



UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Dispense del corso di Fondamenti di Informatica 1

Il linguaggio C - Preprocessore e compilazione

Aggiornamento del 24/10/2019

I programmi si complicano

- Abbiamo visto che è possibile scrivere una funzione in C che calcoli la radice quadrata di un numero.
- Supponiamo adesso che con pazienza ci mettiamo a scrivere le funzioni per calcolare seno, coseno, tangente, arcoseno, arcocoseno, arcotangente, logaritmo naturale, logaritmo in base arbitraria, esponenziale, ecc...
- Alla fine ci troveremmo facilmente con diverse centinaia di linee di codice C.
- Adesso vogliamo fare un programma che utilizzi una o più di queste funzioni che abbiamo preparato: come procediamo?
- Beh, la soluzione più immediata sarebbe fare copia/incolla dal file in cui abbiamo le funzioni, mettendole nel file in cui dobbiamo utilizzarle.
- Se il programma è però complesso potrebbe avere bisogno di 10 di queste funzioni e quindi otterremmo un programma molto lungo e scomodo da "maneggiare": scroll continuo per andare alle funzioni, difficoltà di identificare che cosa abbiamo scritto di specifico adesso e che cosa invece è un copia incolla da file già stabili e sicuri.

Modifiche difficili

- Inoltre, nell'usare una funzione, potremmo accorgerci di un difetto, oppure potremmo trovare un modo per farla andare più velocemente. Quindi diventerebbe necessario correggerla.
- Questo vorrebbe dire che la correzione dovrebbe essere applicata in tutti i file in cui abbiamo utilizzato quella funzione.
- Chiaramente non è possibile procedere in questo modo.
- Per questo motivo i progettisti del C, hanno introdotto un altro sistema che agisce *prima di effettuare la compilazione*. Questo sistema è il **preprocessore C**.
- Inizialmente questo era un programma separato che veniva invocato prima del compilatore, ma è rapidamente diventato talmente importante da essere integrato direttamente in tutti i compilatori.
- Come funziona?

La direttiva `#include`

- Il preprocessore prevede delle «direttive» che sono comandi specifici per il preprocessore e che vengono sostituite prima ancora della compilazione.
- Ogni direttiva è composta dal simbolo `#` seguito da una parola.
- La prima direttiva che ci interessa è `#include`.
- A questa viene fatto seguire un nome di file indicato tra " " (virgolette doppie) o tra < > (minore e maggiore).
 - L'unica differenza è che nel primo caso la ricerca viene fatta anche nella cartella corrente, nell'altro caso solo nei percorsi che contengono le librerie del compilatore.
- Che cosa fa questa direttiva? È come se il programma aprisse il file richiesto, ne copiasse il contenuto e lo incollasse al posto della riga che contiene `#include`.
- Il bello è che questo non viene fatto permanentemente, ma solo per la compilazione. Quindi il file rimane uguale a prima.

Esempio

- Sia dato questo file:

main.c

```
double sqrt (double a) {  
    double t, x = a;  
  
    if (x<=0.)  
        return 0.;  
  
    do {  
        t = x;  
        x = 0.5*(t + a/t);  
    } while (x!=t);  
  
    return x;  
}  
  
int main(void) {  
    double d = sqrt(137.0);  
  
    return 0;  
}
```

Esempio

- Possiamo dividerlo così:

main.c

```
#include "radice.c"

int main(void) {
    double d = sqrt(137.0);

    return 0;
}
```

radice.c

```
double sqrt (double a) {
    double t, x = a;

    if (x<=0.)
        return 0.;

    do {
        t = x;
        x = 0.5*(t + a/t);
    } while (x!=t);

    return x;
}
```

- In questo modo, il nostro programma, quando compiliamo `main.c`, è esattamente uguale a prima, ma il file è molto più "pulito".
- Inoltre qualsiasi modifica fatta nel file `radice.c` viene automaticamente riportata anche durante la compilazione di `main.c`

Ancora problemi

- Purtroppo quando le cose si complicano, sorgono altre difficoltà.
- In particolare, in progetti software molto complessi, potrebbero essere disponibili e necessarie ad esempio 1000 funzioni, anche molto sofisticate, divise in 10 file con 100 funzioni l'uno.
- Diciamo che per compilare una di queste funzioni sia necessario 1 s.
- In un programma si utilizza una sola funzione presa da ogni file ed è presente una funzione main che richiede 1 s per essere compilata.
- Quanto tempo richiede la compilazione del programma?

Ancora problemi

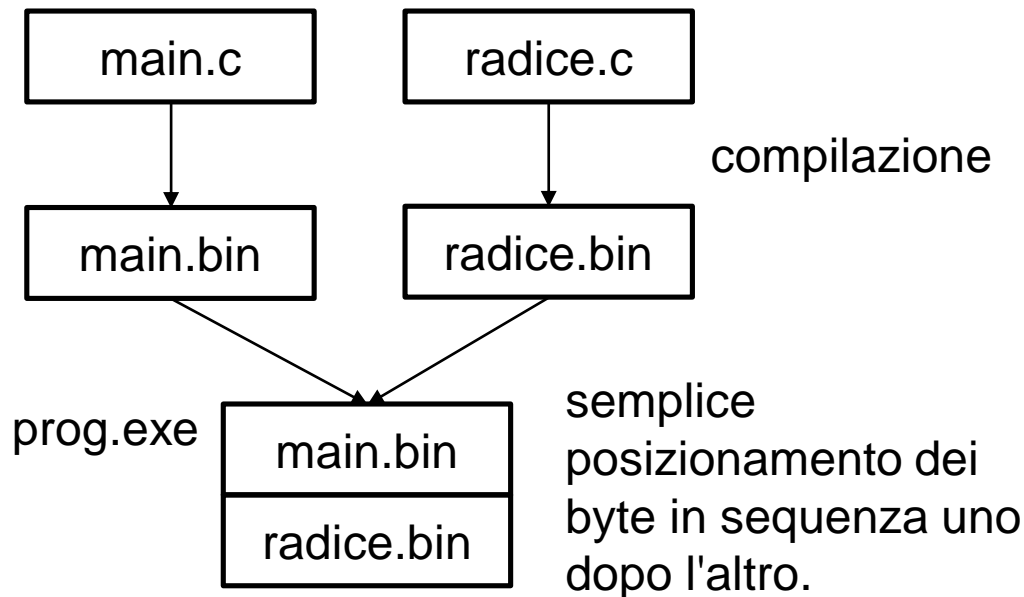
- Purtroppo quando le cose si complicano, sorgono altre difficoltà.
- In particolare, in progetti software molto complessi, potrebbero essere disponibili e necessarie ad esempio 1000 funzioni, anche molto sofisticate, divise in 10 file con 100 funzioni l'uno.
- Diciamo che per compilare una di queste funzioni sia necessario 1 s.
- In un programma si utilizza una sola funzione presa da ogni file ed è presente una funzione main che richiede 1 s per essere compilata.
- Quanto tempo richiede la compilazione del programma?
- Se facciamo il copia/incolla delle 10 funzioni, servono 11 s.

