Introduzione alla programmazione C

Introduzione ai linguaggi di Programmazione e al Linguaggio C

Sommario

- Linguaggi di Programmazione
- Fasi della Programmazione
  - Dal Problema al Programma
  - Il ciclo di sviluppo di un software
- La shell
  - welcome.c
  - Comandi della Shell
- Compilatore gcc
  - Opzioni del compilatore
  - Le fasi della Compilazione
- Struttura del sorgente C
  - Istruzioni per il preprocessore
  - Keyword
  - La funzione Main e l'istruzione return
  - La funzione printf()

Linguaggi di Programmazione

INTRODUZIONE ALLA PROGRAMMAZIONE IN C

CAPITOLO 1

Key Word

- #Linguaggi
  #Astrazione, #Sintassi, #Semantica, #Dizionario
- #Classificazioni Linguaggi,
  #Interpretati, #Compilati
- #Portabilità,
- #Ambiente Esecuzione
  #Sorgente, #Eseguibile, #Run-Time,
  #Compile-Time


DIZIONARIO
SINTASSI
ALFABETO
SEMANTICA
LINGUAGGIO

Evoluzione dei linguaggi: ASTRAZIONE!

LINGUAGGI

Linguaggio…	Naturale	Alto Livello	Basso Livello
Orientato a …	Uomini	Compilatore	singola CPU
Caratteristiche	Ambiguo, ricco, flessibile	Non ambiguo, Formale, Rigido	
Alfabeto	A, B,… , Z (26 + punteggiature)	Codice ASCII (128)	0, 1 (2)
Dizionario	Linguistico	Keyword (32 nel C) + funzioni	Instruction Set Architecture (ISA)
Sintassi (Es.)	'.' alla fine di una frase	';' alla fine di un'istruzione	A capo alla fine di un'istruzione

Esempio: calcolare la somma di due numeri A e B

Linguaggio macchina

```
00000010101111001010
00000010111111001000
00000011001110101000
```

Linguaggio assembly

```
LOAD   A
ADD    B
STORE  S
```

Linguaggio C

```
S = A + B;
```

Esempio:

"Stampa sullo schermo la somma fra C ed il prodotto di A e B":

Linguaggio ad alto livello (C++):

```
cout << A * B + C;
```

Linguaggio Assembly:

```
mov eax,A
mul B
add eax,C
call WriteInt
```

Linguaggio macchina:

```
A1     00000000
F7 25 00000004
03 05 00000008
E8 00500000
```

Codifica Esadecimale
$(A)_{16} = (1010)_2$

Esempio: funzione per il calcolo della media di N numeri in C e Assembly

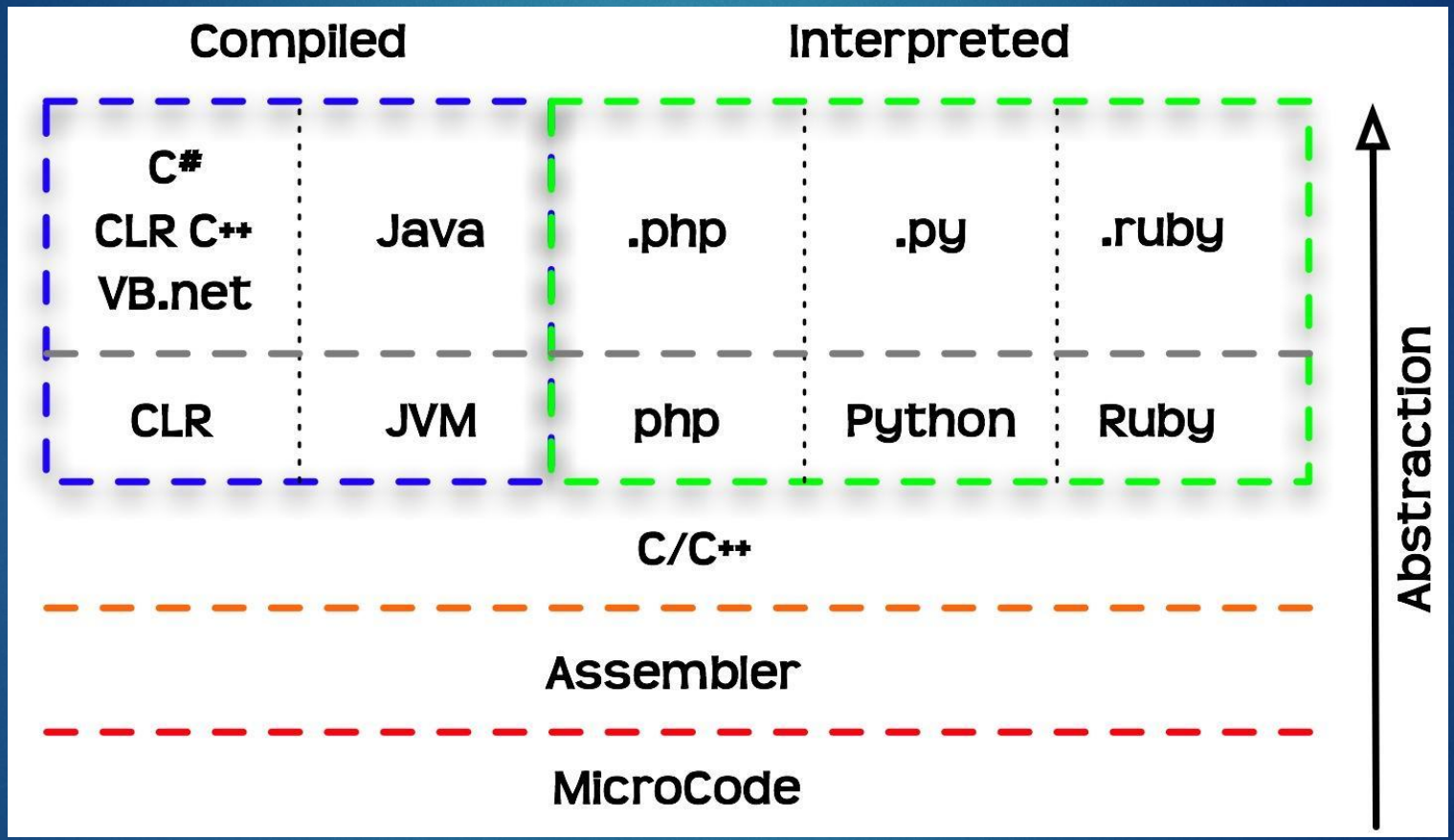
Linguaggio C (alto livello)

```
double mean (double* x, unsigned n)
{
  double m = 0;
  int i;
  for (i=0; i<n; i++)
     m += x[i];
  m /= n;
  return m;
}
```

