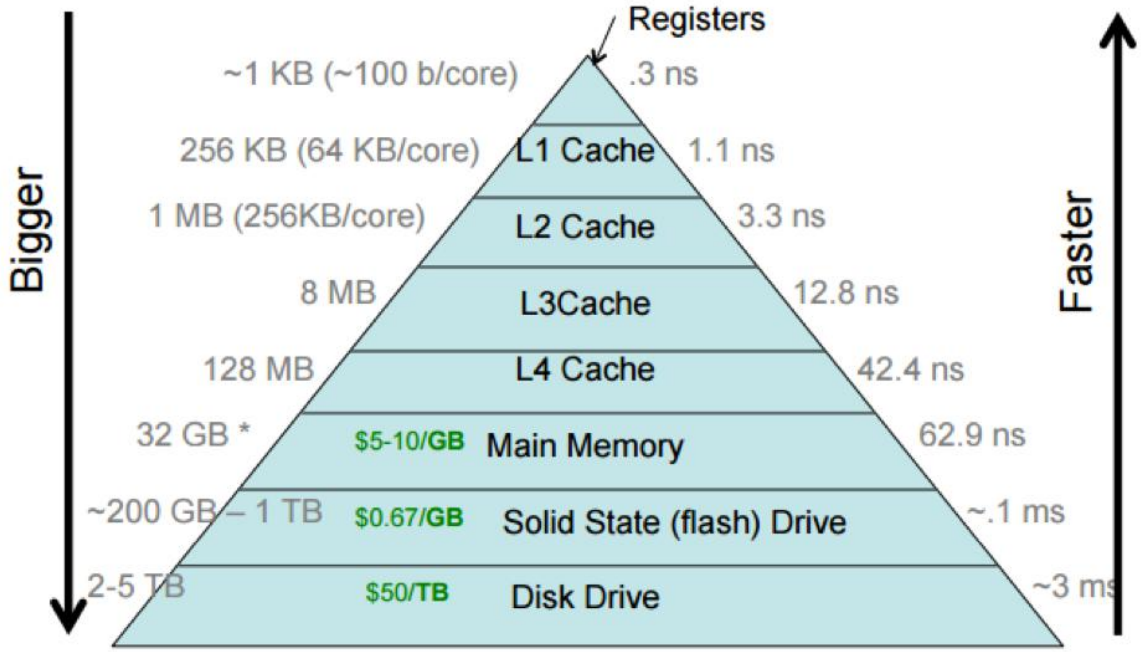
Il modello di memoria

- La memoria è un nastro molto lungo, diviso in celle (*modello di memoria piatta*)
- Ciascuna cella è identificata da un numero intero (*indirizzo*)
- Ogni gella ha una dimensione prefissata (un *byte*, tipicamente composto da 8 *bit*)
- Una “testina virtuale” si muove sul nastro per leggere o scrivere dati da/sulle celle (può essere effettuata solo una delle due operazioni alla volta)
- Una cella viene necessariamente scritta nella sua interezza



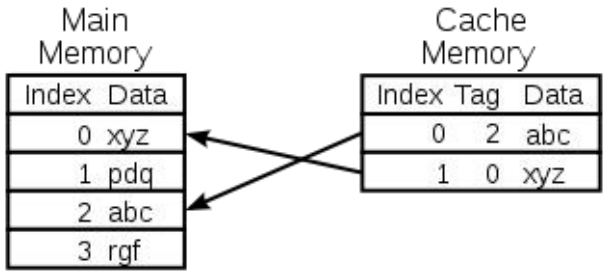
La gerarchia di memoria

- Un'unità di memorizzazione può essere realizzata con varie tecnologie differenti
- C'è un *tradeoff* tra costo e velocità



Le cache

- Una cache (un *deposito*) è più piccola dell'intera memoria principale di lavoro (RAM)
- Le cache mantengono *sottoinsiemi* dei dati
  - Tipicamente, le cache contengono copie dei dati
- Il numero, la natura e l'organizzazione delle cache possono rendere la scrittura di programmi più difficile e/o più performante



L’attrezzatura per questo corso

Lo “stack software”

- Il C è un linguaggio compilato: avete bisogno di un compilatore
- Su Linux:
  - È il sistema più semplice su cui configurare lo sviluppo in C:
  - La vita è più semplice se usate la riga di comando
  - Su Debian/Ubuntu lanciate:
    - `$ sudo apt-get install build-essential`
  - Su Fedora:
    - `$ sudo yum groupinstall development-tools`
  - Su Arch:
    - `$ sudo pacman -Suy gcc`
  - Su altre distribuzioni:
    - Cercate su Internet “c development tools”
  - Dopo l’installazione, questo comando dovrebbe funzionare:
    - `$ cc --version`
    - Con alta probabilità vi ritroverete ad usare gcc, ma anche CLang va benissimo per il corso