Ancora problemi

- Purtroppo quando le cose si complicano, sorgono altre difficoltà.
- In particolare, in progetti software molto complessi, potrebbero essere disponibili e necessarie ad esempio 1000 funzioni, anche molto sofisticate, divise in 10 file con 100 funzioni l'uno.
- Diciamo che per compilare una di queste funzioni sia necessario 1 s.
- In un programma si utilizza una sola funzione presa da ogni file ed è presente una funzione main che richiede 1 s per essere compilata.
- Quanto tempo richiede la compilazione del programma?
- Se facciamo il copia/incolla delle 10 funzioni, servono 11 s.
- Se facciamo l'`#include` dei 10 file, servono 1001 s, ovvero 16 minuti e 40 secondi!
- Qual è il problema?
- Ogni volta che includiamo un file in questo modo, dobbiamo compilare tutte le funzioni presenti nel file, **anche se non le usiamo!**

Come risolvere?

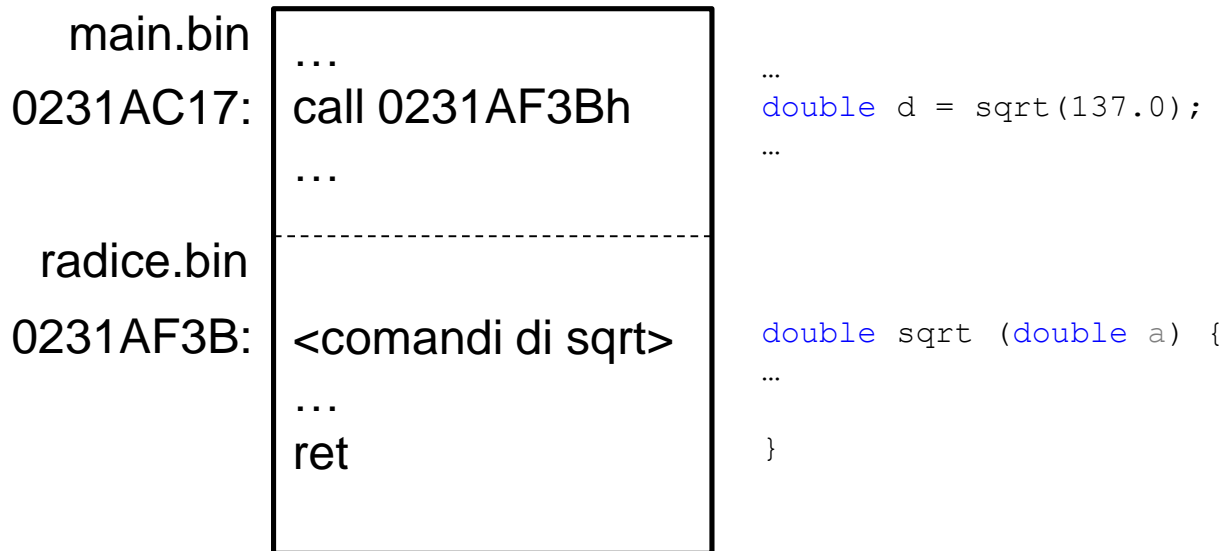
- È evidente che la soluzione che utilizza l'#include non è ragionevole.
- Serve qualcosa di diverso che permetta di **non ricompilare il codice già compilato**.
- Ma come è possibile? Non ci sarebbero problemi se i due "pezzi" di programma in linguaggio macchina fossero indipendenti:



- In questo modo, potremmo compilare solo `main.c` e attaccare `radice.bin` al codice macchina così ottenuto.

Non funziona!

- Purtroppo la soluzione proposta suppone che i due blocchi siano indipendenti, ma se lo fossero non avremmo bisogno di usarli insieme!
- Infatti (gli indirizzi sono un esempio a caso):



- In pratica, per collegare assieme i due pezzi di programma in linguaggio macchina dovremmo sapere a che indirizzo si trova la funzione sqrt. Ma noi non lo sappiamo fino a che non li colleghiamo assieme!
- Quindi?!

File Oggetto

- La soluzione di questo problema richiede un passaggio intermedio, ovvero prima di generare il file finale, generare dei file intermedi in cui i riferimenti esterni non siano già dei numeri definiti, ma solo dei "segnaposto".
- Questi file intermedi si chiamano file "oggetto" o *object file*.
- La loro struttura in casi reali segue dei formati standard ben definiti, come ad esempio COFF o ELF.
- Semplificando al massimo, bisogna che nel file ci siano tre informazioni:
 - Il codice macchina con le istruzioni e i dati (codice). Se serve far riferimento ad un indirizzo esterno, questo viene lasciato a 0.
 - L'elenco delle posizioni nel file, associate al *nome*, in cui è possibile trovare funzioni o variabili globali definite in questo file (export).
 - L'elenco delle posizioni nel file che abbiamo lasciato a zero, associato al *nome* della funzione o variabile globale a cui si voleva fare riferimento (import).
- Facciamo un esempio sempre con riferimento al caso precedente, usando un formato inventato (ipotizziamo indirizzi a 16 bit in big endian).