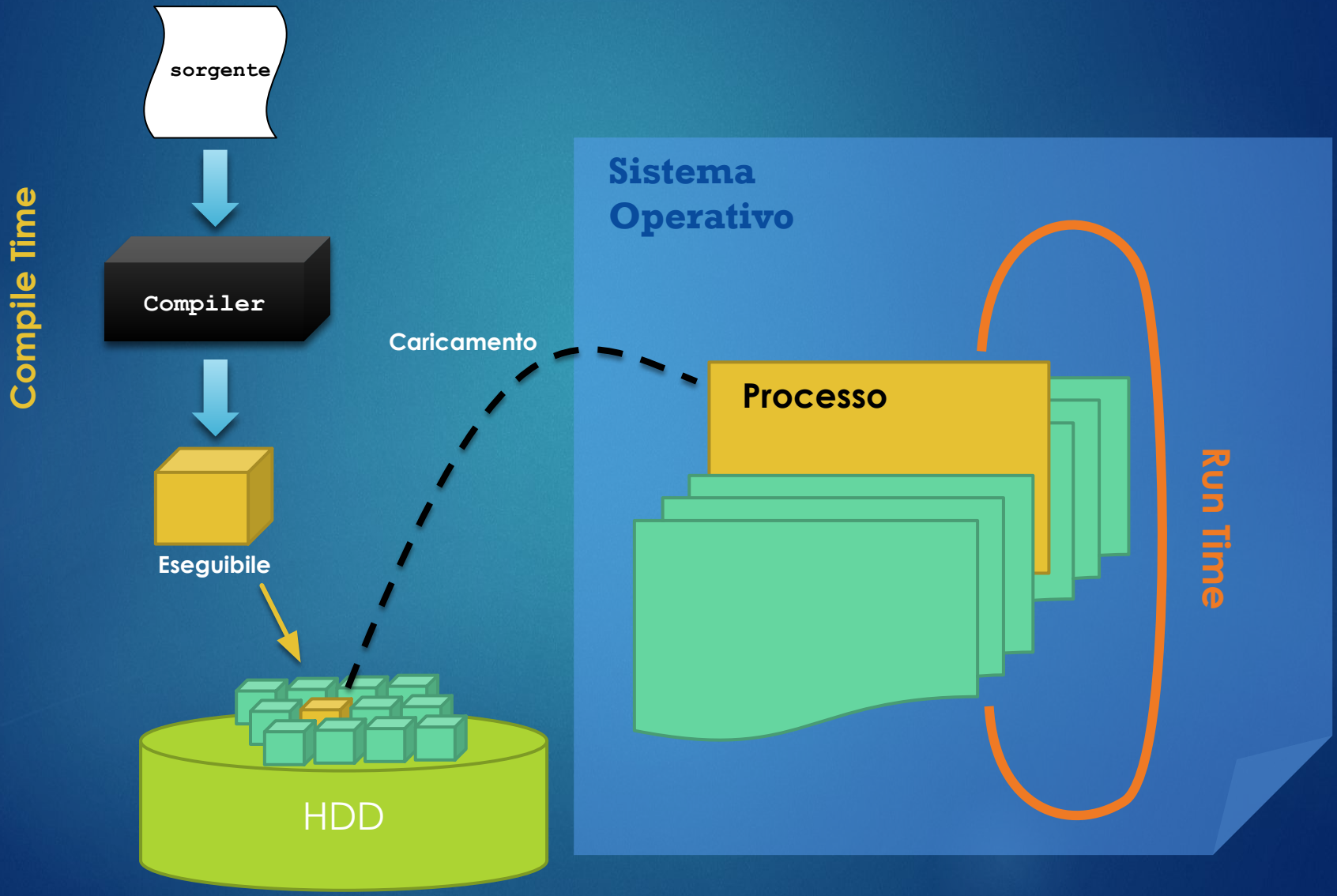
Linguaggio Assembly

```
            .file       "qq.c"
            .text
.globl mean
            .type       mean,@function
mean:
            pushl       %ebp
            movl        %esp, %ebp
            subl        $24, %esp
            movl        $0, -8(%ebp)
            movl        $0, -4(%ebp)
            movl        $0, -12(%ebp)
.L2:
            movl        -12(%ebp), %eax
            cmpl        12(%ebp), %eax
            jb          .L5
            jmp         .L3
.L5:
            movl        -12(%ebp), %eax
            leal        0(,%eax,8), %edx
            movl        8(%ebp), %eax
            fldl        -8(%ebp)
            faddl       (%eax,%edx)
            fstpl       -8(%ebp)
            leal        -12(%ebp), %eax
            incl        (%eax)
            jmp         .L2
.L3:
            movl        12(%ebp), %eax
            movl        $0, %edx
            pushl       %edx
            pushl       %eax
            fildll      (%esp)
            leal        8(%esp), %esp
            fldl        -8(%ebp)
            fdivp       %st, %st(1)
            fstpl       -8(%ebp)
            movl        -8(%ebp), %eax
            movl        -4(%ebp), %edx
            movl        %eax, -24(%ebp)
            movl        %edx, -20(%ebp)
            fldl        -24(%ebp)
            leave
            ret
.Lfe1:
            .size       mean,.Lfe1-mean
            .ident      "GCC: (GNU) 3.2.2
```

