

Appunti sull'Assembler

Calcolatori Elettronici

Gabriele Frassi

A.A 2020-2021 - Secondo semestre

Indice degli appunti

1 Unimap	4
2 RL - Indirizzamento degli operandi	6
I Lezioni introduttive e sull'Assembler	8
3 Lunedì 01/03/2021	9
3.1 Scopo del corso	9
3.2 La confusione iniziale	9
3.3 Facciamo un passo indietro: il Manchester Baby	10
3.4 Domanda finale della lezione	14
4 Martedì 02/03/2021	15
4.1 Architettura IBM-compatibile con processore Intel a 64 bit	15
4.1.1 Sviluppi del processore INTEL	16
4.1.2 Confronto col Manchester Baby	16
4.1.2.1 Spazio di memoria	17
4.1.2.2 Introduzione della memoria ROM, utilità della memoria ROM	18
4.1.2.3 Differenza tra hardware e software	18
4.1.3 Riflessioni approfondite sulla CPU	19
4.1.3.1 Spazio di memoria e regole di indirizzamento (CISC)	19
4.1.3.2 Problema: offset dell'indirizzo a 32 bit	19
4.1.3.3 Utilità degli indirizzamenti nelle strutture dati	20
4.1.3.4 Registri	21
5 Giovedì 04/03/2021 e Venerdì 05/03/2021	22
5.1 Unità di misura della memoria	22
5.2 Spazio di memoria non completo e "buco"	23
5.3 Esempio di programma Assembler	23
5.3.1 Debugger	29
5.4 Indirizzi	33
5.4.1 <i>Offset</i> (o scostamento)	33
5.4.2 Intervalli (<i>range</i>)	34
5.4.2.1 Divisione dello spazio di memoria in parti uguali	34
6 Lunedì 08/03/2021	36
6.1 Riprendiamo sullo spazio di memoria	36
6.1.1 <i>endiannes</i>	36
6.1.2 <i>parallelismo</i>	37
6.1.2.1 Immagine dello spazio di memoria	38
6.1.2.2 Organizzazione della RAM	39

6.1.2.3	Osservazioni su lettura disallineata	40
7	Martedì 09/03/2021	42
7.1	Iniziamo ad uccidere il compilatore	42
7.1.1	Esercizio dai lucidi del prof.Frosini	43
8	Giovedì 11/03/2021	47
8.1	Programmi misti C++/Assembler	47
8.1.1	g++ e <i>startfiles</i>	47
8.1.2	g++ e <i>overloading</i>	48
8.1.3	Uso dei registri	51
8.1.4	Rappresentazione dei dati	52
8.1.4.1	Tipi fondamentali	52
8.1.4.2	Tipi derivati	54
8.1.4.3	Primo esempio di struttura	54
8.1.4.4	Secondo esempio di struttura	55
8.1.4.5	Terzo esempio di struttura	55
9	Venerdì 12/03/2021, Lunedì 15/03/2021 e Martedì 16/03/2021	56
9.1	Programmazione mista: funzioni	56
9.1.1	Record di attivazione	57
9.1.1.1	Passaggio di parametri in ingresso	58
9.1.1.2	Parametro in uscita	59
9.1.2	Esempio di struttura passata come parametro in ingresso	59
9.1.3	Esercizio dalle diapositive del prof.Frosini	60
9.1.3.1	Passaggio di parametri per valore (pag.35)	60
9.1.3.2	Passaggio di parametri per riferimento (pag.41)	64
9.1.3.3	Passaggio di parametro array (pag.50)	67
9.1.3.4	Passaggio di parametro struttura (pag.64)	72
10	Venerdì 19/03/2021	76
10.1	Traduzione dei nomi da C++ ad Assembler	76
10.1.1	Regole per le etichette delle funzioni	77
10.1.1.1	Tipi base	77
10.1.1.2	Struttura	78
10.1.1.3	Puntatori	78
10.1.1.4	Riferimenti	79
10.1.1.5	Costanti	79
10.1.1.6	Array	80
10.1.1.7	Esempi di etichette con costanti	81
11	Lunedì 22/03/2021	83
11.1	Corrispondenza C++-Assembler: classi	83
11.1.1	Prima implementazione (da C++ a C): compilatore <i>cfront</i>	83
11.1.2	Implementazione attuale (da C++ ad Assembler direttamente)	85
11.1.2.1	Traduzione dei nomi delle funzioni membro	85
11.2	Esercizio dalle diapositive di Frosini	86
11.2.1	Primo esercizio	86
11.2.2	Secondo esercizio	92

12 Martedì 23/03/2021	94
12.1 Punti su cui stare attenti nella prova pratica	94
12.1.1 Funzioni che restituiscono strutture/classi per valore	94
12.1.2 Esercizio dalle diapositive di Frosini con struttura superiore a 16byte	95
12.1.3 Situazioni su cui riflettere	98
12.1.3.1 Singolo oggetto	98
12.1.3.2 Chiamata di funzione	98
12.1.3.3 Costruttore	99
12.1.3.4 Costruttore di copia	100
12.1.3.5 Elisione dei costruttori di copia (ottimizzazione, RVO ed NRVO)	100
13 Giovedì 28/03/2021 e Venerdì 26/03/2021	103
13.1 Comandi per la lettura di esami passati	103
13.2 Esercizio di traduzione [Scritto del 26/02/2020]	103
13.3 Esercizio di traduzione [Scritto del 19/09/2018]	113
14 Lunedì 29/03/2021 e Martedì 30/03/2021	117
14.1 Concludiamo su compilatore, assemblatore e collegatore	117
14.1.1 Preprocessore	118
14.1.1.1 Analisi di quanto fatto dal preprocessore	118
14.1.1.2 Inclusione di files	119
14.1.1.3 Definizione di macro	120
14.1.1.4 Inclusione condizionale di parti di testo	120
14.1.1.5 Macro predefinite	121
14.1.1.6 Curiosità: eredità dell'indipendenza del preprocessore	121
14.1.2 Assemblatore	122
14.1.3 Formato ELF (<i>Executable and Linking Format</i>)	122
14.1.3.1 Tabella delle sezioni	124
14.1.3.2 Sezione <i>syntab</i> : Tabella dei simboli	124
14.1.3.3 Sezione <i>rela.section_name</i> : tabella di rilocazione	125
14.1.4 Collegatore	126
14.1.4.1 Tabella dei segmenti	127
14.2 Librerie statiche	128
14.2.1 Come si crea una libreria?	128
14.2.2 Comandi utilizzati fino ad ora e librerie	129

Capitolo 1

Unimap

1. **Lun 01/03/2021 10:30-12:30 (2:0 h)** lezione: Introduzione al corso. L'esempio dell'elaboratore SSEM: processore, memoria, linguaggio macchina, caricamento di un programma. (GIUSEPPE LETTIERI)
2. **Mar 02/03/2021 10:30-12:30 (2:0 h)** lezione: Il processore Intel/AMD a 64 bit. Registri, modalità di accesso degli operandi, indirizzamento della memoria. (GIUSEPPE LETTIERI)
3. **Gio 04/03/2021 14:30-16:30 (2:0 h)** lezione: Indirizzi canonici. Esempio di programma assembler: le direttive, le etichette. Utilizzo di assemblatore, collegatore e debugger. (GIUSEPPE LETTIERI)
4. **Ven 05/03/2021 10:30-12:30 (2:0 h)** lezione: Spazio di indirizzamento: confini, offset, intervalli, allineamenti. Scomposizione degli indirizzi. (GIUSEPPE LETTIERI)
5. **Lun 08/03/2021 10:30-12:30 (2:0 h)** lezione: Endianness. Parallelismo negli accessi allo spazio di memoria. Le linee di byte enable. Collegamento al bus di un modulo di RAM. (GIUSEPPE LETTIERI)
6. **Mar 09/03/2021 10:30-12:30 (2:0 h)** esercitazione: Scrittura di programmi assembler su più file. (GIUSEPPE LETTIERI)
7. **Gio 11/03/2021 14:30-16:30 (2:0 h)** lezione: Introduzione alla scrittura di programmi misti C++/Assembler. Funzione main. Registri preservati e scratch. Regole di allineamento dei tipi base e strutturati. (GIUSEPPE LETTIERI)
8. **Ven 12/03/2021 10:30-12:30 (2:0 h)** lezione: Programmazione mista C++/Assembly: record di attivazione, passaggio dei parametri tramite i registri. (GIUSEPPE LETTIERI)
9. **Lun 15/03/2021 10:30-12:30 (2:0 h)** esercitazione: Esercizi di traduzione da C++ ad Assembly, con passaggio di parametri per valore e riferimento. (GIUSEPPE LETTIERI)
10. **Mar 16/03/2021 10:30-12:30 (2:0 h)** esercitazione: Esercizi su corrispondenza tra C++ e Assembly: funzioni con argomenti di tipo array e struttura. (GIUSEPPE LETTIERI)
11. **Gio 18/03/2021 14:30-16:30 (2:0 h)** lezione: Memoria Cache: principi di località, controllore, memoria ad accesso diretto e associativa ad insiemi, politiche di write-through/write-back. (GIUSEPPE LETTIERI)
12. **Ven 19/03/2021 10:30-12:30 (2:0 h)** lezione: Algoritmo pseudo-LRU per cache associative a 4 vie. Traduzione dei nomi C++ in assembler: funzioni globali con parametri di tipi base, definiti dall'utente e composti. (GIUSEPPE LETTIERI)
13. **Lun 22/03/2021 10:30-12:30 (2:0 h)** lezione: Corrispondenza tra C++ e Assembler: classi, oggetti, funzioni membro, costruttori. (GIUSEPPE LETTIERI)

14. **Mar 23/03/2021 10:30-12:30 (2:0 h)** lezione: Funzioni che restituiscono strutture, classi o unioni per valore. Regole per l'elisione dei costruttori di copia. Return Value Optimization e Named Return Value Optimization. (GIUSEPPE LETTIERI)
15. **Gio 25/03/2021 14:30-16:30 (2:0 h)** esercitazione: Svolgimento di testi d'esame su corrispondenza tra C++ e Assembler. (GIUSEPPE LETTIERI)
16. **Ven 26/03/2021 10:30-12:30 (2:0 h)** esercitazione: Svolgimento di testi d'esame su corrispondenza tra C++ e Assembler. (GIUSEPPE LETTIERI)

Capitolo 2

RL - Indirizzamento degli operandi

L'indirizzamento di memoria è complicato, ma fondamentale: il processore, per la maggior parte del tempo, copia pezzi di memoria da una parte a un'altra. L'indirizzamento di memoria è possibile sia con l'operando sorgente che con quello destinatario, ma non è possibile farlo in entrambi in una stessa istruzione (l'assemblatore da errore se ci proviamo).

Caso più generale Il caso più generico di indirizzo è il seguente

$$\text{Indirizzo} = |\text{base} + \text{indice} \times \text{scala} \pm \text{displacement}|_{2^{32}}$$

cioè

```
OPCODEsfx +-disp(base, indice, scala)
```

- la base e l'indice consistono in registri generali a 32 bit. Precedentemente era obbligatorio porre un registro B in base e un registro I in indice: oggi si offre maggiore flessibilità e si possono utilizzare tutti i registri generali.
- scala è una costante che può avere per valore 1 (valore default se non indicato), 2, 4, 8.
- displacement è una costante intera.