File Oggetto

- Consideriamo il file main.c: in questo file definiamo una sola funzione (main) e facciamo riferimento ad una sola funzione (sqrt).

main.obj	
codice	CD 6D F5 9E 56 9C CF A7 D5 03 03 4F 46 99 15 58 43 17 AC 96 10 D3 01 AB F3 1D B0 6A DD 07 56 A5 94 8C 93 00 00 67 58 E4 39 5A A3 B7 F6 A1 F5 81 4E AE FC E1 A4 99 8E BB 90 75 AF 43 56 83 D7 2B
export	main 0010
import	sqrt 0023

File Oggetto

- Consideriamo il file main.c: in questo file definiamo una sola funzione (main) e facciamo riferimento ad una sola funzione (sqrt).

main.obj	
codice	CD 6D F5 9E 56 9C CF A7
	D5 03 03 4F 46 99 15 58
	43 17 AC 96 10 D3 01 AB
	F3 1D B0 6A DD 07 56 A5
	94 8C 93 00 00 67 58 E4
	39 5A A3 B7 F6 A1 F5 81
	4E AE FC E1 A4 99 8E BB
	90 75 AF 43 56 83 D7 2B
export	main 0010
import	sqrt 0023

Qui inizia il main

File Oggetto

- Consideriamo il file main.c: in questo file definiamo una sola funzione (main) e facciamo riferimento ad una sola funzione (sqrt).

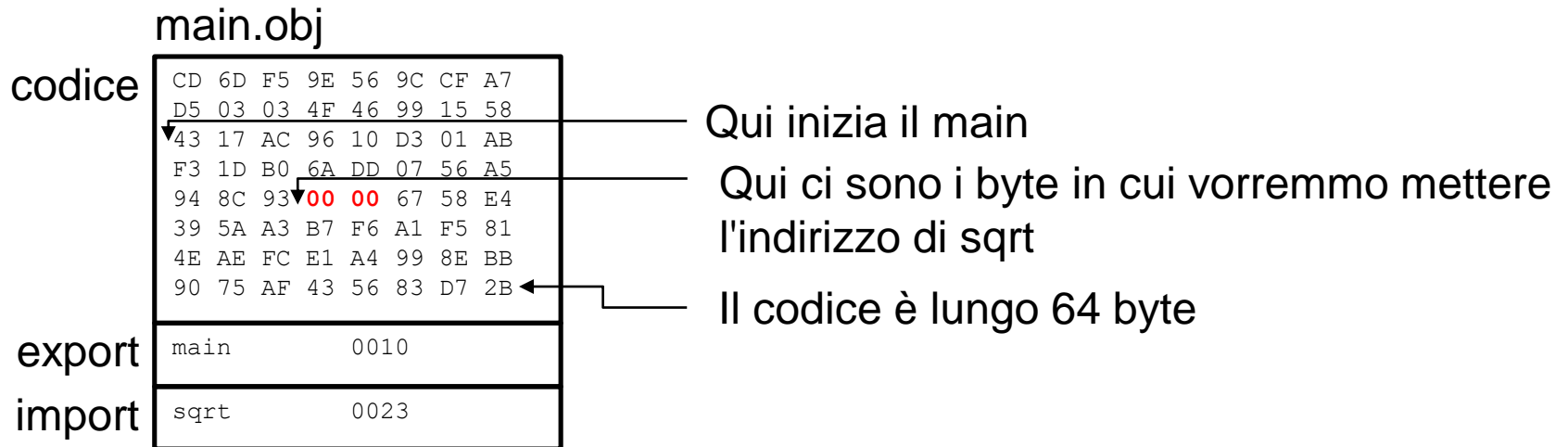
main.obj	
codice	CD 6D F5 9E 56 9C CF A7
	D5 03 03 4F 46 99 15 58
	43 17 AC 96 10 D3 01 AB
	F3 1D B0 6A DD 07 56 A5
	94 8C 93 00 00 67 58 E4
	39 5A A3 B7 F6 A1 F5 81
	4E AE FC E1 A4 99 8E BB
	90 75 AF 43 56 83 D7 2B
export	main 0010
import	sqrt 0023

Qui inizia il main

Qui ci sono i byte in cui vorremmo mettere l'indirizzo di sqrt

File Oggetto

- Consideriamo il file main.c: in questo file definiamo una sola funzione (main) e facciamo riferimento ad una sola funzione (sqrt).



ipotizziamo indirizzi a 16 bit in big endian

File Oggetto

- Facciamo la stessa cosa anche con radice.c: in questo file definiamo una sola funzione (sqrt).

	main.obj															
codice	CD 6D F5 9E 56 9C CF A7															
	D5 03 03 4F 46 99 15 58															
	43 17 AC 96 10 D3 01 AB															
	F3 1D B0 6A DD 07 56 A5															
	94 8C 93 00 00 67 58 E4															
	39 5A A3 B7 F6 A1 F5 81															
	4E AE FC E1 A4 99 8E BB															
	90 75 AF 43 56 83 D7 2B															
export	main 0010															
import	sqrt 0023															

	radice.obj															
codice	D6 31 6E C2 B4 8C 9E 3F															
	C5 C7 E4 80 4B E4 94 9A															
	51 83 86 D1 B5 0A 7C 6B															
	8F C0 4A 11 4F C5 CB A7															
	78 2A 06 56 5F C5 37 E4															
	3A 74 EA AB F5 A8 CE A2															
	5A 08 29 2A A4 1C 69 B5															
	46 4A 53 A2 47 A4 7C 8D															
	B4 C9 88 B6 44 A4 CC EF															
	89 0A AC 79 CE 2E 7F 6B															
	5D 5A 2B 04 CA 0A 49 6A															
	CA 5B AC E8 92 12 6E E5															
	39 05 99 D2 EB 7E 8B 3D															
	C9 76 37 2B 49 81 0D 49															
export	sqrt 000A															
import																

ipotizziamo indirizzi a 16 bit in big endian

File Oggetto

- Facciamo la stessa cosa anche con radice.c: in questo file definiamo una sola funzione (sqrt).