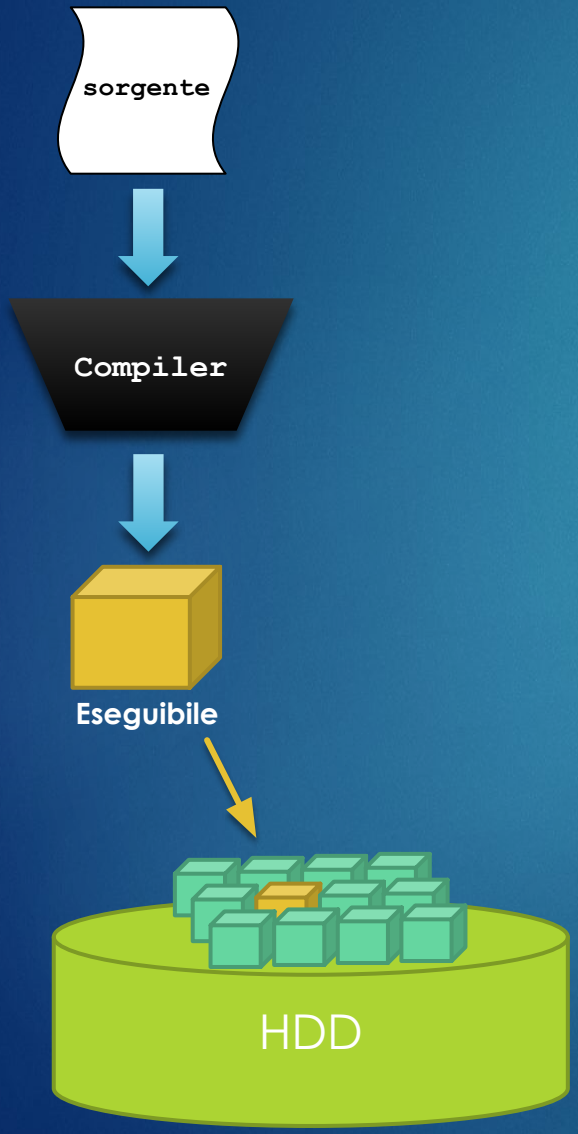
Classificazioni

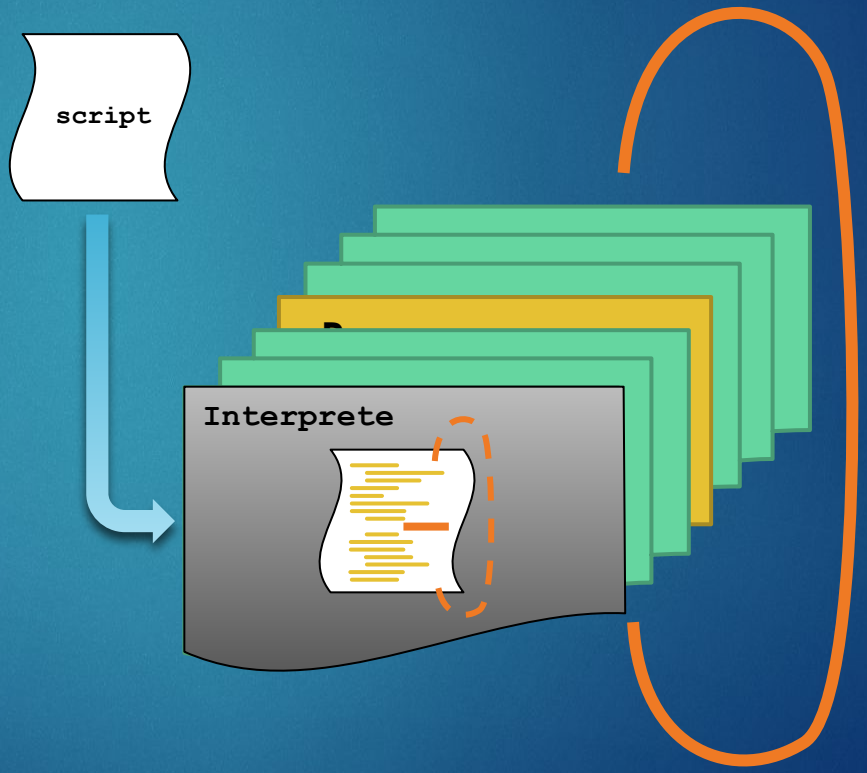


Compiler vs Interpreter



Compiler vs Interpreter





Compiler vs Interpreter

#	COMPILER	INTERPRETER
1	Compiler works on the complete program at once. It takes the **entire program** as input.	Interpreter program works line-by-line. It takes **one statement at a time** as input.
2	Compiler generates intermediate code, called the o**bject code or machine code.**	Interpreter does not generate intermediate object code or machine code.
3	Compiler executes conditional control statements (like if-else and switch-case) and logical constructs **faster than interpreter.**	Interpreter execute conditional control statements at a much **slower speed**.
4	**Compiled programs take more memory** because the entire object code has to reside in memory.	Interpreter does not generate intermediate object code. As a result, **interpreted programs are more memory efficient.**
5	Compile once and run anytime. Compiled program does not need to be compiled every time.	Interpreted programs are interpreted line-by-line every time they are run.
6	Errors are reported after the **entire program is checked** for syntactical and other errors.	Error is reported as soon as the first error is encountered. Rest of the program will not be checked until the existing error is removed.
7	A compiled language is more difficult to debug.	Debugging is easy because interpreter stops and reports errors as it encounters them.
8	Compiler does not allow a program to run until it is completely error-free.	Interpreter runs the program from first line and stops execution only if it encounters an error.
9	Compiled languages are more efficient but difficult to debug.	Interpreted languages are less efficient but easier to debug. This makes such languages an ideal choice for new students.
10	**Examples** of programming languages that use compilers: C, C++, COBOL	**Examples** of programming languages that use interpreters: BASIC, Visual Basic, Python, Ruby, PHP, Perl, MATLAB, Lisp

In COnclusion

Compiled		Interpreted	
PROS	CONS	PROS	CONS
ready to run	**not** cross platform	cross-platform	interpreter required
often **faster**	inflexible	simpler to test	often **slower**
source code is **private**	extra step	easier to debug	source code is **public**