Lo “stack software”

- Su MacOs:
  - L’installazione è più semplice, anche se dovete tirarvi dentro tantissima roba pressoché inutile
  - Scaricate l’ultima versione di XCode
    - Il download è tipicamente molto pesante e richiede molto tempo
  - Per confermare che il vostro compilatore C funziona, scrivete in un terminale:
    - `$ cc --version`
  - Dovreste ritrovarvi ad utilizzare una qualche versione del compilatore CLang, ma se avete scaricato una versione più vecchia di XCode vi ritroverete ad usare gcc. Entrambi vanno bene.

Lo “stack software”

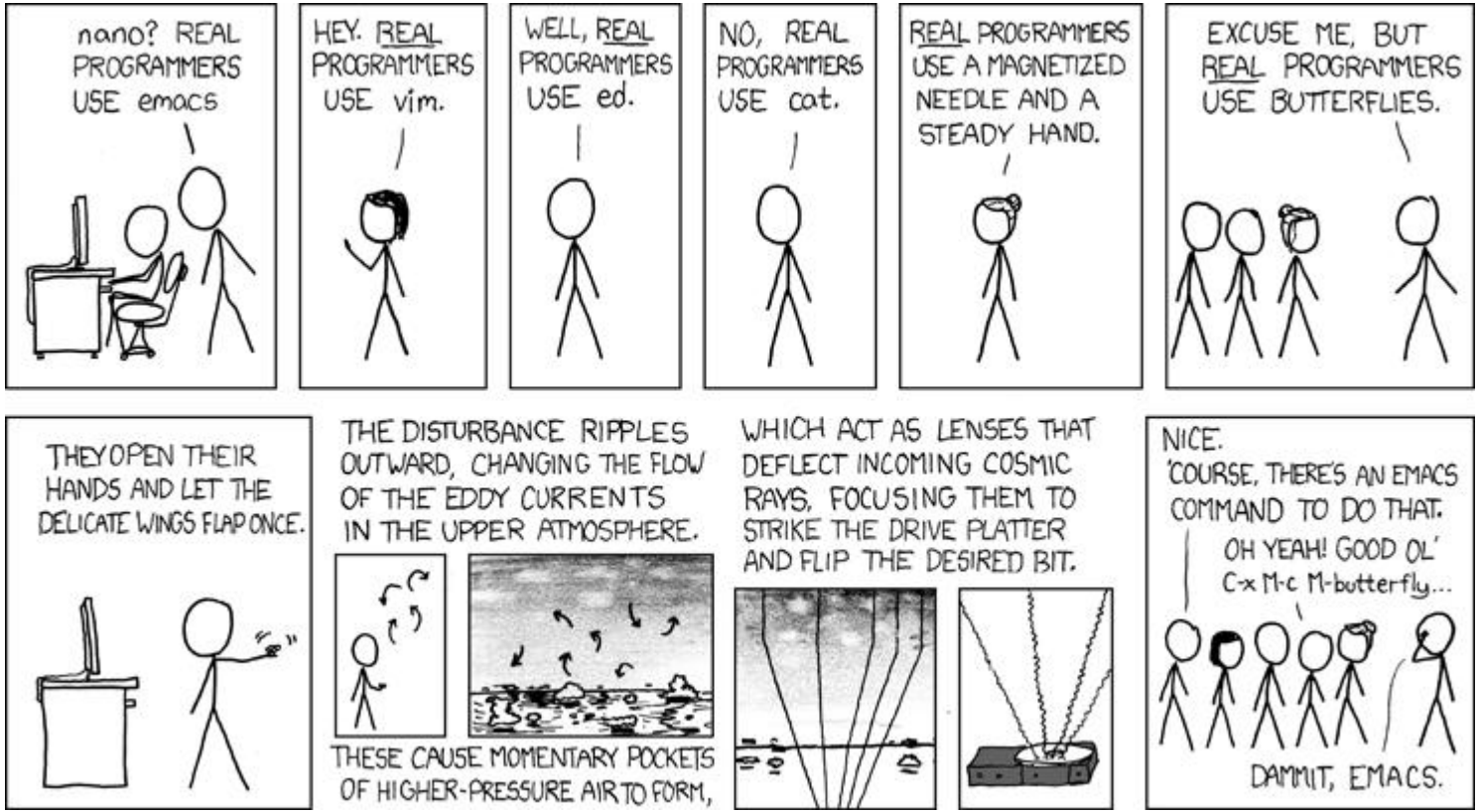
- Su Windows:
  - Il compilatore Microsoft installato con Visual Studio è pieno di troppe sovrastrutture inutili per questo corso
  - La soluzione più “semplice” è di scaricare e installare Cygwin
    - https://www.cygwin.com/
    - Otterrete anche molti strumenti di sviluppo Posix
  - Un’alternativa è il sistema MinGW:
    - http://www.mingw.org/
    - Più minimalista, ma funzionerà comunque
  - Un’opzione più avanzata: scaricate Virtual Box e installate una Virtual Machine con Linux
  - Un’opzione ancora più avanzata: usate il Windows Subsystem for Linux (WSL) su Windows 10 per lanciare Ubuntu in Windows
  - Un’opzione a “lungo termine”: create una nuova partizione e iniziate a usare Linux