	main.obj							
codice	CD	6D	F5	9E	56	9C	CF	A7
	D5	03	03	4F	46	99	15	58
	43	17	AC	96	10	D3	01	AB
	F3	1D	B0	6A	DD	07	56	A5
	94	8C	93	00	00	67	58	E4
	39	5A	A3	B7	F6	A1	F5	81
	4E	AE	FC	E1	A4	99	8E	BB
	90	75	AF	43	56	83	D7	2B
export	main				0010			
import	sqrt				0023			

	radice.obj																
codice	D6	31	6E	C2	B4	8C	9E	3F									
	C5	C7	E4	80	4B	E4	94	9A									
	51	83	86	D1	B5	0A	7C	6B									
	8F	C0	4A	11	4F	C5	CB	A7									
	78	2A	06	56	5F	C5	37	E4									
	3A	74	EA	AB	F5	A8	CE	A2									
	5A	08	29	2A	A4	1C	69	B5									
	46	4A	53	A2	47	A4	7C	8D									
	B4	C9	88	B6	44	A4	CC	EF									
	89	0A	AC	79	CE	2E	7F	6B									
	5D	5A	2B	04	CA	0A	49	6A									
	CA	5B	AC	E8	92	12	6E	E5									
	39	05	99	D2	EB	7E	8B	3D									
	C9	76	37	2B	49	81	0D	49									
	export	sqrt		000A													
	import																

Qui inizia la funzione sqrt

ipotizziamo indirizzi a 16 bit in big endian

Il collegamento (linking)

- Per collegare i file dovremo mettere i due blocchi di codice in sequenza:

CD	6D	F5	9E	56	9C	CF	A7
D5	03	03	4F	46	99	15	58
43	17	AC	96	10	D3	01	AB
F3	1D	B0	6A	DD	07	56	A5
94	8C	93	00	00	67	58	E4
39	5A	A3	B7	F6	A1	F5	81
4E	AE	FC	E1	A4	99	8E	BB
90	75	AF	43	56	83	D7	2B

D6	31	6E	C2	B4	8C	9E	3F
C5	C7	E4	80	4B	E4	94	9A
51	83	86	D1	B5	0A	7C	6B
8F	C0	4A	11	4F	C5	CB	A7
78	2A	06	56	5F	C5	37	E4
3A	74	EA	AB	F5	A8	CE	A2
5A	08	29	2A	A4	1C	69	B5
46	4A	53	A2	47	A4	7C	8D
B4	C9	88	B6	44	A4	CC	EF
89	0A	AC	79	CE	2E	7F	6B
5D	5A	2B	04	CA	0A	49	6A
CA	5B	AC	E8	92	12	6E	E5
39	05	99	D2	EB	7E	8B	3D
C9	76	37	2B	49	81	0D	49

main.obj		
export	main	0010
import	sqrt	0023

radice.obj		
export	sqrt	000A
import		

Il collegamento (linking)

- Per collegare i file dovremo mettere i due blocchi di codice in sequenza:

CD	6D	F5	9E	56	9C	CF	A7
D5	03	03	4F	46	99	15	58
43	17	AC	96	10	D3	01	AB
F3	1D	B0	6A	DD	07	56	A5
94	8C	93	00	00	67	58	E4
39	5A	A3	B7	F6	A1	F5	81
4E	AE	FC	E1	A4	99	8E	BB
90	75	AF	43	56	83	D7	2B

D6	31	6E	C2	B4	8C	9E	3F
C5	C7	E4	80	4B	E4	94	9A
51	83	86	D1	B5	0A	7C	6B
8F	C0	4A	11	4F	C5	CB	A7
78	2A	06	56	5F	C5	37	E4
3A	74	EA	AB	F5	A8	CE	A2
5A	08	29	2A	A4	1C	69	B5
46	4A	53	A2	47	A4	7C	8D
B4	C9	88	B6	44	A4	CC	EF
89	0A	AC	79	CE	2E	7F	6B
5D	5A	2B	04	CA	0A	49	6A
CA	5B	AC	E8	92	12	6E	E5
39	05	99	D2	EB	7E	8B	3D
C9	76	37	2B	49	81	0D	49

main.obj		
export	main	0010
import	sqrt	0023

radice.obj		
export	sqrt	000A
import		

- A seguito di questa operazione l'indirizzo di sqrt è cambiato. Però il cambiamento è facilmente calcolabile: basta sommare al suo indirizzo la dimensione del codice precedente, ovvero 64 (40 in esadecimale).

ipotizziamo indirizzi a 16 bit in big endian

Il collegamento (linking)

- Uniamo allora la tabella degli export e degli import:

CD	6D	F5	9E	56	9C	CF	A7
D5	03	03	4F	46	99	15	58
43	17	AC	96	10	D3	01	AB
F3	1D	B0	6A	DD	07	56	A5
94	8C	93	00	00	67	58	E4
39	5A	A3	B7	F6	A1	F5	81
4E	AE	FC	E1	A4	99	8E	BB
90	75	AF	43	56	83	D7	2B

D6	31	6E	C2	B4	8C	9E	3F
C5	C7	E4	80	4B	E4	94	9A
51	83	86	D1	B5	0A	7C	6B
8F	C0	4A	11	4F	C5	CB	A7
78	2A	06	56	5F	C5	37	E4
3A	74	EA	AB	F5	A8	CE	A2
5A	08	29	2A	A4	1C	69	B5
46	4A	53	A2	47	A4	7C	8D
B4	C9	88	B6	44	A4	CC	EF
89	0A	AC	79	CE	2E	7F	6B
5D	5A	2B	04	CA	0A	49	6A
CA	5B	AC	E8	92	12	6E	E5
39	05	99	D2	EB	7E	8B	3D
C9	76	37	2B	49	81	0D	49

export	main	0010
	sqrt	004A
import	sqrt	0023

- Resta solo da risolvere quell'import sostituendo alla posizione 0023 l'indirizzo di sqrt, ovvero 004A.