Fasi della Programmazione

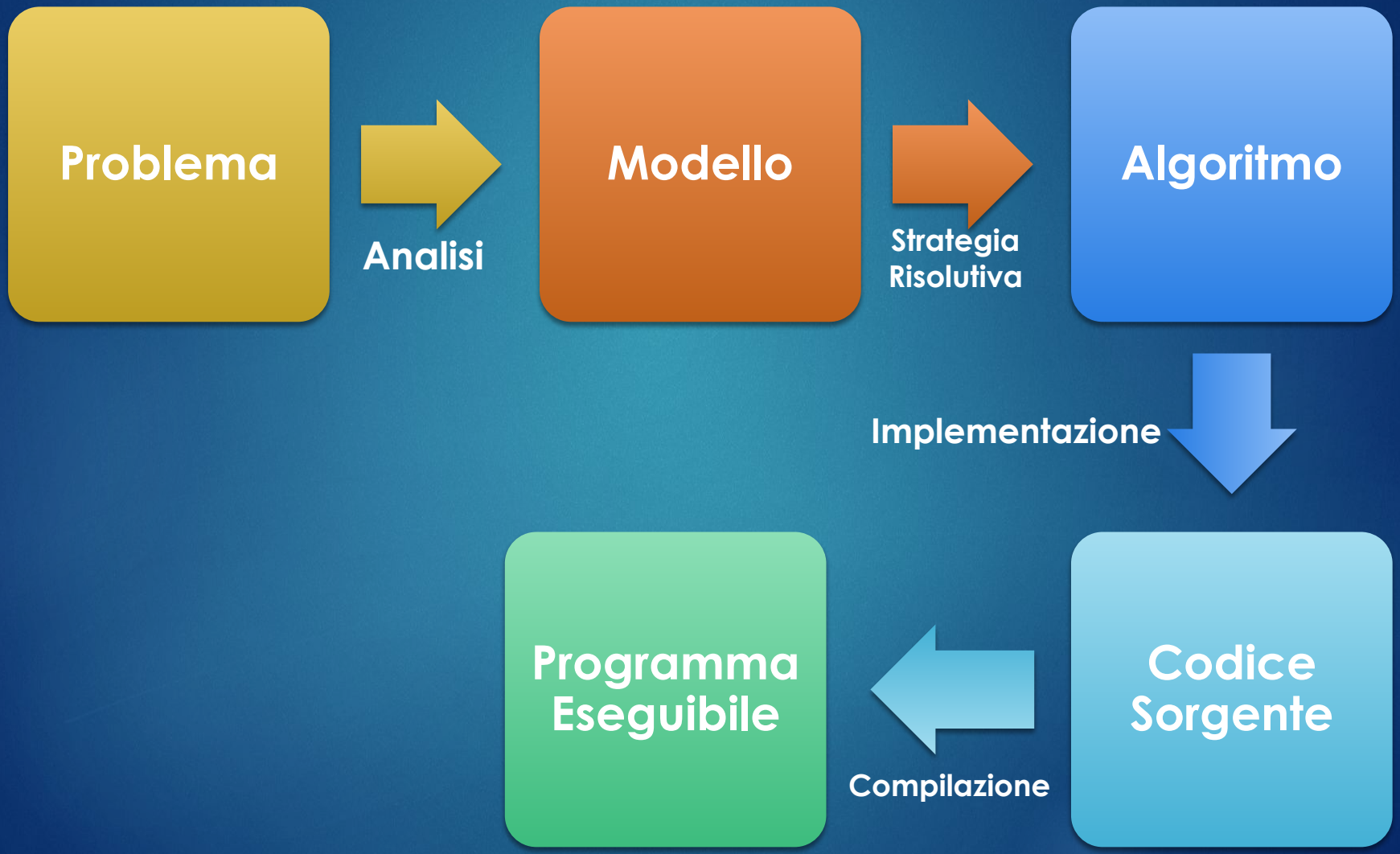
INTRODUZIONE ALLA PROGRAMMAZIONE IN C

CAPITOLO 2

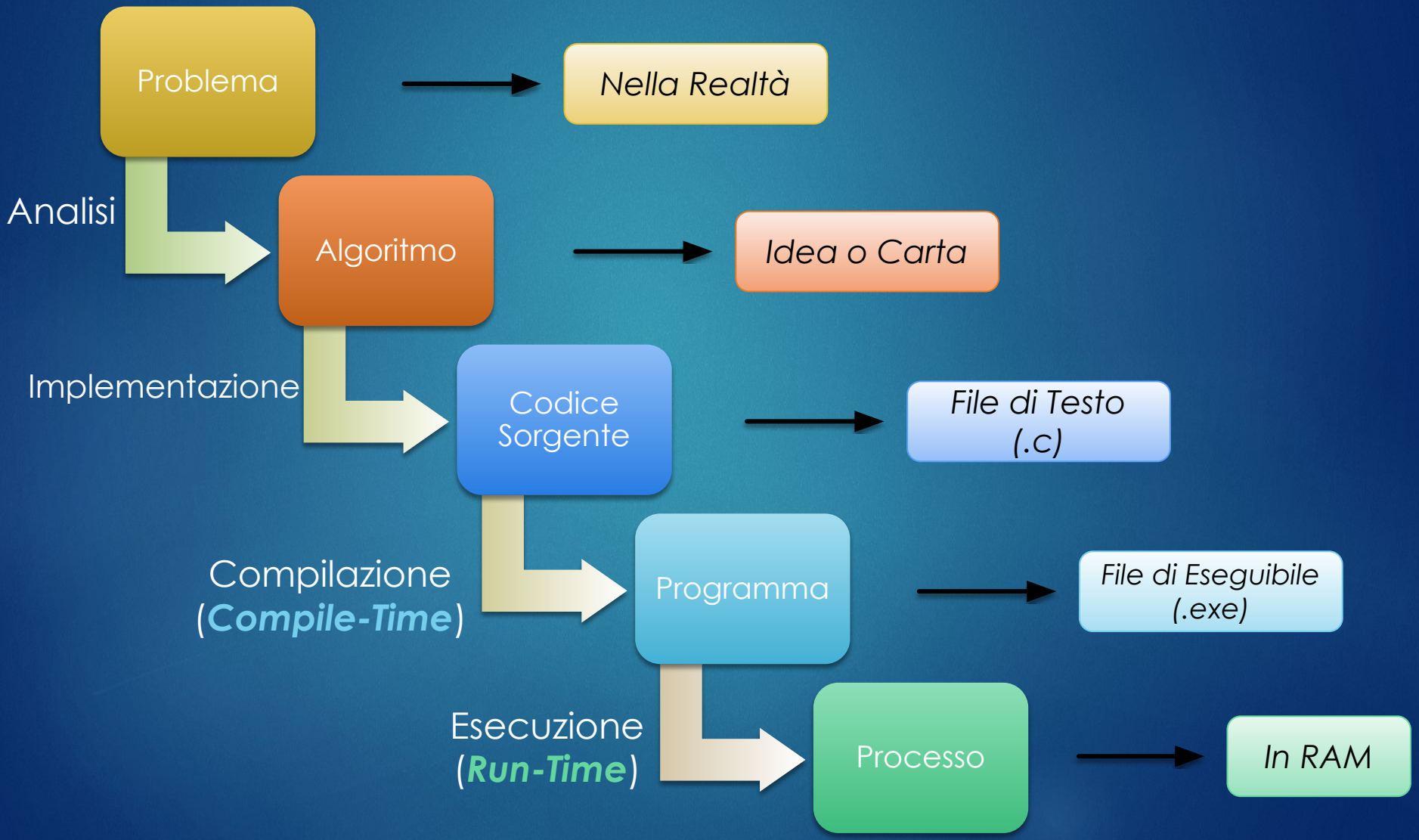
Key Word

- #Problem Solving
  #Analisi, #Modello, #Algoritmo,#Implementazione
- #Ciclo Sviluppo
  #Test, #Debug