L’Editor

ATTENZIONE: Evitate di usare qualsiasi Integrate Development Environment (IDE) mentre state imparando un linguaggio nuovo. Sono utili per velocizzare il lavoro, ma il loro aiuto in genere vi impedisce di imparare davvero il linguaggio.

- Varie opzioni possibili:
  - GEdit su Linux and MacOS
  - TextWrangler su MacOS
  - Notepad++ su Windows
  - Nano, che funziona nel terminale e si trova più o meno ovunque
  - Vim, che ha una certa curva di apprendimento
  - Emacs, che ha una curva di apprendimento più ripida
- Esiste l’editor perfetto per ciascuna persona nel mondo
- Non sprecate adesso troppo tempo a trovare quello perfetto per voi

L'Editor

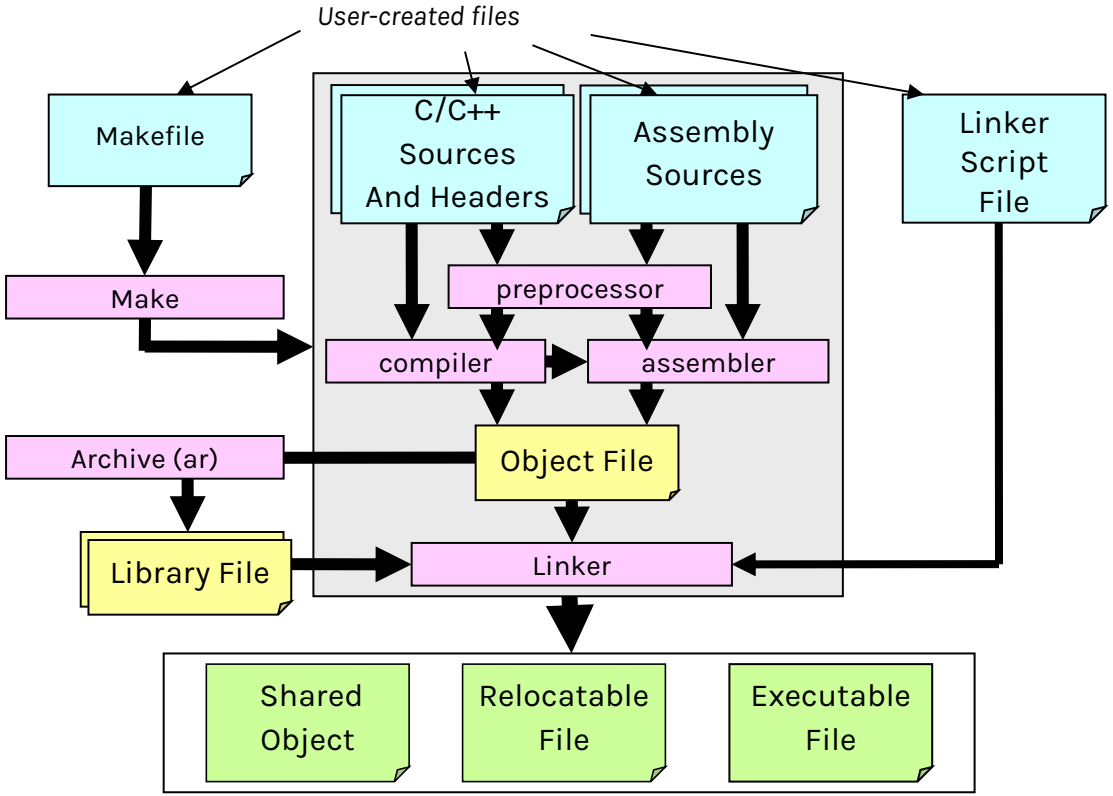

nano? REAL PROGRAMMERS USE emacs
HEY. REAL PROGRAMMERS USE vim.
WELL, REAL PROGRAMMERS USE ed.
NO, REAL PROGRAMMERS USE cat.
REAL PROGRAMMERS USE A MAGNETIZED NEEDLE AND A STEADY HAND.
EXCUSE ME, BUT REAL PROGRAMMERS USE BUTTERFLIES.
THEY OPEN THEIR HANDS AND LET THE DELICATE WINGS FLAP ONCE.
THE DISTURBANCE RIPPLES OUTWARD, CHANGING THE FLOW OF THE EDDY CURRENTS IN THE UPPER ATMOSPHERE.
THESE CAUSE MOMENTARY POCKETS OF HIGHER-PRESSURE AIR TO FORM,
WHICH ACT AS LENSES THAT DEFLECT INCOMING COSMIC RAYS, FOCUSING THEM TO STRIKE THE DRIVE PLATTER AND FLIP THE DESIRED BIT.
NICE.
'COURSE, THERE'S AN EMACS COMMAND TO DO THAT.
OH YEAH! GOOD OL' C-x M-c M-butterfly...
DAMMIT, EMACS.

Debugger

- Su Linux/Cygwin: possiamo usare GDB
  - Su Debian/Ubuntu:
    - `$ sudo apt-get install gdb`
  - Su Fedora:
    - `$ yum install gdb`
  - Su Arch:
    - `$ sudo pacman -Suy gdb`
- Su MacOs, GDB è rimpiazzato da LLDB
  - Se avete installato la toolchain di compilazione tramite XCode, lo troverete già installato

Riscaldamento

Il processo di compilazione



Hello World!

```c
#include <stdio.h>

/* This is a comment. */
int main(int argc, char *argv[])
{
    int times = 100;

    // this is also a comment
    printf("Hello World! I welcome you %d times.\n", times);
    return 0;
}
```