Indirizzamento di tipo diretto

Nelle cose viste fino ad ora abbiamo fatto indirizzamenti di memoria diretti, cioè indirizzamenti con solo il displacement.

```
OPCODEW 0x00002001
```

$$\text{Indirizzo} = |0x00002001|_{2^{32}}$$

Indirizzamento di tipo indiretto

Registro base

```
OPCODEL (%EBX)
```

$$\text{Indirizzo} = | \%EBX |_{2^{32}}$$

dove EBX consiste nel registro contenente l'indirizzo. Attenzione: è necessario indicare il suffisso di lunghezza, non abbiamo un indirizzamento di registri.

Registro indice

OPCODEL (,%ESI, 4)

$$\text{Indirizzo} = | \%ESI \times 4 |_{2^{32}}$$

dove ESI consiste nel registro contenente l'indirizzo. Attenzione: è necessario indicare il suffisso di lunghezza, non abbiamo un indirizzamento di registri.

Indirizzamento con displacement e registro di modifica

OPCODEW 0x002A3A2B (%EDI)

$$\text{Indirizzo} = | \%EDI + 0x002A3A2B |_{2^{32}}$$

Indirizzo un operando a 16bit, che si trova nella doppia locazione il cui indirizzo si ottiene sommando (modulo 2^{32}) il displacement e il contenuto di EDI. Questa cosa è molto versatile per i vettori

Indirizzamento bimodificato senza displacement

OPCODEW (%EBX, %EDI)

$$\text{Indirizzo} = | \%EBX + \%EDI \times 1 |_{2^{32}}$$

OPCODEW (%EBX, %EDI, 8)

$$\text{Indirizzo} = | \%EBX + \%EDI \times 8 |_{2^{32}}$$

utilizzo due registri puntatori ponendo, eventualmente, la scala.

Indirizzamento bimodificato con displacement

In questo caso utilizzeremo tutte le armi a nostra disposizione

OPCODEB 0x002F9000 (%EBX, %EDI)

$$\text{Indirizzo} = | \%EBX + \%EDI \times 1 + 0x002F9000 |_{2^{32}}$$

OPCODEB -0x9000 (%EBX, %EDI)

$$\text{Indirizzo} = | \%EBX + \%EDI \times 1 - 0x9000 |_{2^{32}}$$

Parte I

Lezioni introduttive e sull'Assembler

Capitolo 3

Lunedì 01/03/2021

3.1 Scopo del corso

Catena Il corso è parte di una catena che inizia con Reti logiche e finisce con Sistemi operativi, un percorso che va dal calcolatore al sistema operativo.

Argomenti principali Il corso completa la spiegazione della parte hardware di un calcolatore (iniziata con Reti logiche) introducendo i seguenti argomenti:

- interruzioni;
- protezioni;
- memoria virtuale (paginazione).

queste cose ci permetteranno di implementare la cosiddetta **multiprogrammazione**, ossia la capacità di eseguire più di un programma alla volta. Questa cosa, attenzione, non dipende dalla presenza di più di un processore (in questo corso non parleremo di multiprocessore).

Nucleo di un sistema operativo Come da tradizione non ci limiteremo a queste cose chiaccherando: realizzeremo un nucleo di sistema operativo in grado di eseguire più programmi in contemporanea.

3.2 La confusione iniziale

Il fatto che noi studiamo la struttura del calcolatore avendo già avuto a che fare con un sistema operativo è un problema. Le numerose modifiche apportate ai calcolatori per renderli più efficienti hanno portato ad avere uno strato di software molto spesso: ciò nasconde ai nostri occhi ciò che realmente avviene in un calcolatore, quindi ci porta a pensare cavolate.

Domanda da ripetersi ogni volta Chi fa cosa? Dobbiamo essere in grado di dire, in qualunque situazione, chi fa una certa cosa all'interno di un calcolatore. In un contesto ad elevata astrazione è facile dire, *di fronte a una scena del crimine, che il colpevole è il coltello*.

Le tre divinità In questa nuvola di incertezza e ignoranza siamo abituati a sopravvalutare il potere delle varie componenti di un calcolatore. Noi abbiamo tre divinità:

- processore
- sistema operativo (il software)
- compilatore (l'angelo custode dei nostri programmi)

in questo corso lo scopo è **uccidere** queste tre divinità, cioè rendere chiaro cosa effettivamente fanno queste componenti.

3.3 Facciamo un passo indietro: il Manchester Baby

Soluzione Per eliminare la confusione iniziale cosa buona è tornare indietro nel tempo e ripartire dalle cose semplici.

Manchester Baby Il calcolatore SSEM (*Small-Scale Experimental Machine*), detto *Manchester Baby*, è stato realizzato nel 1948 e può essere considerato il "primo computer moderno" (mettere tra tante virgolette). Perché diciamo questo?

- Presenta un concetto importante tipico dei calcolatori moderni: la centralità della memoria (e non del processore).
- La memoria di cui parliamo, implementata con un tubo catodico, è organizzata in celle: ogni cella ha un numero (un valore) ed è identificata da un indirizzo.
- Risulta possibile accedere liberamente a queste celle, in qualunque ordine (accedere significa leggere o scrivere nella cella).

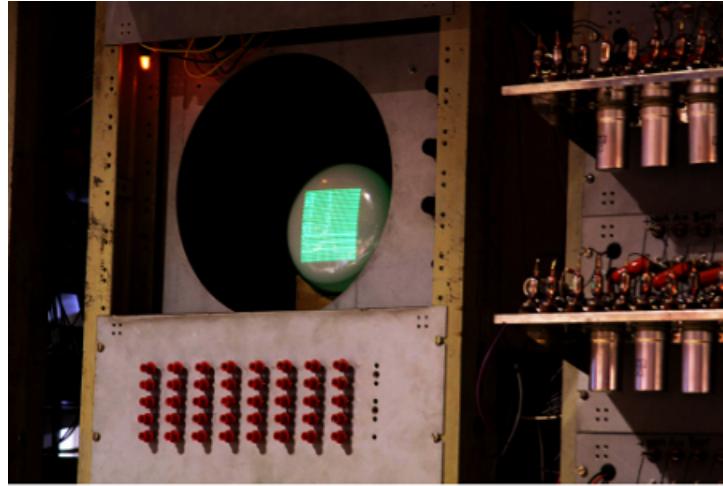
Gli indirizzi sono una delle cose più nascoste dal lato software, ma anche una delle cose più importanti da comprendere per analizzare il comportamento di un calcolatore (l'idea base dell'architettura è quella di una via piena di case, ciascuna con un numero civico, che possiamo visitare per conoscerne i proprietari o per sostituirli).

Andiamo ancora indietro Questo tipo di memoria è stato pensato anche nell'800, ma l'idea era utilizzare la memoria esclusivamente per i dati.

Quindi La novità più rivoluzionaria è l'uso della memoria per ospitare non solo i dati da elaborare, ma ANCHE le istruzioni da eseguire (il programma).

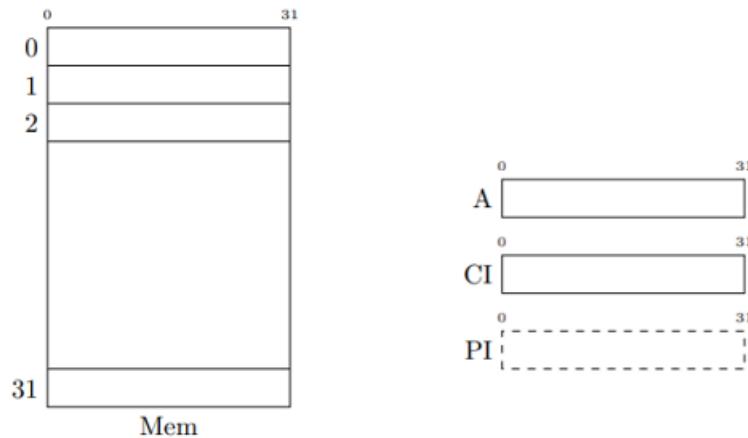
Lo scopo iniziale del calcolatore e le conseguenti evoluzioni I calcolatori sono stati pensati dai pionieri per eseguire operazioni aritmetiche (*calcolatori* nel senso di *calcolare*), dunque gli schemi pensati sono stati calati forzatamente in altre situazioni. Pensiamo alla MOV: si eredita il concetto degli *operandi* (ripensare all'istruzione in Assembler, definiamo i parametri di ingresso col nome di operandi).

Input e Output L'unica forma di ingresso è caratterizzata da un pannello: abbiamo delle file di bottoni che permettono di settare i bit, singolarmente (si hanno almeno 32 bottoni). Prima seleziono (attraverso altri bottoni) la riga, dopo setto i bit che mi interessano.



Il Manchester Baby fu costruito per testare una nuova tecnologia (nuova nel 1948) basata sui tubi catodici: i bit sono memorizzati come punti o linee su uno schermo fluorescente, vengono scritti dirigendo opportunamente un raggio catodico e riletti tramite una griglia metallica che copre lo schermo. Il vantaggio di avere una memoria a tubo catodico è la possibilità di vederne un'immagine rappresentativa con un normale schermo a tubo catodico (come quello dei vecchi televisori). Si osservi una cosa: il software non ha controllo dell'I/O, differenza sostanziale rispetto ad oggi.

Caratteristiche della memoria e registri



- Abbiamo celle di dimensione a 32 bit.
- Contrariamente a quanto già visto la cifra più significativa sta in posizione 0 e non nell'ultima posizione della cella (ne dobbiamo tenere conto).
- Gli indirizzi non sono visibili sullo schermo, ma posti in input con i bottoni del Manchester Baby. Si impostano i bottoni per ottenere una particolare riga e a quel punto la si manipola.

Il processore del Manchester Baby presenta tre registri, tutti a 32 bit:

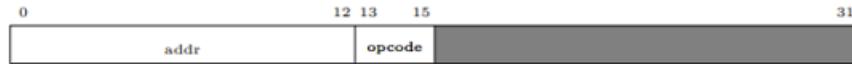
- il registro accumulatore A
- il registro CI (*Current Instruction*)
- il registro PI (*Present Instruction*)

Esecuzione di istruzioni in sequenza Il processore è una macchina che esegue istruzioni elementari contenute in memoria. Riceve dei numeri e li interpreta come istruzioni, precisamente:

1. incrementa CI di 1;
2. legge dalla memoria la locazione di indirizzo CI e la copia in PI;
3. esegue l'istruzione contenuta in PI;
4. se l'istruzione non era di stop, torna al punto 1; altrimenti, accende la lampadina e non fa altro.

Questo ovviamente tenendo conto di eventuali istruzioni di salto (che modifica il contenuto di CI prima che questo venga copiato in PI). Si tenga conto che se l'indirizzo al reset di CI è zero allora la prima istruzione eseguita sarà quella posta all'indirizzo 1. Al di là di queste cose il ragionamento pensato per eseguire istruzioni in sequenza (ed effettuare eventuali salti) è praticamente uguale a quello del processore moderno.