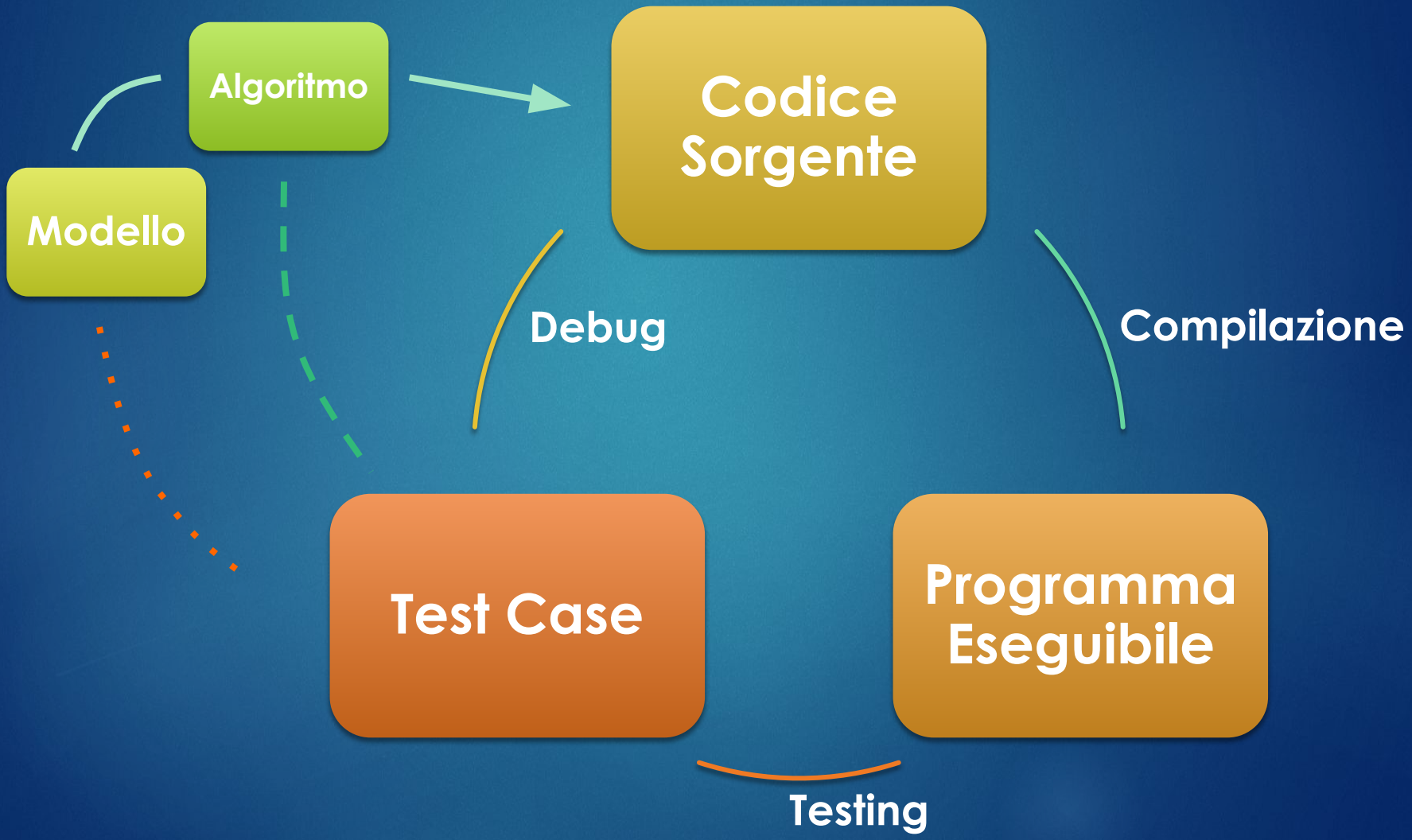
Dal Problema al Programma



Dal Problema al Processo



Ciclo di Sviluppo del Software



La shell

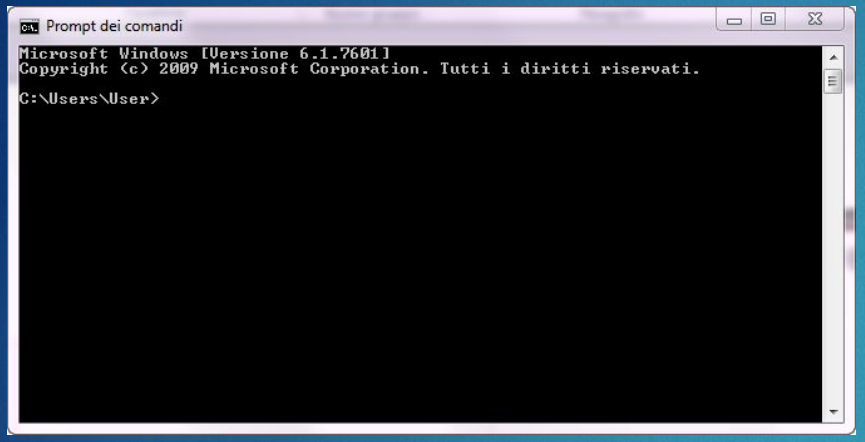
L'AMBIENTE DI ESECUZIONE DA LINEA DI COMANDO (CLI)

CAPITOLO 3

Key Word

- #CMD
- #CLI, #Comandi del DOS

Eseguire un Programma da Shell



«Invocare» il programma:

1. scrivere il suo ***nome.exe***
2. premere INVIO

Esempio: ***calc.exe***

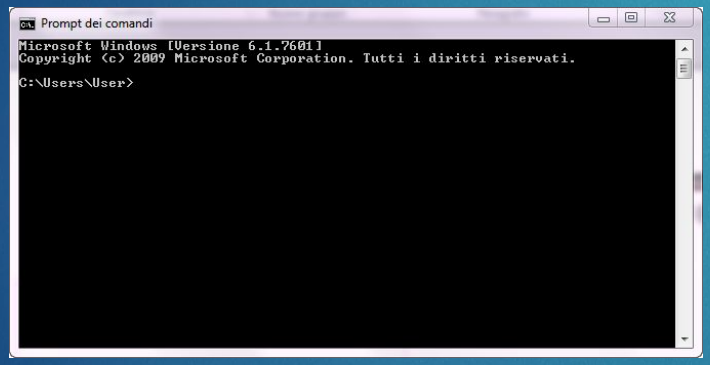
Domanda

Occorrente:

1. Shell (CMD)

In quale cartella si trova il vostro sorgente?