Hello World: Spezzettiamolo

```
1.  #include <stdio.h>
2.
3.  /* This is a comment. */
4.  int main(int argc, char *argv[])
5.  {
6.      int times = 100;
7.
8.      // this is also a comment
9.      printf("Hello World! I welcome you %d times.\n", times);
10.     return 0;
11. }
```

Chiediamo al preprocessore di importare un altro file

Un commento multilinea

I vostri programmi partono sempre dalla funzione `main`. Il sistema operativo carica il programma e lo lancia da qui. Per funzionare, deve restituire un `int` ed accettare due parametri: un `int` per il numero di parametri e un array di stringhe `char *` per i parametri

Inizio di un blocco

Dichiarazione e assegnazione di variabile locale

Un commento a linea singola

Chiamata a funzione. È una funzione strana, con un numero arbitrario di parametri

Valore di ritorno della funzione. In questo caso è il valore di ritorno al sistema operativo

La fine di un blocco

Notate che tutte le asserzioni terminano con un ‘;’ in C!

Makefile

- `make` è un programma che semplifica il processo di compilazione di altri programmi
- Basato su file di testo per guidare il processo di compilazione
- Basato su insiemi di regole:
  - la definizione di regole definisce i passi con cui eseguire un passo di compilazione
- Ciascuna regola ha delle *dipendenze*
  - `make` verifica automaticamente se un file nell'elenco delle dipendenze è stato modificato dall'ultima applicazione della regola di compilazione
  - In caso positivo, riapplica la regola di compilazione

Makefile per Hello World

```
CFLAGS=-Wall -Wextra
```

Dichiarazione di una variabile. Tutte le variabili sono stringhe. In questo caso chiediamo di generare dei *warning* per tutte le inesattezze del nostro programma, che non sono comunque errori.

```
all:ex1.c
```

Il nome di una regola. "`all`" è la regola di default, se non vengono passati parametri a `make`. `ex1.c` è l'unica dipendenza di questa regola.

```
	$(CC) $(CFLAGS) ex1.c -o ex1
```

tab, non spazi!