Il collegamento (linking)

- Questo è il risultato finale:

CD	6D	F5	9E	56	9C	CF	A7
D5	03	03	4F	46	99	15	58
43	17	AC	96	10	D3	01	AB
F3	1D	B0	6A	DD	07	56	A5
94	8C	93	00	4A	67	58	E4
39	5A	A3	B7	F6	A1	F5	81
4E	AE	FC	E1	A4	99	8E	BB
90	75	AF	43	56	83	D7	2B

D6	31	6E	C2	B4	8C	9E	3F
C5	C7	E4	80	4B	E4	94	9A
51	83	86	D1	B5	0A	7C	6B
8F	C0	4A	11	4F	C5	CB	A7
78	2A	06	56	5F	C5	37	E4
3A	74	EA	AB	F5	A8	CE	A2
5A	08	29	2A	A4	1C	69	B5
46	4A	53	A2	47	A4	7C	8D
B4	C9	88	B6	44	A4	CC	EF
89	0A	AC	79	CE	2E	7F	6B
5D	5A	2B	04	CA	0A	49	6A
CA	5B	AC	E8	92	12	6E	E5
39	05	99	D2	EB	7E	8B	3D
C9	76	37	2B	49	81	0D	49

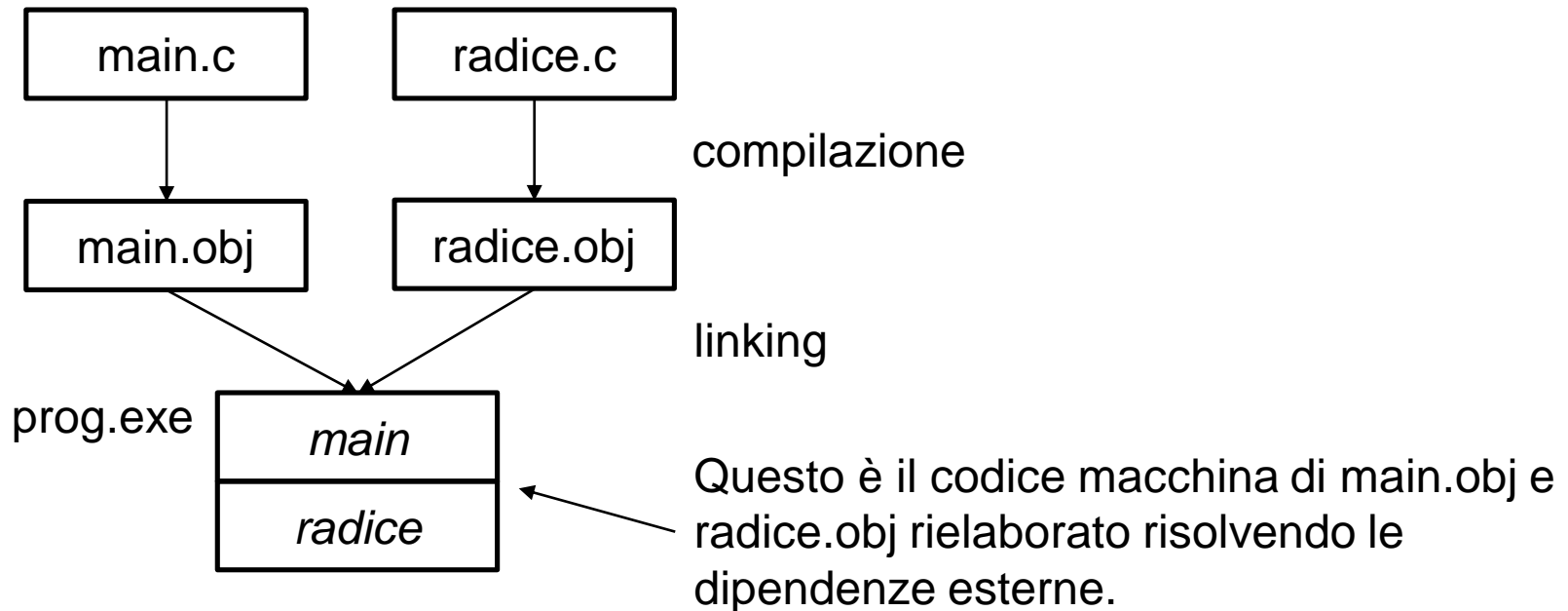
export	main	0010
	sqrt	004A
import	sqrt	0023

- Il programma che fa questa operazione si chiama appunto *linker*.
- Quale diventa allora il processo completo?

ipotizziamo indirizzi a 16 bit in big endian

Processo di compilazione e linking

- Il processo completo diventa quindi il seguente:



- In questo modo abbiamo trovato una soluzione interessante che potrebbe permetterci di compilare una sola volta `radice.c` e poi successivamente compilare `main.c`, da solo, più volte **senza ricompilare `radice.c`**.
- Siamo a posto?

Identificatori non definiti

- Purtroppo non è finita. Infatti abbiamo detto che vorremmo compilare `main.c` e `radice.c` ognuno indipendentemente, quindi i file dovrebbero essere così:

main.c

```
int main(void) {  
    double d = sqrt(137.0);  
  
    return 0;  
}
```

radice.c

```
double sqrt (double a) {  
    double t, x = a;  
  
    if (x<=0.)  
        return 0.;  
  
    do {  
        t = x;  
        x = 0.5*(t + a/t);  
    } while (x!=t);  
  
    return x;  
}
```

- La compilazione di `radice.c` non crea alcun problema, ma quando compiliamo `main.c` il compilatore trova la chiamata alla funzione `sqrt`, ma non sa che cosa sia e quindi ci segnala che questo identificatore è sconosciuto.
- Non abbiamo infatti definito `sqrt`. Come risolviamo?