Localizziamolo ora con la *shell*



Fase:

- ✓ Editing del codice
- ○ Compilatore
- ○ Compilazione
- ○ Esecuzione
- ○ Test

Comandi della shell

"Navigare" tra le cartelle

Comando	Funzione	Esempi
dir	Elencare i file	
cd	Cambiare directory	• `cd ..` • `cd \` • `cd Directory\SubDirectory`
md	Crea Directory	`md Prova`
>	Pipe	`dir > pippo.txt`
copy	Copia un file	`copy pippo.txt pippo.old`
ren	Rinomina file	`ren pippo.old pippo_01.txt`
cls	Cancellare una schermata	`cls`

Fase:

-  Editing del codice
- ◯ Compilazione
- ◯ Esecuzione
- ◯ Test

… l'idea generale

Compilatore gcc

COMPILAZIONE DA LINEA DI COMANDO

Key Word

- #GCC
  #Porting, #MinGW
- #IDE
  #DevC, #CodeBlocks
- #Editing
- #Variabili Ambiente, #PATH
- #Opzioni Compilatore
- #Fasi Compilazione
  #Preprocessore, #Compilatore, #Assemblatore, #Linker

Compilatori

Noi useremo **gcc** (*GNU C Compiler*)

… o meglio il suo *porting* su windows:

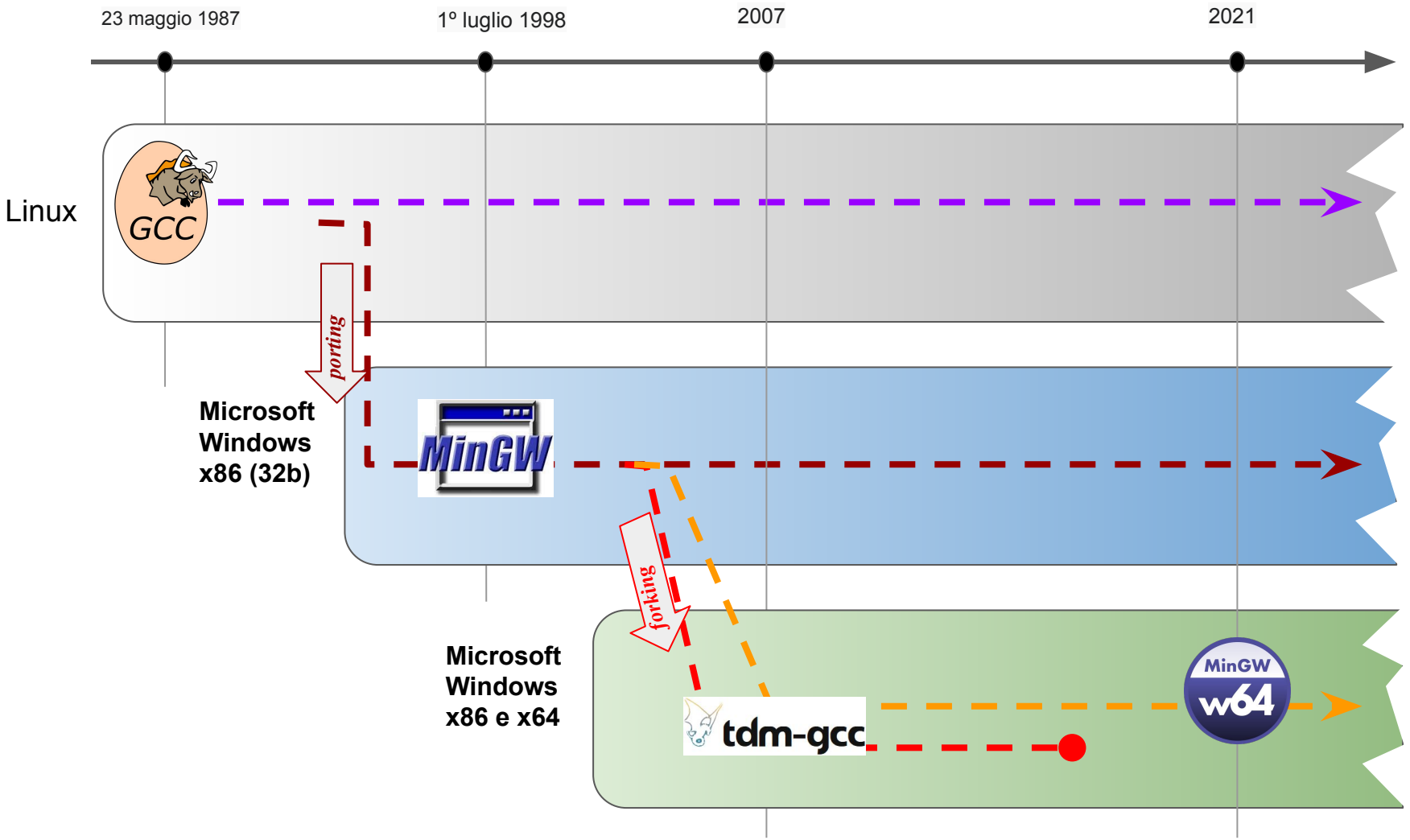
MinGW

Si tratta di una suite di **programmi** CLI e **librerie** che interagiscono con le API di Windows.

La versione che utilizzeremo è preconfezionata nell'installazione dell'***Ambiente di Sviluppo Integrato*** (IDE) che dovrete installare: Orwell DEV-C++

Evoluzione ***gcc***:

Da Linux A Windows tra *Porting* e *Forking*



Welcome.c

Occorrente:

1. Notepad.exe

Il nostro primo SORGENTE:

```
/* welcome.c
*  Primo programma in C
*/
#include <stdio.h>

int main()
{
    printf("Welcome to C!\n");

    return 0;
}
```

Fase:

- (✓) Editing del codice
- ( ) Compilatore
- ( ) Compilazione
- ( ) Esecuzione
- ( ) Test

Salviamo il nostro file: **`welcome.c`**

Compilatore gcc.exe

Occorrente:

1. Shell (CMD)
2. compilatore

Dove si trova il compilatore?

Localizziamolo ora con la *shell*

- Spesso viene installato nella cartella radice

```
> cd \
> dir
```

- Solitamente è nella cartella dell'IDE:

```
> cd Dev-Cpp\bin
> dir
```

- Funziona?

```
> gcc
> gcc -v
```

In questo esempio supponiamo che il Dev sia stato installato direttamente nella radice del disco C

Fase:

- ✓ Editing del codice
- ✓ Compilatore
- ◯ Compilazione
- ◯ Esecuzione
- ◯ Test

Compilatore gcc.exe

Occorrente:

1. Shell (CMD)

Ma ogni volta, dobbiamo metterci nella cartella dove si trova il compilatore?