Programmare Programmare significa porre il contenuto iniziale in memoria in modo tale che la macchina possa lavorare da sola. La cosa non è cambiata: anche oggi programmare significa questo! Il numero posto in una cella, abbiamo già detto, viene considerato dal processore come un'istruzione. Vediamo formato e possibili istruzioni



opcode	mnemonico	effetto
000	JMP	$CI \leftarrow Mem[addr]$
100	JRP	$CI \leftarrow CI + Mem[addr]$
010	LDN	$A \leftarrow -Mem[addr]$
110	STO	$Mem[addr] \leftarrow A$
-01	SUB	$A \leftarrow A - Mem[addr]$
011	CMP	<i>se A < 0, CI ← CI + 1</i>
111	STP	<i>halt</i>

Figura 6: Il formato e l'insieme delle istruzioni del SSEM.

- Formato:

- Dalla posizione 0 alla posizione 12 abbiamo l'indirizzo (*addr*), cioè l'operando.

- Dalla posizione 13 alla posizione 15 abbiamo l'OPCODE, cioè l'identificativo dell'istruzione.
- I bit rimanenti non presentano contenuto rilevante.

- Istruzioni:

- JMP: istruzione di salto non condizionato, pongo in CI il contenuto della cella di memoria Mem[addr]
- JRP: jump relativa, aggiorno CI sommando ad esso il contenuto della cella di memoria Mem[addr]
- LDN (*Load Negative*: aggiorno il registro accumulatore ponendo l'opposto del numero memorizzato in Mem[addr] (rappresentazione in C2)
- STO: pongo come contenuto della cella di memoria Mem[addr] il contenuto del registro A.
- SUB: sottrazione, sottraggo al contenuto del registro A il contenuto della cella di memoria Mem[addr]. Il risultato della sottrazione è posto nel registro A. Dovendo ridurre all'osso hanno pensato che la sottrazione sia più utile (e che si possa fare un'addizione utilizzando la LDN). Per fare l'addizione dobbiamo fare il seguente calcolo

$$\text{ADDIZIONE} = -(-X - Y)$$

- CMP: salto condizionato, se il contenuto del registro accumulatore è minore di 0 viene saltata una istruzione (riduzione all'osso del meccanismo visto in Assembler).
- HLT: *halt*, l'esecuzione sequenziale di istruzioni viene fermata e la spia rossa del calcolatore accesa.

Esempio di programma che restituisce l'opposto

istr.	commento
LDN X	carichiamo X , ottenendo $-x$ nell'accumulatore
CMP	se $-x$ è negativo, x era positivo e non dobbiamo fare niente
STO X	altrimenti sostituiamo x con $-x$
STP	fermiamo la macchina

Figura 7: Un esempio di programma. La locazione X deve contenere un valore (x) che alla fine deve essere sostituito dal suo valore assoluto.

Ulteriore esempio Nelle appendici è presente la dispensa di Lettieri sul Manchester Baby. Troverete un ulteriore esempio di programmazione col Manchester Baby.

3.4 Domanda finale della lezione

Chi comanda?

- La memoria è una risposta troppo generica.
- Il processore non può essere perché esegue gli ordini del software e sa solo ciò che è contenuto nei registri. Non conosce il programma passato, ne tantomeno quello futuro. Non conosce gli effetti complessivi di una set di istruzioni, le esegue singolarmente senza analizzarle da un punto di vista globale.

Nel calcolatore chi comanda è il software, è lì che risiede l'intelligenza del programma: noi poniamo codice nella memoria, e ciascuna riga indica un'istruzione da eseguire. Questa cosa è l'essenza del *calcolatore programmabile*: **modificare ciò che il calcolatore fa senza modificare l'hardware**.

Flusso di controllo Il software controlla il processore e gli fornisce l'istruzione successiva (e lo fa in modo esplicito nel caso in cui voglia compiere un'istruzione di salto). Con un processore si ha un unico flusso di controllo.

Cosa succede se il software è composto da più parti? Supponiamo di avere funzioni di libreria: cosa succede? Si dice che il programma cede il controllo del processore alla funzione di libreria (il processore è come una palla che può rimbalzare da una persona a un'altra). Alla fine la funzione di libreria restituirà il controllo al programma.

Capitolo 4

Martedì 02/03/2021

4.1 Architettura IBM-compatibile con processore Intel a 64 bit

Iniziamo a vedere un architettura IBM compatibile con processore Intel a 64 bit. Facciamo alcune premesse.

- Dire che una cosa è uscita in un certo anno non significa dire che quella cosa sia stata inventata in quel momento. Certe volte si sente dire che Steve Jobs o Bill Gates hanno inventato il computer, e ciò è una castroneria.
- A partire dagli anni 70 si è avuto un processo di miniaturizzazione, che ha portato ad avere calcolatori di dimensione minore, accettabili per l'uso domestico.

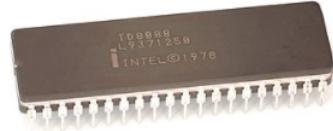
IBM Nel 1981 l'IBM decide di entrare nel mercato dei personal computer. Inizia a commercializzare l'IBM PC 5150 e brevetta il personal computer. Il successo è stratosferico, e la concorrenza sbaragliata.



L'architettura era aperta, quindi fu possibile creare cloni legali detti *IBM compatibili*. Il processore scelto dall'IBM fu l'8088 dell'Intel. Si tenga conto che i personal computer, inizialmente, erano molto meno potenti rispetto ai calcolatori per uso professionale. Nel tempo i PC sono stati dotati di tutte le funzionalità che all'epoca erano presenti solo sui calcolatori per uso professionale (interruzioni e protezioni sono cose che non sono state inventate con i PC).

4.1.1 Sviluppi del processore INTEL

- **8088 16bit**, con le interruzioni
- **80286 16bit**, con interruzioni e protezione
- **80386 32bit**, con interruzioni, protezione e memoria virtuale



Dopo il terzo modello indicato non si hanno più novità rilevanti. Le novità da allora hanno riguardato soprattutto nuovi set di istruzioni e modifiche interne per velocizzare il processore. Verso la fine degli anni 90 la Intel ha introdotto *Itanium* per passare da 32 bit a 64 bit. In contemporanea la AMD ha sviluppato un'estensione dell'architettura Intel esistente, nota come *AMD64*. L'Itanium non ha avuto successo e la Intel ha reso i suoi processori compatibili con l'estensione AMD.

Brutture L'architettura che andremo a vedere è brutta, complicata e irregolare, vittima del suo successo. Il software sviluppato, imponente, è cresciuto negli anni principalmente per motivi di retrocompatibilità. Molte scelte del passato sopravvivono anche oggi in questi processori.

4.1.2 Confronto col Manchester Baby¹

Facciamo un confronto con quanto visto ieri.

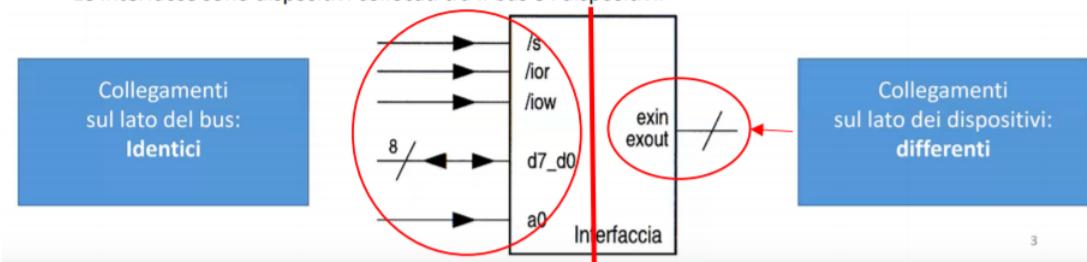
- **CPU:** il meccanismo di esecuzione delle istruzioni è sostanzialmente simile.
 - Prelievo l'istruzione,
 - eseguo l'istruzione, e
 - passo all'istruzione successiva.
- **Memoria:** vediamo le differenze principali.
 - **Differenza di lana caprina:** memoria di maggiore dimensione
 - **Differenze sostanziali:**
 - * Nel Manchester Baby si può accedere soltanto a un'intera riga di 32bit (cioè operazioni di lettura e scrittura). Non si poteva intervenire su sottorighe.
 - * Le architetture erano sostanzialmente divise in due categorie: macchine scientifiche con accesso stile Manchester Baby (celle di dimensione ampia), e macchine ad uso commerciale con memorie organizzate a caratteri (byte che avevano dimensione di 5/6 bit). La macchina moderna ingloba entrambe le categorie:
 - Ogni singolo byte ha un indirizzo (la dimensione del byte è stata standardizzata, 8 bit).
 - La memoria è accessibile non solo a singoli byte, ma anche a multipli del byte (*word*, *doubleword* e *quadrupleword*, rispettivamente 16bit, 32bit e 64bit).

¹Sono consapevole che questa sezione risulterà un po' antipatica. Riguardatevi modo la struttura del calcolatore).

- **I/O:**

- L'I/O non è collegato direttamente alla memoria, ma raggiungibile attraverso il bus.
- **Contrariamente al Manchester Baby è necessario del software:** quanto posto nella periferica rimane al suo interno e deve essere recuperato dal software per essere usato (per esempio quando vogliamo conoscere l'ultimo carattere premuto sulla tastiera). L'interfaccia appare al processore come una memoria, come un insieme di registri che possono essere letti e scritti.

- Le interfacce sono dispositivi collocati tra il bus e i dispositivi.



La differenza importante è che operazioni di lettura e scrittura possono avere effetti collaterali sulle periferiche.

- * Gli indirizzi devono essere pensati come qualcosa che appartiene al bus, separato dalla memoria e dall'I/O. Gli indirizzi, tra memoria e I/O, devono essere UNIVOCI.
- * Un'entità attiva, per dialogare, pone sul bus l'indirizzo della cella con cui vuole parlare. Questo indirizzo è visto da tutti coloro che sono collegati al bus, **non solo da chi reagisce**.
- * Ognuno degli oggetti collegati al bus deve sapere quali sono i propri indirizzi: in base agli indirizzi ricevuti l'oggetto capisce se deve reagire o meno. Considerando l'univocità è chiaro che in una situazione caratterizzata dalla presenza di tante periferiche solo una di queste risponderà.
- **Attenzione.** Storicamente l'architettura prevedeva una distinzione netta tra memoria e I/O (indirizzamenti separati, ripensare alla struttura del calcolatore vista a Reti logiche). Si capiva al volo se la CPU voleva parlare con la memoria o con l'I/O. Nei pc moderni la distinzione è meno netta.

4.1.2.1 Spazio di memoria

Segue che lo spazio di memoria è una cosa diversa dalla memoria.

- **Lo spazio di memoria è l'insieme di tutto ciò che è indirizzabile**, cioè degli indirizzi che il processore genera quando sta eseguendo un'istruzione che ha un operando in memoria, oppure quando sta prelevando un'istruzione.
- Tutto ciò che è presente nello spazio di memoria può essere letto con l'istruzione MOV (ovviamente tenendo conto di meccanismi come la protezione).