Implementazione della regola. `$(CC)` è una variabile di ambiente che identifica il compilatore predefinito di sistema. `$(CFLAGS)` richiama la variabile precedentemente dichiarata. ex1.c è il file di input al compilatore. `-o` è un flag che indica come chiamare l'eseguibile compilato (default: `a.out`)

```
clean:
```

Altra regola

```
	-rm ex1
```

Dice a make di ignorare gli errori nella riga

Un Makefile più generale

```
CFLAGS=-Wall -Wextra
TARGET=ex1
SRCS=ex1.c
OBJS=$(SRCS:.c=.o)

%.o:%.c
	$(CC) $< -c -o $@

all: $(OBJS)
	$(CC) $(OBJS) -o $(TARGET)

.PHONY: clean
clean:
	-$(RM) $(TARGET) $(OBJS)
```

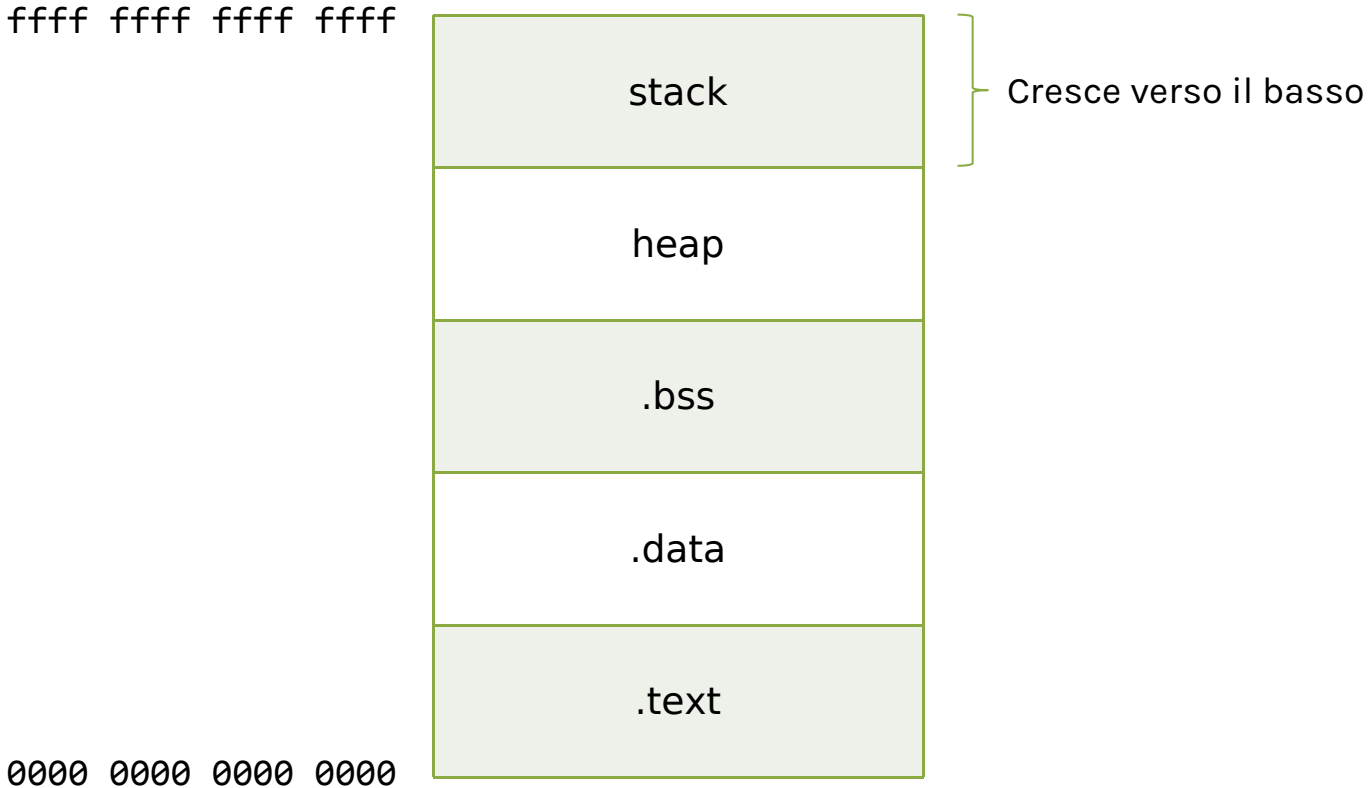
Comando di sostituzione testuale. Cerca tutte le occorrenze della stringa ".c" nella variabile SRCS e le rimpiazza con la stringa ".o"

Il % a sinistra fa corrispondere la regola a qualsiasi regola inizi con una sequenza qualsiasi di caratteri ma finisca con ".o". Il % a destra ha lo stesso valore del % a sinistra.