Impostiamo la *variabile d'ambiente* **PATH**

```
set path=%path%;c:\directoryPath
```

`directoryPath` = path del compilatore

Nel nostro caso:

Per verificare che il path del compilatore sia stato aggiunto alla variabile d'ambiente %path% ci basta visualizzarla con il comando **echo**:

```
echo %path%
```

Fase:

- ✓ Editing del codice
- ✓ Compilatore
- ○ Compilazione
- ○ Esecuzione
- ○ Test

Compilatore gcc.exe

Occorrente:

1. Shell (CMD)
2. gcc
3. welcome.c

Proviamo a compilare?

Torniamo nella cartella del sorgente

Proviamo ora a lanciare:

```
>gcc welcome.c
```

Non è successo nulla? …controlliamo:

```
>dir
```

a.exe? …e chi lo ha messo lì? Proviamo a lanciarlo:

```
>a
```

Fase:

-  Editing del codice
-  Compilatore
-  Compilazione
-  Esecuzione
- Test

Compilatore gcc.exe

Funziona!!!!!!!

Cerchiamo di capire qualcosa di più…

Occorrente:

1. Shell (CMD)
2. gcc
3. welcome.exe

Proviamo ora a lanciare:

```
>gcc welcome.c –o welcome.exe
>dir
```

gcc supporta **parecchie opzioni**:

-o permette di specificare il nome del target

--help fornisce l'help

-Wall visualizza tutti i *warnings*

-c compila e assembla ma non linka

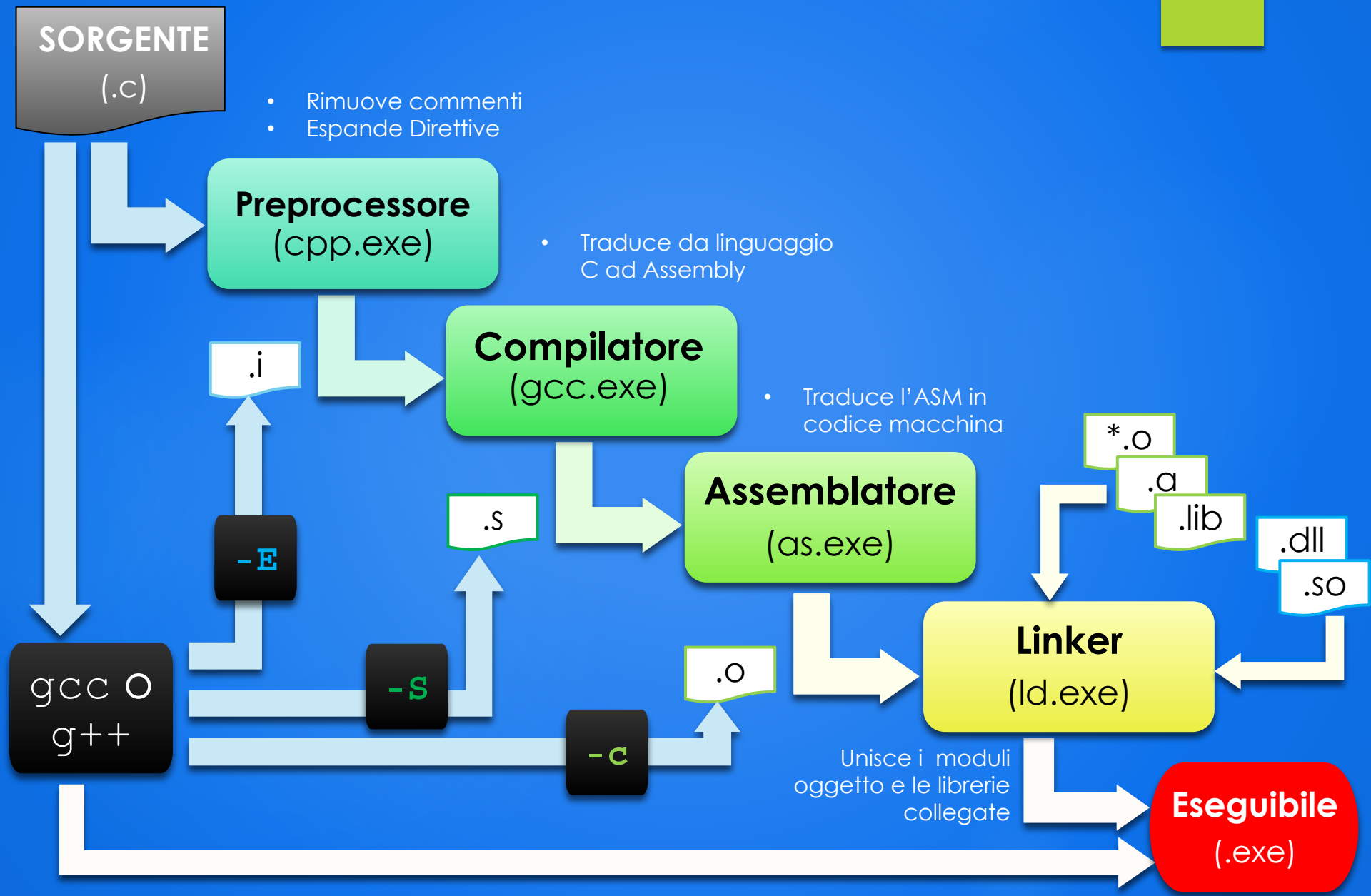
-S compila ma non assembla

-E solo preprocessore

Fase:

- [x] Editing del codice
- [x] Compilatore
- [x] Compilazione
- [x] Esecuzione
- [ ] Test

FASI DELLA COMPILAZIONE



Le tappe di gcc.exe

Vediamo come evolve il codice nelle fasi di compilazione

Cosa fa il **Preprocessore**?

```
>gcc -E welcome.c -o welcome.i
```

Apriamo **welcome.i** con Blocco Note…

… e il **Compilatore**?

```
>gcc -S welcome.c -o welcome.s
```

Apriamo **welcome.s** con Blocco Note…

… e l'**Assemblatore**?

```
>gcc -c welcome.c -o welcome.o
```

Apriamo **welcome.o** con Blocco Note… (meglio HxD)

… e il **Linker**?

```
>gcc welcome.c -o welcome.exe
```

Apriamo **welcome.o** con Blocco Note…

Nota

Con l'opzione

`-save-temps`

di gcc è possibile generare tutti e 4 i tipi di file con un solo comando.

Le opzioni di gcc.exe

Attenzione!