Spazio di memoria comprende lo spazio di I/O?

- Nell'architettura che vedremo lo spazio di I/O non è parte dello spazio di memoria: la CPU genera un indirizzo nello spazio di I/O quando esegue le istruzioni IN e OUT.
- Vedremo che in alcuni casi le periferiche (solitamente quelle "più vecchie") hanno i registri nello spazio di I/O, mentre altre (solitamente quelle più recenti) hanno i loro registri nello spazio di memoria (quindi vi si può accedere con una semplice MOV, basta conoscere l'indirizzo).

Dunque

- La CPU genera un accesso sul bus nello spazio di memoria.
- Lo schema prevede che ogni sottoparte dell'oggetto collegato al bus abbia un suo indirizzo diverso: una memoria RAM occupa un intervallo di indirizzi, stessa cosa ogni signora periferica.
- Una volta che l'oggetto ha capito che deve lavorare deve individuare la sottoparte chiamata in causa: prendiamo una periferica, il processore non solo vuole parlare con la periferica, ma vuole anche leggere/scrivere su uno specifico registro.

4.1.2.2 Introduzione della memoria ROM, utilità della memoria ROM

Altra differenza rispetto al Manchester Baby è la presenza di una memoria ROM, oltre che di una RAM. perché ci serve?

- La CPU non fa niente se non c'è del software.
- Dobbiamo eseguire il programma bootstrap, cioè fornire del software base. Se io non eseguo questo programma mi ritrovo con una memoria piena di spazzatura, e la CPU interpreta cose casuali come istruzioni (se possibile).
- Nel Manchester Baby non abbiamo un programma bootstrap, ma non è un problema visto che **possiamo inserire roba in memoria senza dover avere qualcosa già in memoria**. Nell'architettura moderna tutto è gestito dal software, incluse le periferiche di I/O: se non ho software non ho né input né output. Esco da questo ciclo solo introducendo una memoria di sola lettura.

4.1.2.3 Differenza tra hardware e software

Il software consiste in tutto ciò che è contenuto in memoria, l'hardware è tutto il resto.

La confusione è lecita: alcune cose fatte in software potrebbero essere fatte in hardware, e viceversa

4.1.3 Riflessioni approfondite sulla CPU

Abbiamo capito che il suo scopo è eseguire istruzioni in linguaggio macchina. Queste istruzioni possono fare riferimento ad operandi (ricordarsi che l'istruzione base è quella aritmetica, due operandi che producono un risultato da memorizzare da qualche parte).

`OPCODE source, destination`

Dove si trovano gli operandi?

1. In un registro,
2. direttamente nell'istruzione (*operando immediato*, costante), o
3. nello spazio di memoria (ricordiamoci, spazio di memoria adesso è una cosa decisamente più generica).

Per facilitare il parsing l'operando costante inizia col dollaro, mentre il registro con la percentuale. In tutti gli altri casi si ha un operando immediato. Ricordiamoci la regola fondamentale: tutto ciò che ha a che fare con una istruzione deve stare nel processore. Pensiamo al calcolatore a 32 bit visto a Reti logiche: nella fase di fetch tutto ciò che non è presente nel processore viene portato al suo interno.

4.1.3.1 Spazio di memoria e regole di indirizzamento (CISC)

Lo spazio di memoria è la cosa più complessa. L'architettura del processore è detta CISC (*Complex Instruction Set Computer*): il set delle istruzioni è complesso, nel senso che una singola istruzione può fare tante cose². Se dobbiamo generare un indirizzo nello spazio di memoria possiamo chiedere al processore di calcolarlo: le metodiche sono le stesse viste a Reti logiche (si vedano le pagine sulle regole di indirizzamento, riprese dalla dispensa di Reti logiche). Si consideri che Lettieri chiama il *displacement* con un altro nome: *offset*.

4.1.3.2 Problema: offset dell'indirizzo a 32 bit

L'offset ha dimensione di 32bit. Si è deciso di non espanderlo a 64bit per evitare un aumento eccessivo della dimensione delle istruzioni (inutile nella maggior parte dei casi). perché la cosa è fastidiosa? Prendiamo la seguente istruzione

`MOV offset, %RAX`

il fatto che l'offset sia a 32bit ci permette di porre come source solo i primi 4GB di memoria. La AMD ha introdotto due modi per aggirare il problema:

- l'istruzione MOVABS, l'unica istruzione che ha l'offset a 64bit.

```
MOVABS offset, %RAX <---- N.B. UNICO REGISTRO UTILIZZABILE  
MOVABS $costante, %RAX  
MOVABS %RAX, offset
```

²**Definizione della Wikipedia inglese.** A complex instruction set computer is a computer in which single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions

- L'indirizzamento con registro base e offset a 32bit con segno.

`offset(%RIP)`

che sostanzialmente equivale a dire

$$\text{Indirizzo} = |\text{base} \pm \text{offset}|_{\text{modulo}_{2^{64}}}$$

ricordarsi che RIP è un registro a 64bit e che l'offset viene esteso con segno (cioè replicando il bit più significativa per il numero necessario di volte). In un certo senso equivale a stabilire il punto di riferimento: da questo punto posto in RIP possiamo leggere 2GB indietro e 2GB avanti (ricordarsi del segno). Si osservi che questo tipo indirizzamento **già esisteva**, ma è stato reso esplicito per gli operandi. Era già presente nelle istruzioni di salto, introdotte così a Reti logiche:

`JMP %EIP +- displacement`

4.1.3.3 Utilità degli indirizzamenti nelle strutture dati

L'idea centrale è che uno piazza le cose in memoria e accede liberamente, nel modo più semplice possibile. La cosa interessante è che attraverso certi tipi di indirizzamenti posso creare delle strutture dati e calcolare velocemente gli indirizzi delle componenti.

- **Array:** l'array è un insieme di elementi consecutivi, tutti dello stesso tipo. Il dimensionamento di ogni singolo elemento è lo stesso, quindi mi basta indicare il numero di bit di ciascun elemento attraverso la scala e incrementare il registro indice ogni volta.

$$\text{Indirizzo} = |\text{base} \pm \text{indice} \times \text{scala}|_{\text{modulo}_{2^{64}}}$$

- **Struttura:** ho una serie di elementi consecutivi, di dimensione diversa tra loro. Ho indirizzi diversi, ma uno scostamento costante rispetto all'indirizzo base. Questo scostamento è posto nella definizione della struttura.
- **Puntatore:** il puntatore equivale all'indirizzamento di tipo indiretto con registro base. bit di ciascun elemento attraverso la scala e incrementare il registro indice ogni volta.

$$\text{Indirizzo} = |\text{base}|_{\text{modulo}_{2^{64}}}$$

4.1.3.4 Registri

Rispetto a Reti logiche abbiamo più registri, tutti di dimensione maggiore. Sono tutti interni al processore e sono la unità di memorizzazione più veloce che ci sia. Un registro può essere letto e scritto in un ciclo di clock, cosa assolutamente impossibile con una RAM (la lettura/scrittura può richiedere centinaia di clock).

64b	32b	16b	8b	8b
RAX	EAX	AX	AL	AH
RCX	ECX	CX	CL	CH
RDX	EDX	DX	DL	DH
RBX	EBX	BX	BL	BH
RSP	ESP	SP	SPL	
RBP	EBP	BI	BPL	
RSI	ESI	SI	SIL	
RDI	EDI	DI	DIL	
R8	R8D	R8W	R8B	
R9	R9D	R9W	R9B	
R10	R10D	R10W	R10B	
R11	R11D	R11W	R11B	
R12	R12D	R12W	R12B	
R13	R13D	R13W	R13B	
R14	R14D	R14W	R14B	
R15	R15D	R15W	R15B	

Tabella 1: Nomi dei sotto-registri.

Figura 1: I registri del processore.

- I registri già esistenti nell'architettura a 32bit sono stati mantenuti.
- Il registro dei flag è stato esteso a 64bit, ma non contiene nuovi flag. Anche l'instruction pointer è stato esteso a 64 bit.
- I registri rimanenti sono stati estesi e contengono la loro vecchia versione a 32bit: i nomi per richiamarle sono esattamente gli stessi!
- La AMD ha uniformato gli accessi, nel senso che nei rimanenti registri a 64bit troviamo SEMPRE la versione a 16bit e 8bit (cosa che non era valida per alcuni registri, ripensare a Reti logiche).
- Resta una irregolarità: solo i registri A, B, C, D sono accessibili anche nella parte più significativa dei primi 16bit (AH, BH, CH, DH), gli altri sono accessibili solo nella parte meno significativa.

A differenza del manchester baby abbiamo (cose già viste a Reti):

- istruzioni che alterano in modo implicito il registro stackpointer (per esempio la chiamata di funzione, tutto ciò che è legato alla pila);
- istruzioni che alterano il registro dei flag settandoli (per esempio STD e CLD);
- istruzioni che verificano se sono presenti specifiche configurazioni di flag e decidono, conseguentemente, se fare salto o meno (JMP condizionate, il Manchester Baby prevedeva un JMP condizionata molto primitiva³).

³Verificare se il numero posto nel registro accumulatore è positivo o negativo, dunque fare un eventuale salto DI UNA SOLA ISTRUZIONE.

Capitolo 5

Giovedì 04/03/2021 e Venerdì 05/03/2021

5.1 Unità di misura della memoria

Dato un numero di bit, vogliamo indicare la dimensione dell'area di memoria con le apposite unità di misura. Osserviamo le potenze

$$2^2 = 4 \quad 2^3 = 8 \quad 2^4 = 16 \quad 2^5 = 32 \quad 2^6 = 64 \quad 2^7 = 128 \quad 2^8 = 256 \quad 2^9 = 512 \quad 2^{10} = 1024$$

A questo punto i dubbi ci vengono. Noi informatici abbiamo sempre abusato di queste lettere....

$$\begin{array}{ll} 2^{10} \simeq 1000 \longrightarrow K & 2^{20} \simeq 1000000 \longrightarrow M \\ 2^{30} \simeq \dots \longrightarrow G & 2^{40} \simeq \dots \longrightarrow T \end{array}$$

Cerchiamo di risolvere la confusione con la seguente tabella

Misure binarie			Misure decimali		
Nome	Fattore	Valore in byte	Nome	Fattore	Valore in byte
kibibyte (KiB)	2^{10}	1,024	kilobyte (KB)	10^3	1,000
mebibyte (MiB)	2^{20}	1,048,576	megabyte (MB)	10^6	1,000,000
gibibyte (GiB)	2^{30}	1,073,741,824	gigabyte (GB)	10^9	1,000,000,000
tebibyte (TiB)	2^{40}	1,099,511,627,776	terabyte (TB)	10^{12}	1,000,000,000,000

Abbiamo introdotto delle nuove unità di misura. Quelle "tradizionali" a cui siamo abituati sono *misure decimali*, mentre quelle nuove sono *misure binarie*.

$$\begin{array}{lll} 1 \text{ KB} = 1000 \text{ byte} & 1 \text{ KiB} = 1024 \text{ byte} & \Delta = 24 \text{ byte} \\ 2 \text{ KB} = 2 * 1000 = 2000 \text{ byte} & 2 \text{ KiB} = 2 * 1024 = 2048 \text{ byte} & \Delta = 48 \text{ byte} \\ 3 \text{ KB} = 3 * 1000 = 3000 \text{ byte} & 3 \text{ KiB} = 3 * 1024 = 3072 \text{ byte} & \Delta = 72 \text{ byte} \\ = \dots & = \dots & = \dots \\ N \text{ KB} = N * 1000 = N * 10^3 & N \text{ KiB} = N * 1024 = N * 10^3 + N * 24 & \Delta = N * 24 \text{ byte} \end{array}$$

Risulta facile capire che più le dimensioni aumentano più le misure binarie si discostano da quelle decimali.