La dichiarazione

- Dobbiamo introdurre (finalmente) il concetto di *dichiarazione*.
- Con dichiarazione si intende un modo per dire al compilatore che un identificatore (variabile o funzione) esiste e di che tipo è.
- Per fare questo utilizziamo la parola chiave *extern*, che segnala al compilatore che quell'identificatore verrà definito successivamente (in questo o in un altro file). Per le funzioni inoltre, invece che fornire il codice della funzione tra parentesi graffe, terminiamo la dichiarazione con un punto e virgola.
- Esempio di dichiarazione di una variabile x di tipo int:

```
extern int x;
```

- Esempio di dichiarazione di una funzione frazione che accetta due int e restituisce un float:

```
extern float frazione(int a, int b);
```

La dichiarazione

- Mentre per le variabili la parola chiave `extern` è l'unico modo di far capire che non stiamo definendo la variabile, ma solo dichiarandola, per le funzioni la differenza è resa evidente dalla mancanza del blocco di comandi: pertanto l'uso di `extern` è facoltativo.
- **In questo corso però lo utilizzeremo sempre** per sottolineare la natura analoga delle dichiarazioni di variabili e funzioni.
- Inoltre nella dichiarazione di funzioni non serve fare riferimento ai parametri (infatti manca il corpo della funzione). L'unica cosa importante è conoscerne il tipo. Il nome dei parametri può anche essere omesso, ma **in questo corso e anche nella pratica comune questo è da evitare**.
- Sono allora equivalenti le seguenti dichiarazioni:

```
extern float fraz(int a, int b);  
float fraz(int a, int b);  
extern float fraz(int, int);  
float fraz(int, int);
```

← Noi scriveremo
sempre così

L'esempio corretto

- Possiamo allora finalmente avere una versione corretta dell'esempio precedente:

main.c

```
extern double sqrt (double a);

int main(void) {
    double d = sqrt(137.0);

    return 0;
}
```

radice.c

```
double sqrt (double a) {
    double t, x = a;

    if (x<=0.)
        return 0.;

    do {
        t = x;
        x = 0.5*(t + a/t);
    } while (x!=t);

    return x;
}
```

- In questo modo quando compiliamo main.c il compilatore trova la dichiarazione della funzione sqrt e sa che cos'è, quali parametri accetta e quale tipo di dato ritorna.
- Quando incontra la chiamata, può indicare nel file oggetto che gli serve l'indirizzo di sqrt per poterla chiamare.
- Finito?

Ancora difficoltà e copia/incolla

- Il nostro esempio è un caso semplice, ma se torniamo al caso più complesso ipotizzato prima (10 file con 100 funzioni l'uno e un main che ne utilizza 10) vediamo che ci tocca scrivere le 10 dichiarazioni delle funzioni.
- Se poi usiamo le stesse funzioni da un'altra parte, dovremo fare il copia/incolla delle dichiarazioni. Se ne utilizziamo altre 10, dovremo riscrivere le dichiarazioni anche di quelle.
- Serve un meccanismo più comodo, che ci permetta di scrivere le dichiarazioni una volta sola.
- Ma il meccanismo per fare questo ce l'abbiamo già! È la direttiva del preprocessore `#include`.
- Quello che possiamo fare è creare accanto al file con le definizioni, un file in cui inseriamo tutte le dichiarazioni.
- La prassi prevede che per ogni file `.c` che deve fungere da libreria, si crei un file `.h` con lo stesso nome che contiene le corrispondenti dichiarazioni.
- Per utilizzare le funzioni così definite e dichiarate, si può semplicemente includere il file `.h` e linkare assieme tutti i `.c`

L'esempio corretto con #include

- Sempre con l'esempio di prima, si potrebbe fare così:

main.c

```
#include "radice.h"

int main(void) {
    double d = sqrt(137.0);

    return 0;
}
```

radice.h

```
extern double sqrt (double a);
```

radice.c

```
double sqrt (double a) {
    double t, x = a;

    if (x<=0.)
        return 0.;

    do {
        t = x;
        x = 0.5*(t + a/t);
    } while (x!=t);

    return x;
}
```

- Se nel file radice.c aggiungessimo la definizione di altre funzioni matematiche (ad esempio la radice cubica e la radice con indice arbitrario), sarebbe sufficiente inserire le corrispondenti dichiarazioni in radice.h e in main.c potremmo utilizzarle senza dover fare altro.

Le macro

- Il preprocessore del C contiene un'altra utile funzionalità che viene spesso utilizzata durante la scrittura di codice C.
- Questa funzionalità prende il nome di *macro* e consiste nella possibilità di definire delle porzioni di testo che debbono essere sostituite ovunque nel codice, ovvero una versione automatizzata del comando *Sostituisci tutto* presente negli editor di testo.
- Esistono due versioni di macro, senza parametri e con parametri:

```
#define <identificatore> <testo da sostituire>
```

```
#define <identificatore>(<lista di parametri>) <testo da sostituire>
```
- È importante notare che tra l'identificatore e la parentesi aperta, nel secondo caso, **non deve essere presente alcuno spazio**, altrimenti tutto il resto della riga diventa il testo da sostituire.
- Le macro infatti terminano normalmente alla fine della riga. Se si desidera continuare la macro alla riga successiva bisogna aggiungere il carattere `\` prima del ritorno a capo.

Esempio di uso del #define

- Come si può utilizzare il #define?
- Ad esempio per dare un significato a valori numerici comuni che non necessitano esplicitamente di un tipo:

```
#define TRUE 1
#define FALSE 0
...
int finito = FALSE;
while (finito != TRUE) {
    ...
}
```

- In questo esempio, si utilizza il #define per evitare di usare direttamente i valori numerici 1 e 0, chiarendo che lo 0 con cui si inizializza la variabile finito serve a indicare la condizione di falso e che il ciclo continua fino a che finito è diverso da vero.
- Di che tipo è FALSE?

Esempio di uso del #define

- Come si può utilizzare il #define?
- Ad esempio per dare un significato a valori numerici comuni che non necessitano esplicitamente di un tipo:

```
#define TRUE 1
#define FALSE 0
...
int finito = FALSE;
while (finito != TRUE) {
    ...
}
```

- In questo esempio, si utilizza il #define per evitare di usare direttamente i valori numerici 1 e 0, chiarendo che lo 0 con cui si inizializza la variabile finito serve a indicare la condizione di falso e che il ciclo continua fino a che finito è diverso da vero.
- Di che tipo è FALSE?
- Di nessun tipo. Non è una variabile e non è una espressione. Non viene neppure compilato. Semplicemente, prima di compilare, tutte le occorrenze di FALSE vengono sostituite con il testo "0".

Macro con parametri

- Come si utilizzano le macro con parametri? Un esempio può essere il seguente:

```
#define QUADRATO(x) x*x
```

- In questa macro è necessario specificare un parametro (cioè un pezzo di testo) che verrà sostituito a tutte le occorrenze di x. Quindi se scrivo:

```
int a = QUADRATO(3);
```

questo viene sostituito con il testo:

```
int a = 3*3;
```

- Se scrivo:

```
int a = QUADRATO(3+1);
```

con che valore viene inizializzato a?