%< ha il valore del file di input alla regola (la dipendenza). $@ ha il valore del file di output della regola

.PHONY dice a `make` che il nome della regola non è un file. Cosa accadrebbe se esistesse un file chiamato "`clean`"?

Anatomia dei programmi in memoria



Hello World: Anatomia del programma

```
1. #include <stdio.h>
2.
3. /* This is a comment. */
4. int main(int argc, char *argv[])
5. {
6.     int times = 100;
7.
8.     // this is also a comment
9.     printf("Hello World! I
         welcome you %d times.\n",
         times);
10.    return 0;
11. }
```

```
.data
$LC0:
        .ascii  "Hello World! I welcome
                   you %d times.\012\000"

.text
main:
        addiu   $sp, $sp,-32
        sw      $ra, 28($sp)
        li      $a1, 100
        lui     $a0, %hi($LC0)
        addiu   $a0, $a0, %lo($LC0)
        jal     printf

        move    $v0, $zero
        lw      $ra, 28($sp)
        jr      $ra
        addiu   $sp, $sp, 32
```

Hello World: Anatomia del programma

```
.data
$LC0:
        .ascii  "Hello World! I welcome you %d times.\012\000"

.text
main:
        addiu   $sp, $sp, -32
        sw      $ra, 28($sp)
        li      $a1, 100
        lui     $a0, %hi($LC0)
        addiu   $a0, $a0, %lo($LC0)
        jal     printf

        move    $v0, $zero
        lw      $ra, 28($sp)
        jr      $ra
        addiu   $sp, $sp, 32
```

direttive assembly

etichette

variabili globali

opcode

registri destinazione

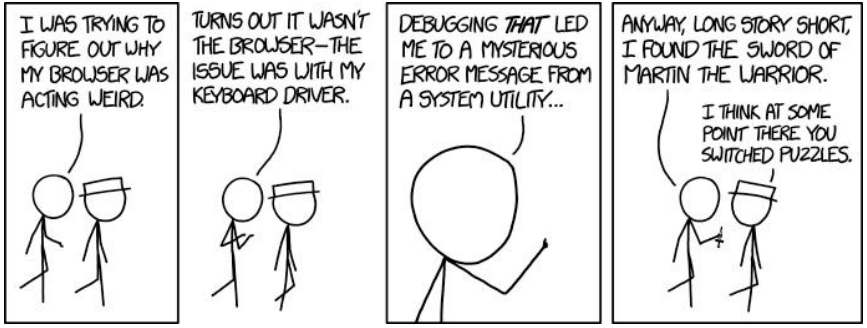
registri sorgente

costanti

indirizzi

Utilizzo di un debugger

- I debugger ci consentono di controllare l'esecuzione di un programma, interrompendo l'esecuzione in maniera selettiva
- Quando l'esecuzione è interrotta, ci consentono di osservare lo stato della memoria, lo stato dei registri, il codice assembly e, in taluni casi, anche il codice sorgente originale
- Possiamo anche modificare lo stato di un programma (cambiando, ad esempio, il contenuto di una variabile) prima di riprendere l'esecuzione
- Sono strumenti fondamentali, ma hanno una curva di apprendimento ripida
  - Iniziate ad utilizzare subito il debugger che avete installato in precedenza!



GDB Cheatsheet (LLDB è praticamente uguale)

- Compilate aggiungendo l'opzione `-g`
- Per lanciare il debugger: `gbd --args ./program [args]`
- Comandi principali:
  - `run [args]`: avvia il programma passando gli argomenti `[args]`.
  - `break [file:]function`: imposta un breakpoint.
  - `backtrace`: mostra un backtrace delle funzioni chiamate fin'ora.
  - `print: expr`: mostra il valore di `expr`.
  - `continue`: continua l'esecuzione del programma.
  - `next`: vai alla prossima riga nel sorgente, ma non entrare nelle funzioni.
  - `step`: vai alla prossima riga nel sorgente, entrando nelle funzioni.
  - `step instruction`: esegui una singola istruzione assembly
  - `quit`: termina l'esecuzione del debugger.
- Modalità interattiva: `CTRL-x, A` (o `--tui`)
  - `layout regs`: mostra una finestra con il contenuto dei registri
  - `layout asm`: mostra il sorgente assembly
  - `layout src`: mostra il sorgente C
  - `focus [what]`: sposta il focus su una particolare finestra