Quanto spazio ho con 48 bit? $2^{48} = 2^{40} \cdot 2^8 = 256 \text{ TiB}$

Quanto spazio ho con 57 bit? $2^{57} = 2^{50} \cdot 2^7 = 128 \text{ PiB}$

5.2 Spazio di memoria non completo e "buco"

Lo spazio di memoria non è completo: non tutti i 64bit che compongono un indirizzo sono implementati. La AMD si è riservata di poterne aggiungere in futuro. Lo ha già fatto: i primi processori avevano solo 48 bit significativi, adesso i bit significativi sono 57. Bit in più comportano complicazioni in più, soprattutto costo maggiore nell'utilizzo.

Formato degli indirizzi

I bit non utilizzati devono essere uguali al bit più significativo tra quelli utilizzati.

Gli indirizzi che rispettano il formato sono detti in **forma canonica**. All'interno del processore esiste un meccanismo di protezione che impedisce al processore di andare avanti se l'indirizzo posto non rispetta il formato. La AMD ha adottato questo meccanismo remore dell'esperienza della IBM. Questa, quando ha cercato di implementare indirizzi più grandi, ha avuto problemi: molti programmi hanno smesso di funzionare perché utilizzavano i fili di indirizzo non implementati per altri scopi.

Conseguenza All'interno dello spazio di memoria abbiamo una sorta di buco, che va dall'ultimo indirizzo avente tutti e 7 i bit più significativi uguali a 0 al primo indirizzo avente tutti e 7 i bit più significativi uguali ad 1.

5.3 Esempio di programma Assembler

Esami su Linux Gli esami saranno svolti su macchina virtuale con Debian. Non è necessario avere grande dimestichezza verso questo sistema operativo.

Funzionamento

- Produciamo un file editor.s
- L'assemblatore produce, a partire dal file precedente, un file oggetto .o. I file oggetto sono semplici dati, nulla di eseguibile.
- Il collegatore unisce questo file oggetto con altri files oggetto (library) e crea un eseguibile.

Contrariamente al Manchester Baby **stiamo creando programmi utilizzando altri programmi**. Non è necessario che i files oggetto siano presenti sul dispositivo dove avvieremo l'eseguibile: quando acquistiamo un programma abbiamo solo ciò che serve per eseguire il programma, non gli elementi utilizzati per creare il programma.

Come creiamo programmi in una macchina nuova che non ha assemblatore? Creiamo l'assemblatore e il collegatore su una macchina già esistente: il risultato gira sulla macchina vecchia, ma funziona anche sulla macchina nuova. Otteniamo l'eseguibile sulla macchina vecchia e lo poniamo nella macchina nuova.

In che linguaggio è scritto il compilatore per il C? In C. Come è possibile?

- I compilatori moderni sono compilati da versioni precedenti.
- La prima versione del compilatore in C è scritta in un altro linguaggio.
- I primi compilatori, successivamente, sono stati scritti in linguaggio assembler.
- I primi assemblatori sono stati scritti in linguaggio macchina.

Assemblatore L'assemblatore traduce dal linguaggio assembler al linguaggio macchina. Precisamente, lo vediamo dal nome, l'assemblatore assembra il contenuto della memoria (programmare significa indicare il contenuto iniziale della memoria). Fa ciò che noi abbiamo fatto a mano per scrivere un programma nel Manchester Baby.

perché separare assemblatore e collegatore? Conviene scrivere questi pezzi di memoria senza decidere dall'inizio dove dovranno essere caricati. Scrive le cose, ma rimanda la decisione: questo perché vuole scrivere tanti pezzi (quando scrive un pezzo non sa riguardo gli altri).

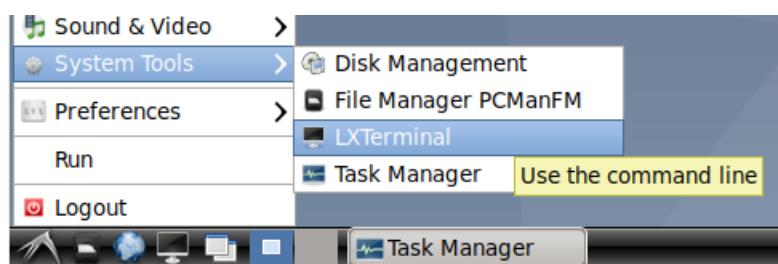
Codice

```
.data
num1:
    .quad 0x1122334455667788
num2:
    .quad 0x9900aabbcdddeeff
risu:
    .quad -1

.text
.global _start
_start:
    movabs num1, %rax
    mov %rax, %rbx
    movabs num2, %rax
    add %rbx, %rax
    movabs %rax, risu

# uscita
    mov $0, %rbx
    mov $1, %rax
    int $0x80
```

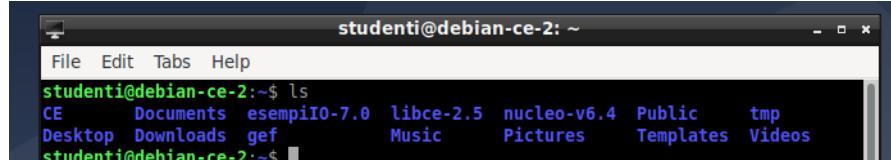
Raggiungiamo il terminale



Comandi eseguiti In una riga pongo prima il comando, e poi gli argomenti

- **ls**

carico la lista dei files/presenti nella directory

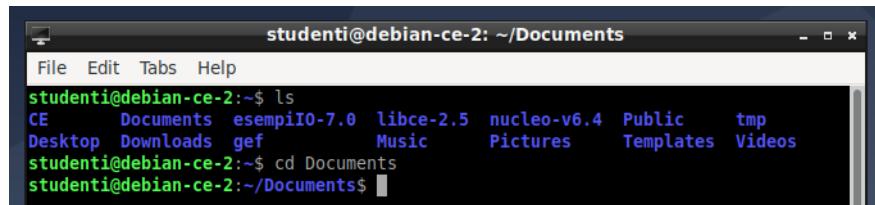


```
studenti@debian-ce-2:~$ ls
CE      Documents  esempiIO-7.0  libce-2.5  nucleo-v6.4  Public    tmp
Desktop  Downloads  gef          Music     Pictures    Templates  Videos
studenti@debian-ce-2:~$
```

se non sono presenti files/cartelle non restituisce nulla.

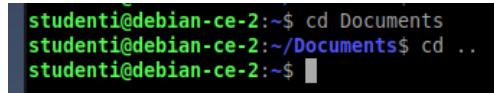
- **cd nome_cartella**

accedo a una directory



```
studenti@debian-ce-2:~/Documents$ ls
CE      Documents  esempiIO-7.0  libce-2.5  nucleo-v6.4  Public    tmp
Desktop  Downloads  gef          Music     Pictures    Templates  Videos
studenti@debian-ce-2:~/Documents$ cd Documents
studenti@debian-ce-2:~/Documents$
```

Nel caso in cui la directory non esista viene restituito l'errore *No such file or directory*.
Per tornare indietro di una directory basta porre due punti come nome della cartella



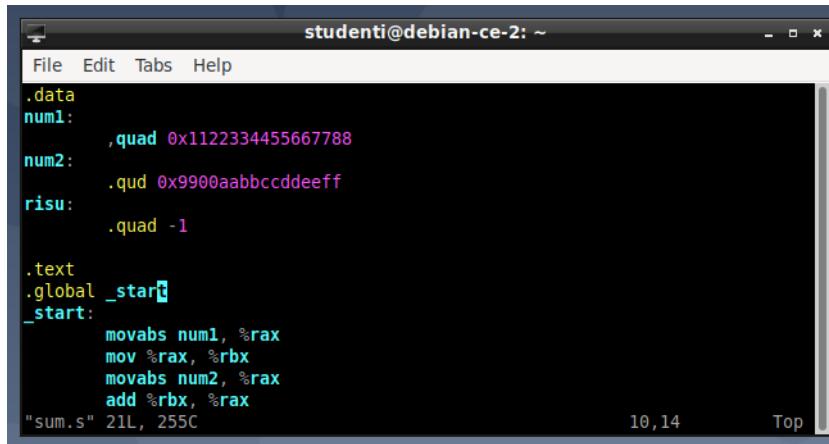
```
studenti@debian-ce-2:~$ cd Documents
studenti@debian-ce-2:~/Documents$ cd ..
studenti@debian-ce-2:~$
```

- **mkdir nome_cartella**

creo una cartella col nome indicato.

- **vi sum.s**

carico l'editor di modifica per il file *sum.s*.



```
studenti@debian-ce-2:~$ vi sum.s
File Edit Tabs Help
.data
num1:
    .quad 0x1122334455667788
num2:
    .quad 0x9900aabbcdddeeff
risu:
    .quad -1

.text
.global _start
_start:
    movabs num1, %rax
    mov %rax, %rbx
    movabs num2, %rax
    add %rbx, %rax
"sum.s" 21L, 255C
10,14           Top
```

Se il file esiste viene caricato l'editor di modifica per quel file, se non esiste viene caricato l'editor lo stesso e in caso di salvataggio viene creato. Si tenga conto che al caricamento dell'editor entriamo in modalità comando.

- Per entrare in modalità modifica digitare la seguente lettera

i

che permette di iniziare la modifica a partire dal carattere dove si trova il cursore del terminale (altre lettere permettono di entrare in modalità modifica, la differenza sta nel punto da dove inizieremo a modificare).

- Per ritornare alla modalità comando basta utilizzare il tasto *ESC*.

- Per annullare modifiche precedenti digitare la seguente lettera

u

- Per chiudere l'editor e salvare le modifiche fatte utilizzare il seguente comando

:wq

se invece non vogliamo salvare le modifiche

:q

- **as -o sum.o sum.s**

indico che voglio assemblare il programma. Tradizione UNIX è l'assenza di messaggi se tutto va bene. Indico col secondo parametro il nome da dare all'output (*sum.o*), mentre col terzo indico il file Assembler che voglio assemblare (*sum.s*).