Macro con parametri

- Come si utilizzano le macro con parametri? Un esempio può essere il seguente:

```
#define QUADRATO(x) x*x
```

- In questa macro è necessario specificare un parametro (cioè un pezzo di testo) che verrà sostituito a tutte le occorrenze di x. Quindi se scrivo:

```
int a = QUADRATO(3);
```

questo viene sostituito con il testo:

```
int a = 3*3;
```

- Se scrivo:

```
int a = QUADRATO(3+1);
```

con che valore viene inizializzato a? **Con 7!** Perché?

Macro con parametri

- Come si utilizzano le macro con parametri? Un esempio può essere il seguente:

```
#define QUADRATO(x) x*x
```

- In questa macro è necessario specificare un parametro (cioè un pezzo di testo) che verrà sostituito a tutte le occorrenze di x. Quindi se scrivo:

```
int a = QUADRATO(3);
```

questo viene sostituito con il testo:

```
int a = 3*3;
```

- Se scrivo:

```
int a = QUADRATO(3+1);
```

con che valore viene inizializzato a? **Con 7!** Perché?

- Perché le macro effettuano una sostituzione di **testo**. Quello che scrivo tra parentesi viene sostituito nella parte successiva, quindi:

```
int a = 3+1*3+1;
```

- Questo però non è quello che volevamo. Come risolvo?

Macro con parametri

- Nelle macro è importante **indicare sempre i parametri tra parentesi tonde** ad ogni loro utilizzo:

```
#define QUADRATO(x) (x) * (x)
```

- In questo modo l'esempio precedente diventa:

```
int a = QUADRATO(3+1);
```

- tradotto come

```
int a = (3+1) * (3+1);
```

- Che effettivamente corrisponde a quanto intendevamo.
- Ci sono altri problemi però:

```
int a = QUADRATO(f(x));
```

```
int a = QUADRATO(++x);
```

- Nel primo caso la funzione viene invocata due volte e nel secondo caso si genera un undefined behavior!
- Insomma, le macro con parametri nascevano per avere semplici simil-funzioni che evitassero la chiamata a funzione, perché sostituite prima della compilazione. Ma oggi i compilatori sono più che bravi e quindi **in questo corso eviteremo di usare le macro con parametri.**

Altre funzionalità del preprocessore

- Il preprocessore del C non si limita a fornire la funzionalità di `#include`, ma può essere uno strumento molto potente per aiutare il programmatore.
- Supponiamo di avere una serie di funzioni che nel sistema operativo Windows sono definite nel file `mialib_win32.h`, mentre sotto Linux sono in `mialib_unix.h`. Per compilare sotto Windows, il programma avrà una struttura di questo tipo:

main.c

```
#include <mialib_win32.h>

int main(void) {
    ...
    return 0;
}
```

- Sotto Linux invece:

main.c

```
#include <mialib_unix.h>

int main(void) {
    ...
    return 0;
}
```

Compilazione condizionale

- È chiaro che se si tratta di un solo file, nel compilare sui due sistemi, non è un gran problema cambiare a mano quella riga.
- Però se il progetto è composto di 50 file e tutti usano qualcosa di specifico a seconda del sistema operativo e ogni file fa qualcosa di diverso, ad ogni nuova versione sarebbe necessario rimodificare tutti e 50 i file.
- Per questo ed altri casi, il preprocessore mette a disposizione delle direttive per la *compilazione condizionale*.
- Ovvero, delle direttive che consentono di selezionare che cosa viene lasciato nel file e che cosa no, al termine della passata del preprocessore.
- La sintassi è di questo tipo:

```
#if <condizione>
```

```
<righe di testo>
```

```
#else
```

```
<altre righe di testo>
```

```
#endif
```

Compilazione condizionale

- Quali condizioni si possono utilizzare? Oltre a tutte quelle che si possono scrivere in C (però basate solo su macro del preprocessore e ovviamente non su variabili) è disponibile anche un operatore *defined* (attenzione alla d finale!) che vale 1 se una macro è stata definita.
- È infatti possibile definire delle macro senza specificare il testo da sostituire.
- Facciamo un esempio (con il caso precedente) tanto per chiarire il concetto:

main.c

```
#if defined _WIN32
    #include <mialib_win32.h>
#else
    #include <mialib_unix.h>
#endif

int main(void) {
    ...
    return 0;
}
```

- In questo caso, abbiamo un solo file che prevede l'inclusione della libreria mialib_win32.h se è stata definita la macro _WIN32 (Visual Studio definisce di default questa macro), oppure mialib_unix.h in caso contrario.
- In questo modo, con un solo file potremo gestire le diverse possibilità.

Compilazione condizionale

- E se avessimo bisogno di una versione della libreria anche per Mac? Potremmo scrivere:

main.c

```
#if defined _WIN32
    #include <mialib_win32.h>
#else
    #if defined __APPLE__
        #include <mialib_mac.h>
    #else
        #include <mialib_unix.h>
    #endif
#endif

int main(void) {
    ...
    return 0;
}
```

- Le cose diventano parecchio lunghe in questo modo. Per verificare una ulteriore condizione il preprocessore mette a disposizione la direttiva `#elif` che significa `else+if`.
- Inoltre esiste la direttiva `#error` per segnalare **durante la compilazione** che qualche condizione non è rispettata.