Soprattutto nei primi tempi, per apprendere il linguaggio è opportuno compilare utilizzando SEMPRE le seguenti opzioni:

-Wall → Garantisce la segnalazione di warnings importanti

-Wextra → Garantisce la segnalazione di ulteriori warnings

-std=c11 → Verifica che il codice rispetti lo Standard C11

-D__USE_MINGW_ANSI_STDIO → Garantisce l'uso della libreria stdio.h di gcc*

… in definitiva il comando di compilazione raccomandato diventa

```
>gcc -Wall -Wextra -std=c11 -D__USE_MINGW_ANSI_STDIO welcome.c -o welcome
```

*Nota:
le funzioni di tale libreria sono comunemente utilizzate, ma in pochi sanno che, sotto windows, sono fornite dalla libreria dinamica (MSVCRT.DLL) che non supporta correttamente i dati a 64bit. L'opzione suggerita forza gcc a compilare il sorgente utilizzando la propria libreria statica (aderente allo standard).

Keywords e Struttura del sorgente C

INTRODUZIONE ALLA PROGRAMMAZIONE

CAPITOLO 4

Struttura di un sorgente C

```
/* welcome.c
*   Primo programma in C
*/
#include <stdio.h>   //printf


int main(void)
{
    printf("\nWelcome to C!");

    return 0;
}
```

INTESTAZIONE

Inclusioni e dichiarazioni globali

Funzione main() (*entry point*)

Struttura di un sorgente C

```
/* welcome.c
*   Primo programma in C
*/
#include <stdio.h>   //printf

i          d)
{
                come to C!");

}
```

INTESTAZIONE

Inclusioni e dichiarazioni globali

START

"Welcome to C!"

END

Funzione main() (*entry point*)

Ricapitolando: comandi per il Preprocessore

- **Commenti**
  - In linea → `// bla bla bla ...`
  - Su più righe → `/* bla bla bla ...`
    `bla bla */`
- **Direttive**
  - Inclusioni → `#include <nome_libreria.h>`
  - Macro → `#define ...` (non ancora)

Struttura di un sorgente C

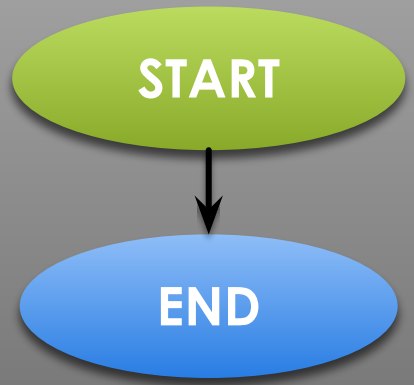
```
/* stupido.c
*  Secondo programma in C
*/

int main(void)
{

    return 0;
}
```

Keywords del C

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Parole riservate del C (**istruzioni**)

```
/* stupido.c
*  Secondo programma in C
*/

int main(void)
{

    return 0;

}
```



Codice d'errore per il SO

Codice	Interpretazione
0	Tutto OK
>0	Qualche errore

Keywords del C

auto	double	**int**	struct
break	else	long	switch
case	enum	register	typedef
char	extern	**return**	union
const	float	short	unsigned
continue	for	signed	**void**
default	goto	sizeof	volatile
do	if	static	while

Funzione di Output: printf ()

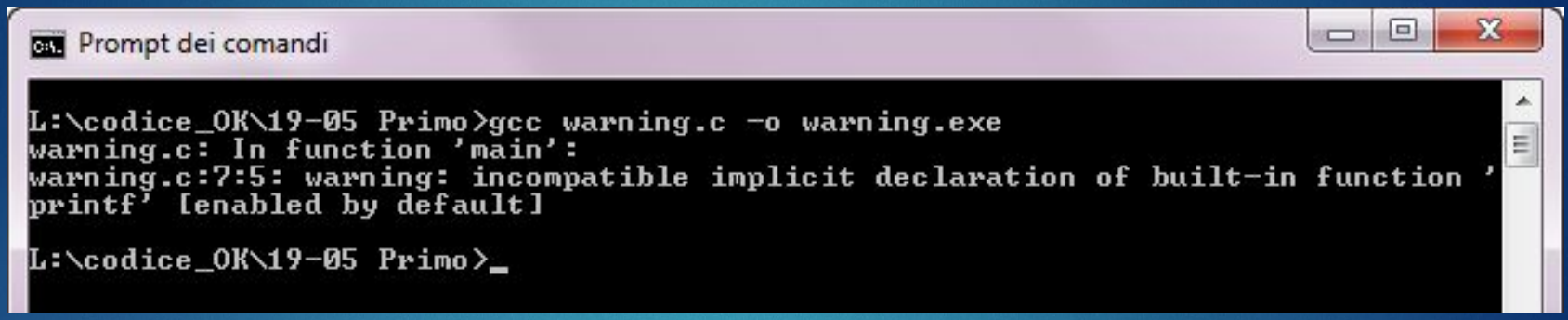
```
/* warning.c
*  Terzo programma in C     //commento su più righe
*/

int main(void)
{
    printf("Hello Word!");    //messaggio di saluto

    printf("\nPosso parlarti…");
    printf("\n…ma ancora non ti sento\n\n");

    printf("Questo\ne'\nun\ntest\n\nHa detto:, \"Come va?\"\n");
    return 0;
}
```

String literal (stringa)

Escape Sequences

Funzione di Output: `printf()`

- Il codice di prima produce un ***warning***:



- «*...implicit declaration...*» cioà qualcosa di implicito, non chiaro: il compilatore si riferisce alla funzione ***`printf`***.
- Abbiamo visto che le funzioni predefinite devono essere «importate» da opportune librerie, come la **`printf`** che si trova in *`stdio.h`*...
- Ma ricontrollando il sorgente ci accorgiamo di aver dimenticato proprio la direttiva **`#include <stdio.h>`** che ci permette di utilizzare la funzione suddetta

Ricapitolando: messaggi del compilatore

Warning

- **Non bloccano** la compilazione
- Il file eseguibile viene prodotto
- **Errori** o ambiguità **semantiche** nel sorgente che potrebbero portare ad errori, crash o bug durante l'esecuzione (spesso così subdoli da sfuggire al Testing)

Error

- **Bloccano** la compilazione
- Il file eseguibile **non** viene prodotto
- **Errori gravi** nel codice che impediscono la traduzione in codice macchina (tipicamente **sintattici**)

Keywords del C

auto	**double**	**int**	struct
break	else	**long**	switch
case	enum	register	typedef
char	extern	**return**	union
const	**float**	**short**	unsigned
continue	for	signed	**void**
default	goto	sizeof	volatile
do	if	static	while