- **objdump -d sum.o**

esamino il contenuto del file oggetto indicato. Con l'opzione d chiedo il *disassembly*, cioè chiedo di rigenerare il file assembler originario partendo dal binario.

```
studenti@debian-ce-2:~$ objdump -d sum.o

sum.o:      file format elf64-x86-64

Disassembly of section .text:
0000000000000000 <_start>:
 0: 48 a1 00 00 00 00 00 00    movabs 0x0,%rax
 7: 00 00 00
 a: 48 89 c3                  mov    %rax,%rbx
 d: 48 a1 00 00 00 00 00 00    movabs 0x0,%rax
14: 00 00 00
17: 48 01 d8                  add    %rbx,%rax
1a: 48 a3 00 00 00 00 00 00    movabs %rax,0x0
21: 00 00 00
24: 48 c7 c3 00 00 00 00 00    mov    $0x0,%rbx
2b: 48 c7 c0 01 00 00 00 00    mov    $0x1,%rax
32: cd 80                      int   $0x80

studenti@debian-ce-2:~$
```

- Il formato del file disassemblato è: *file format elf64-x86-6*
- Si osservi che l’assemblatore non sa dove stanno *_start*, *num1*, *num2* e *risu*.
 - * Nella prima *movabs* non sappiamo dove finirà *num1* (e non può fare ipotesi sull’offset, visto che le parti data e le parti text saranno unificate in presenza di più files). L’istruzione è identificata dal byte *a1*, che rappresenta l’istruzione e il passaggio da un indirizzo al registro RAX. Sono riservati 8 byte per l’indirizzo.
 - * La prima *mov* ha solo operandi registro, è molto più compatta. La codifica identifica la *mov* e il passaggio da registro RAX a registro RBX (*89 c3*).
 - * Il numero 48, posto all’inizio di alcune istruzioni, è il prefisso introdotto dalla AMD per le istruzioni che devono usare operandi a 64 bit (codice riciclato, nell’architettura a 32 bit apparteneva alle istruzioni INC).

- **nm -n sum.o**

lista degli offset assegnati ai vari simboli.

```
studenti@debian-ce-2:~$ nm -n sum.o
0000000000401000 d num1
0000000000401000 T _start
0000000000401008 d num2
0000000000401010 d risu
studenti@debian-ce-2:~$
```

abbiamo le etichette utilizzate in *.data* e in *.text*.

- **ld -o sum sum.o**

collegatore, indico come argomenti i files oggetto da collegare. Viene generato il file eseguibile.

- **./sum**

indico il nome dell’eseguibile per eseguire il programma. In questo caso non viene mostrato nulla, visto che il programma effettua solo spostamenti di memoria.

- **objdump -d sum**

richiedo nuovamente il *disassembly*.

```
Disassembly of section .text:
0000000000401000 <_start>:
401000: 48 a1 00 20 40 00 00    movabs 0x402000,%rax
401007: 00 00 00
40100a: 48 89 c3                mov    %rax,%rbx
40100d: 48 a1 08 20 40 00 00    movabs 0x402008,%rax
401014: 00 00 00
401017: 48 01 d8                add    %rbx,%rax
40101a: 48 a3 10 20 40 00 00    movabs %rax,0x402010
401021: 00 00 00
401024: 48 c7 c3 00 00 00 00    mov    $0x0,%rbx
40102b: 48 c7 c0 01 00 00 00    mov    $0x1,%rax
401032: cd 80                  int   $0x80
studenti@debian-ce-2:~$
```

Otteniamo la stessa cosa di prima, ma vediamo che gli indirizzi dei dati sono definiti. Domanda che sorge spontanea: l'eseguibile è un file oggetto? Il file oggetto visto prima e l'eseguibile hanno lo stesso formato (*file format elf64-x86-6*).

- L'OPCODE occupa al più due byte.
- In presenza di indirizzi e costanti aumenta la dimensione dell'istruzione. Si ha minore occupazione di memoria coi registri.
- Attenzione al *little-endian*.

Ragioniamo sul programma

- Il programma ha lo scopo di sommare due numeri *quad*.
- Si osservino i passaggi che facciamo con l'istruzione *movabs*, necessari per evitare probabili troncamenti dell'indirizzo a causa dell'*offset* a 32 bit (spazio insufficiente per contenere l'intero indirizzo, se succede il programma inchioda e segnala errore). La cosa dipende da dove sarà collocato in memoria il nostro programma: non lo possiamo sapere a priori, dunque conviene prevenire.
- La *movabs* viene eseguita più volte poichè l'unico registro destinatario possibile è *RAX*.
- Il collegatore vede solo le etichette *.global*

```
.global _start
```

In questo caso la utilizzo per dichiarare l'*entry point* del mio programma.

- In *.data* inizializzo i byte che conterranno i miei dati.

```
.data
num1:
    .quad 0x1122334455667788
num2:
    .quad 0x9900aabbccddeeff
risu:
    .quad -1
```

Possiamo inizializzare *byte* (8 bit, 1 byte), *word* (16 bit), *long* (32 bit) e *quad* (64 bit). Si consideri che le etichette non occupano spazio e ci permettono di dare nomi ad aree che vedranno associato un indirizzo solo dopo.

- **Cosa succede se non definisco il valore iniziale dell'insieme di byte introdotto con la direttiva?** Non viene allocato lo spazio!
- **Non è importante ottimizzare il programma:** non vogliamo diventare programmatori di Assembler. Il compilatore è molto più bravo di noi in questo.
- **Sintassi nuova a 64bit.** Riprendiamo la seguente cosa

```
num1(%rip)
```

L’assemblatore prende questa cosa come l’ordine di calcolare l’offset tra num1 e rip, cioè la differenza. Questa ”scappatoia” permette di risolvere il problema del troncamento dell’indirizzo per spazio insufficiente. Il disassemblatore, se abbiamo utilizzato il collegatore, ci restituisce l’indirizzo. Si osservi che in istruzioni diverse l’offset è diverso. Il fatto che non siamo noi a calcolare l’offset è sicuramente una semplificazione. Se riprendiamo gli esempi di indirizzamento di Reti logiche ci ricordiamo che in quei casi dobbiamo indicare noi lo scostamento. Nel caso spiegato ora no.

- **Accesso ad aree di memoria.** L’idea base è che io possa accedere esclusivamente alle cose che ho dichiarato. Questa cosa è assolutamente falsa: io posso scrivere dove mi pare, tenendo conto di alcuni limiti. Il kernel, se accediamo ad aree di memoria dove non abbiamo permessi, ci segnala errore di *Segmentation Fault*. Osserviamo lo spazio allocato per il nostro programma: se io accedo a 402000 (dove c’è num1) non avrò problemi. Posso accedere senza problemi fino all’indirizzo 402ff0 (se eseguo un’operazione di MOV non ho problemi). Non appena mi sposto di un solo byte mi viene nuovamente segnalato segmentation fault.

- **Cosa succede se scrivo..?**

```
mov %rax, num1+16
```

Scrivo su risu. Chi calcola questa espressione? L’assemblatore o il collegatore (in questo caso il collegatore, num1 non è conosciuto dall’assemblatore)! Abbiamo un’espressione costante. Si osservi che l’assemblatore sa fare solo espressioni semplici.

5.3.1 Debugger

Utilizzeremo lo GNU debugger, che possiamo chiamare col seguente comando

```
gdb sum
```

dove *sum* è il nome dell’eseguibile.

```
studenti@debian-ce-2:~$ gdb sum
GNU gdb (Debian 8.2.1-2+b3) 8.2.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from sum... (no debugging symbols found)...done.
(gdb)
```

Il debugger è lo stesso visto a Reti logiche! Purtroppo è abbastanza spartano... Possiamo renderlo un po' più piacevole introducendo l'estensione *gef* col seguente comando

```
source ~/gef/gef.py
```

Il programma non parte finchè non eseguiamo il comando *start*. Il programma parte e subito si ferma, restituendo il controllo al debugger.

```
(gdb) source ~/gef/gef.py
GEF for linux ready, type `gef` to start, `gef config` to configure
92 commands loaded for GDB 8.2.1 using Python engine 3.7
gef> start

[+] Breaking at '{<text variable, no debug info>} 0x401000 <_start>'
[ Legend: Modified register | Code | Heap | Stack | String ] ----- registers -----
$rax : 0x0
$rbx : 0x0
$rcx : 0x0
$rdx : 0x0
$rsp : 0x00007fffffff210 → 0x0000000000000001
$rbp : 0x0
$rsi : 0x0
$rdi : 0x0
$rip : 0x0000000000401000 → <_start+0> movabs rax, ds:0x402000
$r8 : 0x0
$r9 : 0x0
$r10 : 0x0
$r11 : 0x0
$r12 : 0x0
$r13 : 0x0
$r14 : 0x0
$r15 : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000 ----- stack -----
0x00007fffffff210 +0x0000: 0x0000000000000001 ← $rsp
0x00007fffffff218 +0x0008: 0x00007fffffff4f7 → "/home/studenti/sum"
0x00007fffffff220 +0x0010: 0x0000000000000000
0x00007fffffff228 +0x0018: 0x00007fffffff50a → "SHELL=/bin/bash"
0x00007fffffff230 +0x0020: 0x00007fffffff51a → "QT_ACCESSIBILITY=1"
0x00007fffffff238 +0x0028: 0x00007fffffff52d → "COLORTERM=truecolor"
0x00007fffffff240 +0x0030: 0x00007fffffff541 → "XDG_CONFIG_DIRS=/etc/xdg"
0x00007fffffff248 +0x0038: 0x00007fffffff55a → "XDG_SESSION_PATH=/org/freedesktop/DisplayManager/S[...]" ----- code:x86:64 -----
    0x400ffa          add    BYTE PTR [rax], al
    0x400ffc          add    BYTE PTR [rax], al
    0x400fe           add    BYTE PTR [rax], al
→  0x401000 <_start+0>   movabs rax, ds:0x402000
    0x40100a <_start+10>  mov    rbx, rax
    0x40100d <_start+13>  movabs rax, ds:0x402008
    0x401017 <_start+23>  add    rax, rbx
    0x40101a <_start+26>  movabs ds:0x402010, rax
    0x401024 <_start+36>  mov    rbx, 0x0 ----- threads -----
[#0] Id 1, Name: "sum", stopped 0x401000 in _start (), reason: BREAKPOINT ----- trace -----
[#0] 0x401000 → _start()
gef> █
```

L'estensione *gef* ci mostra, ogni volta che si ferma il debugger:

- la lista dei registri con relativo contenuto (vengono evidenziati i registri che sono cambiati di valore rispetto alla "fermata" precedente);

- tra i registri quello dei flag (nomi dei flag, evidenziati se il valore è 1);
- la pila;
- il disassemblato.