Compilazione condizionale

- Il caso precedente potrebbe essere scritto come:

main.c

```
#if defined _WIN32
    #include <mialib_win32.h>
#elif defined __APPLE__
    #include <mialib_mac.h>
#elif defined __unix__
    #include <mialib_unix.h>
#else
    #error "Compilatore non supportato."
#endif

int main(void) {
    ...
    return 0;
}
```

- Notate che i messaggi di errore vanno indicati tra doppi apici.
- E se volessi inserire del testo nel caso `__unix__` **non sia definito?**
- Posso scrivere:

```
#if !defined __unix__
    ... quello che voglio ...
#endif
```

Compilazione condizionale

- Esistono anche due direttive "vecchie", che sono esattamente uguali a `#if defined` e `#if !defined`: `#ifdef` e `#ifndef`.
- Gli esempi precedenti potrebbero essere:

```
#ifdef _WIN32
    #include <mialib_win32.h>
#elif __APPLE__
    #include <mialib_mac.h>
#elif __unix__
    #include <mialib_unix.h>
#else
    #error "Compilatore non supportato."
#endif

#ifndef __unix__
    int x = 7;
#endif
```

- Notate che accanto ad `#elif` non è indispensabile l'uso di `defined` (è sottinteso).
- Usate quelle che preferite, ma per coerenza in questo corso useremo sempre la versione più "nuova": `#if defined`.

A che cosa può servire a noi?

- Al di là della sua indubbia utilità per progetti software complessi e multiplatforma, la compilazione condizionale ha un utilizzo molto frequente e concreto in ogni progetto C.
- In particolare c'è un problema a cui non abbiamo ancora accennato, dovuto al funzionamento del meccanismo di `#include`.
- Quando includiamo un file, stiamo ricopiando le righe contenute nel file incluso nel nostro file. Che cosa accade se le inseriamo due volte?
- Per ora nulla, visto che la dichiarazione di variabili e funzioni non presenta alcun problema se ripetuta più volte. Però non è sempre così.
- Esistono alcuni elementi del C, che vedremo successivamente, che non possono essere definiti più volte.
- In generale comunque è inutile definire più volte le stesse cose e rallenta solo la compilazione.
- La soluzione sarebbe includere i file una sola volta, ma la cosa è più sofisticata di quanto non sembri, infatti un file potrebbe essere incluso in un altro file che includiamo successivamente! Come risolviamo?

L'include guard (1)

- Riprendiamo l'esempio della radice. Il file radice.h, se incluso più volte, dichiara la funzione sqrt più volte (non è un problema, ma per altri tipi di dichiarazioni lo è...):

main.c

```
#include "radice.h"

int main(void) {
    double d = sqrt(137.0);

    return 0;
}
```

radice.h

```
extern double sqrt (double a);
```

radice.c

```
double sqrt (double a) {
    double t, x = a;

    if (x<=0.)
        return 0.;

    do {
        t = x;
        x = 0.5*(t + a/t);
    } while (x!=t);


    return x;
}
```

- Come possiamo fare a garantire l'inclusione una volta sola?
- Potremmo verificare se una certa macro è definita e se lo è non inserire nulla.
- Quale macro? Diciamo di usare il nome del file tutto in maiuscolo, sostituendo i punti con degli _

L'include guard (2)

- Possiamo scrivere così:

<u>main.c</u>	<u>radice.h</u>	<u>radice.c</u>
<pre>#include "radice.h" int main(void) { double d = sqrt(137.0); return 0; }</pre>	<pre>#if !defined RADICE_H extern double sqrt(double a); #endif /* RADICE_H */</pre>	<pre>double sqrt (double a) { double t, x = a; if (x<=0.) return 0.; do { t = x; x = 0.5*(t + a/t); } while (x!=t); return x; }</pre>



Esempio parziale! Non fermatevi qui! Andate alla pagina successiva!

- In questo modo, la dichiarazione di sqrt viene inclusa solo se non è definita la macro RADICE_H.
- Ma quando si definisce questa macro? Bisogna definirla la prima volta che viene incluso il file. Quindi possiamo definirla all'interno del blocco #if/#endif.

L'include guard (3)

- L'include guard completo diventa quindi:

main.c

```
#include "radice.h"

int main(void) {
    double d = sqrt(137.0);

    return 0;
}
```

radice.h

```
#if !defined RADICE_H
#define RADICE_H

extern double sqrt(double a);

#endif /* RADICE_H */
```

radice.c

```
double sqrt (double a) {
    double t, x = a;

    if (x<=0.)
        return 0.;

    do {
        t = x;
        x = 0.5*(t + a/t);
    } while (x!=t);

    return x;
}
```

- Con questa soluzione, se anche includessimo due volte radice.h, il suo contenuto verrebbe copiato solo la prima volta.
- Notate il commento accanto all'#endif, che non ha alcuna funzione pratica, ma che fa ben capire a chi si riferisce quella direttiva. È buona norma metterlo per chiarezza.
- Resta un'ultima accortezza.

L'include guard (4)

- Anche in radice.c includiamo il suo corrispondente .h:

main.c

```
#include "radice.h"

int main(void) {
    double d = sqrt(137.0);

    return 0;
}
```

radice.h

```
#if !defined RADICE_H
#define RADICE_H

extern double sqrt(double a);

#endif /* RADICE_H */
```

radice.c

```
#include "radice.h"

double sqrt (double a) {
    double t, x = a;

    if (x<=0.)
        return 0.;

    do {
        t = x;
        x = 0.5*(t + a/t);
    } while (x!=t);

    return x;
}
```

- Quando si scrive una libreria, è previdente includere (solitamente come prima riga) nel file .c il corrispondente file .h.
- In questo modo qualsiasi dichiarazione utile o anche un #define, viene reso disponibile all'esterno, ma anche all'interno, evitando la necessità di scrivere due volte le stesse cose.

L'include guard (5)

- Ricapitolando allora, **tutte le volte che si scrive una libreria**, indipendentemente dalla sua dimensione o dal numero di volte in cui si pensa di riutilizzarla, si seguono queste regole:

1. Nel file .h si inseriscono le seguenti direttive:

```
#if !defined <nome del file tutto maiuscolo con _ al posto dei punti>
#define <nome del file tutto maiuscolo con _ al posto dei punti>

<il vero contenuto del file>

#endif /* <nome del file tutto maiuscolo con _ al posto dei punti> */
```

2. Nel file .c si inserisce come prima linea la seguente:

```
#include "<nome del file .h corrispondente>"
```

- Aperte qualsiasi file .h e .c di una qualsiasi libreria scaricata da Internet e vi accorgete che tutte seguono questa prassi.
- Dovete farlo sempre!**
- Esistono anche alternative, come `#pragma once` o `#ifndef` al posto del `#if !defined`, ma la sostanza non cambia.