Vediamo alcune azioni che potrebbero tornarci comode:

- **Disassemblato in formato Intel:** il disassemblato di default (quello in foto) è in sintassi Intel, possiamo impostare la sintassi a cui siamo abituati con i seguenti due comandi

```
set disassembly-flavor att <---- imposto la sintassi a cui siamo abituati
context <---- ricarico il contenuto
```

```

0x400ffa      add    BYTE PTR [rax], al
0x400ffc      add    BYTE PTR [rax], al
0x400ffe      add    BYTE PTR [rax], al
~ 0x401000 <_start+0> movabs rax, ds:0x402000
0x40100a <_start+10> mov    rbx, rax
0x40100d <_start+13> movabs rax, ds:0x402008
0x401017 <_start+23> add    rax, rbx
0x40101a <_start+26> movabs ds:0x402010, rax
0x401024 <_start+36> mov    rbx, 0x0

```



```

0x400ffa      add    %al, (%rax)
0x400ffc      add    %al, (%rax)
0x400ffe      add    %al, (%rax)
0x401000 <_start+0> movabs 0x402000, %rax
0x40100a <_start+10> mov    %rax, %rbx
0x40100d <_start+13> movabs 0x402008, %rax
0x401017 <_start+23> add    %rbx, %rax
0x40101a <_start+26> movabs %rax, 0x402010
0x401024 <_start+36> mov    $0x0, %rbx

```

- **Configurazione dell'estensione gef:** il seguente comando

```
gef config
```

mi permette di caricare la lista delle impostazioni dell'estensione gef.

```

context.clear_screen (bool) = True
context.enable (bool) = True
context.grow_stack_down (bool) = False
context.ignore_registers (str) = ""
context.layout (str) = "legend regs stack code args source memory threads trace extra"
context.libc_args (bool) = False
context.libc_args_path (str) = ""
context.nb_lines_backtrace (int) = 10
context.nb_lines_code (int) = 6
context.nb_lines_code_prev (int) = 3

```

- **Modifica di impostazioni dell'estensione gef:** modifichiamo un'impostazione, precisamente la context.layout che indica cosa deve apparirci ogni volta che eseguiamo la context

```
gef config context.layout "regs code memory"
context
```

in questo modo abbiamo rimosso la pila (visibile per default, per il momento non ci serve). Mostriamo solo registri, codice e memoria.

The screenshot shows the gef debugger interface. At the top, it says "registers". Below that is a table of registers:

\$rax	:	0x0
\$rbx	:	0x0
\$rcx	:	0x0
\$rdx	:	0x0
\$rsp	:	0x00007fffffe210 → 0x0000000000000001
\$rbp	:	0x0
\$rsi	:	0x0
\$rdi	:	0x0
\$rip	:	0x0000000000401000 → <_start+0> movabs 0x402000, %rax
\$r8	:	0x0
\$r9	:	0x0
\$r10	:	0x0
\$r11	:	0x0
\$r12	:	0x0
\$r13	:	0x0
\$r14	:	0x0
\$r15	:	0x0

\$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]

scs: 0x0033 **sss:** 0x002b **sds:** 0x0000 **ses:** 0x0000 **sfs:** 0x0000 **sgs:** 0x0000

Below the registers is a table of assembly code:

0x400ffa	add	%al, (%rax)
0x400ffc	add	%al, (%rax)
0x400ffe	add	%al, (%rax)
→ 0x401000 <_start+0>	movabs	0x402000, %rax
0x40100a <_start+10>	mov	%rax, %rbx
0x40100d <_start+13>	movabs	0x402008, %rax
0x401017 <_start+23>	add	%rbx, %rax
0x40101a <_start+26>	movabs	%rax, 0x402010
0x401024 <_start+36>	mov	\$0x0, %rbx

At the bottom left is the prompt "gef>" followed by a small terminal window icon.

- **Contenuto della memoria a partire da un certo indirizzo nella schermata:** col seguente comando chiediamo di mostrare, a partire dall'indirizzo num1, tre qword

```
memory watch &num1 3 qword
context
```

A memory dump table showing three qwords starting at address 0x401024. The table has columns for address, offset, and value.

0x401024 <_start+36>	0x0000000000402000	0x0000000000402008	0x0000000000402010	0x0000000000402010
	+0x0000 <num1+0000> 0x1122334455667788	+0x0008 <num2+0000> 0x9900aabbcdddeeff	+0x0010 <risu+0000> 0xffffffffffffffff	

At the bottom left is the prompt "gef>" followed by a small terminal window icon.

- **Esecuzione di una singola istruzione:** col seguente comando il debugger cede il controllo al programma per eseguire una singola istruzione.

```
si
```

- **Uscita dal debugger.** Ricordiamo il comando per uscire dal debugger

```
quit
```

5.4 Indirizzi

Abbiamo già detto che il processore lavora con indirizzi, e che ogni cosa deve essere identificata mediante un indirizzo.

- **Prima cosa necessaria è avere dimestichezza con i numeri esadecimali:** sono comodi, una cifra esadecimale mi rappresenta quattro cifre binarie.
- Gli indirizzi *vanno pensati come circolari*: ripensare all'operatore modulo di Reti logiche.

$$|11111111 + 00000001|_{2^8} = 00000000$$

- Per rappresentare un indirizzo in C utilizziamo l'***unsigned long***. Attenzione alle direttive:
 - in caso di overflow di un *unsigned long* ripartiamo da zero (cosa pensata proprio per gli indirizzi);
 - in caso di overflow di un *long* sostanzialmente *fa quello che vuole*, si ottiene un *undefined value*.
- Il compilatore del C può assumere, per motivi di efficienza, che la condizione non accada mai. Prendiamo il seguente esempio

```
long x;  
if(x+1 < x)  
...
```

il compilatore ignora completamente la condizione, visto che non sarà mai vera. **Non avverrà la stessa cosa in presenza di un *unsigned long*:** in quel caso la condizione ha senso e permette di verificare se c'è stato overflow.

5.4.1 *Offset* (o scostamento)

- L'*offset* consiste nella distanza tra due indirizzi *x* ed *y*.
- Lo scostamento, precisamente, rappresenta il numero di byte che dobbiamo saltare per passare dall'indirizzo *x* all'indirizzo *y*.
- La cosa vale con $x < y$, ma anche con $x > y$ (cioè l'offset può essere anche negativo). Vale anche quando abbiamo l'overflow, e quindi il modulo riparte da zero (circolarità dell'operatore modulo).
- **Attenzione** al dimensionamento dell'area dove poniamo l'offset. Se è inferiore alla dimensione degli indirizzi potrebbero emergere problemi.

5.4.2 Intervalli (*range*)

Un intervallo è una sequenza di indirizzi.

Convenzione Per convenzione gli intervalli presentano la seguente forma

$$[x, y) = \{z | x \leq z < y\} \quad \text{con } x \geq y$$

Questo ci permette di combinare facilmente due intervalli consecutivi. Per semplicità evitiamo di considerare intervalli che attraversano l'ultimo indirizzo rappresentabile e ripartono da zero (che avrebbero $y < x$). L'insieme $[x, y)$ è vuoto se $y \leq x$.

Grandezza dell'intervallo La grandezza dell'intervallo è esattamente $y - x$.

Base dell'intervallo x è detta *base dell'intervallo*. Il suo indirizzo è l'indirizzo dell'intervallo.

5.4.2.1 Divisione dello spazio di memoria in parti uguali

Molto spesso conviene immaginare lo spazio di memoria diviso in parti uguali, ciascuna di dimensione di una potenza di due. I punti in cui si passa da un intervallo a un altro sono detti *confini*. L'area compresa tra due confini non ha un nome in letteratura, noi la chiameremo *regione naturale*. Un indirizzo che si trova al confine si riconosce in maniera immediata:

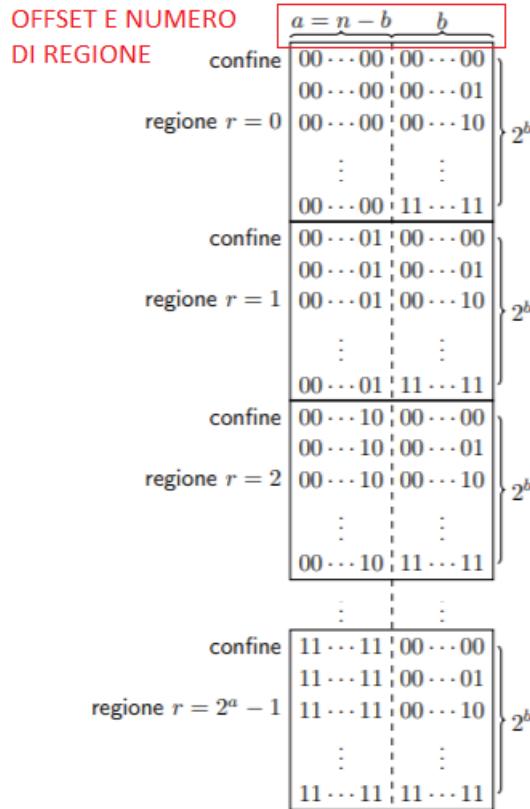


Figura 1: Scomposizione degli indirizzi in regioni naturali di 2^b .

Osserviamo la struttura dell'indirizzo di base e la struttura dell'ultimo indirizzo di un intervallo.
Dati n bit di un indirizzo avrò:

- i b bit meno significativi come **offset** (precisamente l'offset rispetto all'indirizzo di base del relativo intervallo), e
- gli $n - b$ bit più significativi come **numero di regione**.

Come pongo questi numeri in due variabili C++?

```
unsigned long x;  
  
unsigned long nr = x >> b;  
unsigned long off = x & ((1 << b) - 1);
```

- **Numero di regione**: traslo a destra b volte. Non posso usare una maschera perché otterrei il numero di regione naturale moltiplicato per 2^b .
- **offset**: utilizzo l'operatore AND e una maschera che mi lascia solo i b bit meno significativi. L'operazione mi permette di ottenere $2^b - 1$.

Intervallo che inizia in una regione e finisce in un'altra Come trovo l'ultima regione toccata dall'intervallo $[x, y)$?

- Per prima cosa devo controllare che l'intervallo non sia vuoto, per esempio $[x, x)$,
- se non lo è mi basta prendere il numero di regione di y .

perché ci servono queste spiegazioni? Parleremo di memoria RAM e periferiche che occupano porzioni di indirizzi, appunto *intervalli*. Lavoreremo (con una singola istruzione) su oggetti di un byte, due, quattro, otto byte.

Oggetto allineato in memoria Un oggetto si dice *allineato* a qualcosa se il suo indirizzo è multiplo di una qualche potenza di due, cioè se l'indirizzo si trova a confine di una qualche regione naturale. Vediamo delle espressioni frequenti:

- Oggetto allineato a 2^b
- Oggetto allineato a Y (cioè un oggetto tale che $\dim Y = 2^b$)
- Oggetto allineato naturalmente (cioè l'oggetto è allineato alla sua dimensione, $\dim 2^b$).

Capitolo 6

Lunedì 08/03/2021

6.1 Riprendiamo sullo spazio di memoria

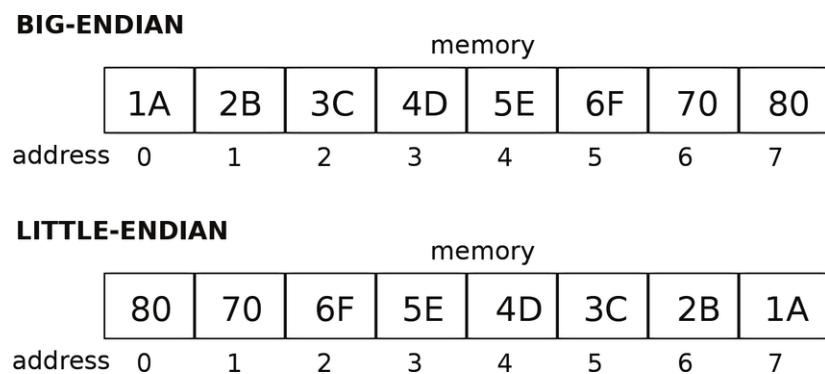
La differenza principale rispetto al Manchester Baby è il fatto che nelle memorie moderne è possibile accedere sia al byte che a multipli del byte. Questo comporta due questioni.

6.1.1 *endianes*

Abbiamo visto nel Manchester Baby una disposizione "strana" dei bit. Finché lavoriamo sulle singole righe, come nel Manchester Baby, non è un problema. Se iniziamo a lavorare su multipli e sottomultipli (come in tutte le architetture moderne) diventa necessario conoscere nel dettaglio come sono disposti i vari byte. Si distinguono¹

- *little-endian*: il byte meno significativo si trova all'indirizzo più piccolo;
- *big-endian*: il byte più significativo si trova all'indirizzo più piccolo.

Rappresentazione dell'indirizzo 0x1A2B3C4D5E6F7080



Quando nasce il problema "si fa in un modo o in un altro"? In due occasioni:

- nascita delle reti, quindi necessità di far comunicare dispositivi con approccio diverso;
- nascita dell'esigenza di portare un sistema operativo da un'architettura a un'altra.

¹Satira derivante dai viaggi di Gulliver, il nome è diventato popolare in letteratura.

I sistemi operativi nascono normalmente per un'architettura, e sono scritti in Assembler (che è *processor-specific*). UNIX è uno dei primi sistemi operativi scritti in C, rendeva possibile il passaggio da un'architettura a un'altra.

Vincitori di questa guerra santa?

- L'architettura Intel è realizzata con approccio *little-endian* (quindi noi dobbiamo ragionare così). Tutte le macchine moderne seguono questo approccio, tramite alcuni calculatori IBM professionali.

Tipi di accessi Accessi di tipo diverso, dato lo stesso indirizzo, restituiscono un contenuto diverso. Negli accessi a 16/32bit si utilizza l'indirizzo più piccolo delle 2/4 locazioni. L'indirizzo più grande contiene i bit più significativi, segue quanto presente nell'immagine

	b7	b0	Lettura a 8 bit (byte)	Indirizzo	Contenuto
0x3F65B432		0x1C		0x3F65B432	0x1C
0x3F65B433		0x39		0x3F65B432	0x391C
0x3F65B434		0xA2	16 bit (word)	0x3F65B432	0xC6A2391C
0x3F65B435		0xC6	32 bit (double word)	0x3F65B432	

- Sulla rete ha vinto *big-endian*. Segue che un processore Intel dovrà scambiare i byte prima di inviarli.

6.1.2 parallelismo

Quando il processore legge una parola di 8 byte il nostro interesse è leggerli tutti insieme, quindi *in parallelo*. Dobbiamo organizzare la memoria in modo tale che ciò sia possibile.

```
MOV %AL, 1000
MOV %AX, 1000
MOV %EAX, 1000
MOV %RAX, 1000
```

queste operazioni scrivono tutte allo stesso indirizzo, ma pongono un numero di byte differenti. **Vogliamo evitare un tempo di esecuzione proporzionale ai byte da considerare.** Un'operazione di lettura in memoria ha bisogno di più istruzioni oltre al semplice indirizzo: dobbiamo indicare anche il numero di byte coinvolti.

Come vengono codificate queste informazioni dalla CPU? Prendiamo lo spazio di memoria e dividiamolo in regioni naturali di 8 byte ciascuna (supponiamo che la dimensione delle nostre parole sia di 8 byte). Il processore specifica

- il numero di riga (l'identificativo della regione naturale, si ignora l'offset);
- otto *byte enabler* (/be0, /be1, /be2, /be3, /be4, /be5, /be6, /be7), uno per ciascuno dei byte all'interno della linea selezionata). Attraverso questi dico, data una riga formata da 8 byte, quali byte mi interessano.

Esempi di accessi

- MOV %AL, 0x3f4

pongo /be4 = 0 e tutti gli altri uguali ad 1

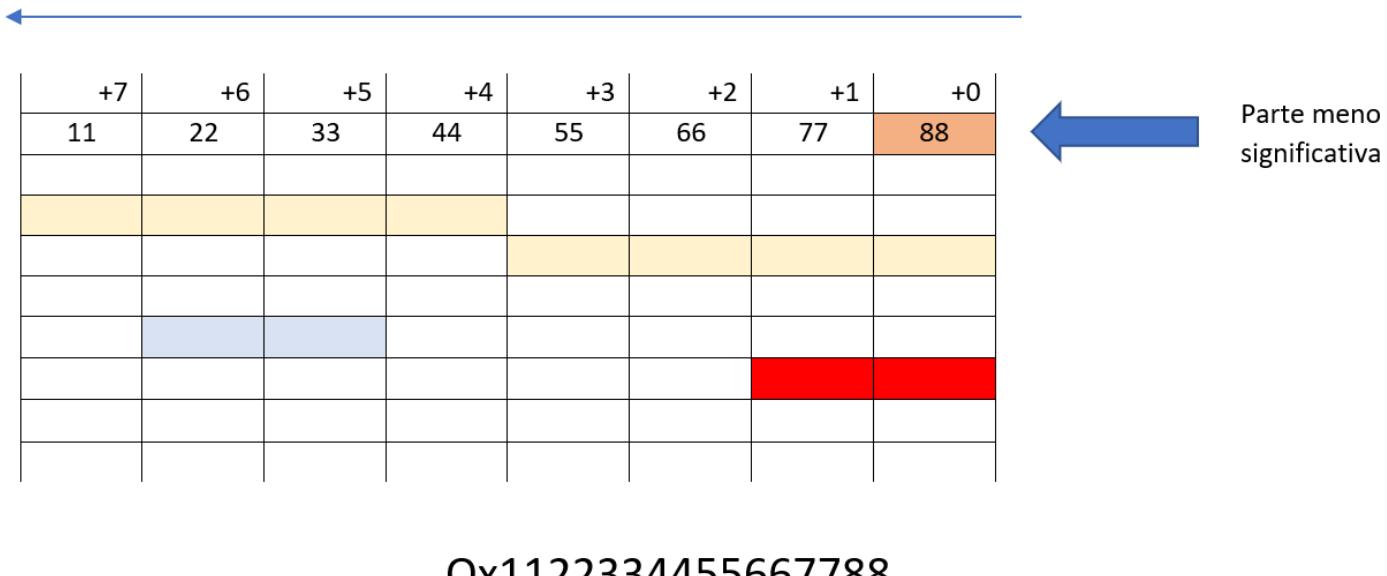
- MOV %AX, 0x3f4

pongo /be4 = /be5 = 0 e tutti gli altri uguali ad 1

- MOV %AX, 0x3f7

si esce dalla regione naturale. Necessario svolgere due accessi a due regioni di memoria diverse: nel primo accesso ho /be7 = 0, nel secondo /be0 = 1. Nei processori Intel la cosa è gestita automaticamente dal processore.

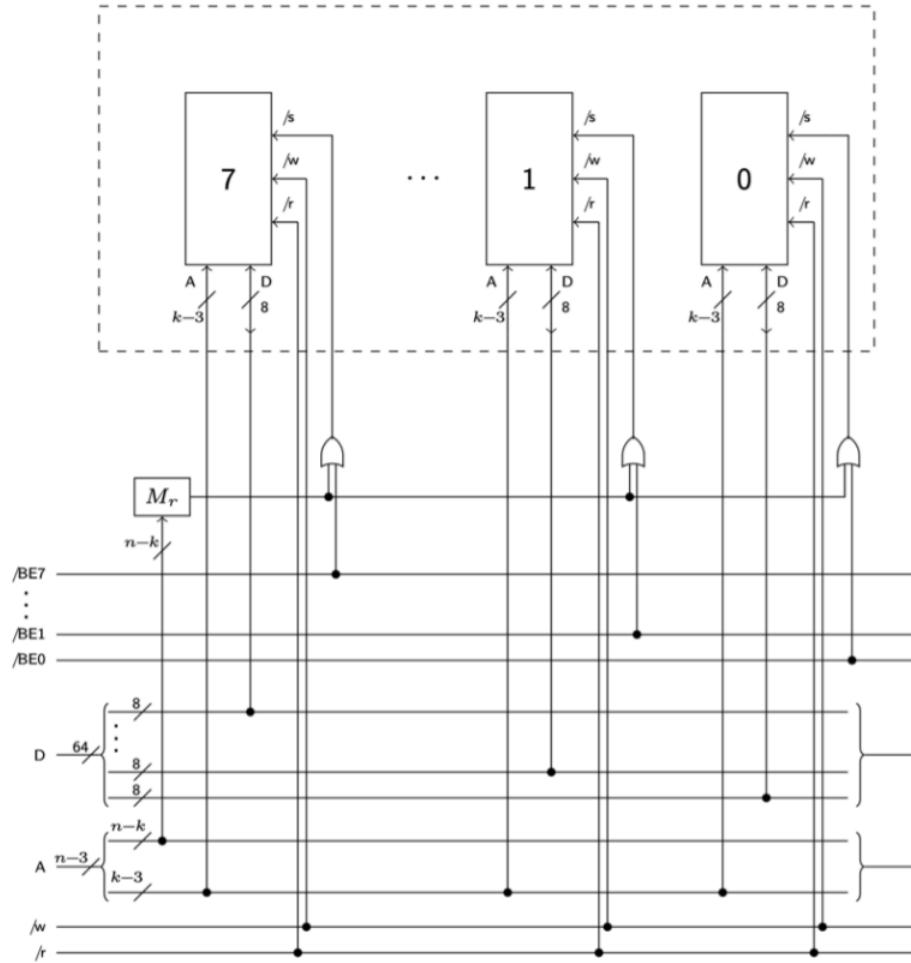
6.1.2.1 Immagine dello spazio di memoria



Ci conviene immaginare lo spazio di memoria come una sequenza di byte, ma organizzata in righe. Per via dell'*endianess* conviene organizzare gli elementi nella direzione indicata nell'immagine. Supponiamo di avere il numero posto in fondo all'immagine: porremo le cifre meno significative nei posti con offset minore.

Importanza dell'allineamento Si capisce dall'ultimo esempio di accesso l'importanza dell'allineamento. Allineamento significa piazzare gli oggetti a confine di regioni grandi quanto l'oggetto. Fare questo ci garantisce che gli oggetti saranno contenuti in una stessa riga (un singolo accesso richiede centinaia di clock, quindi è importante non avere cose disallineate).

6.1.2.2 Organizzazione della RAM



La RAM deve essere organizzata per svolgere queste operazioni in parallelo.

- Non possiamo collegare direttamente al bus un semplice modulo RAM con piedino di select, di lettura, e scrittura, fili di indirizzo e fili di dati. Ne dovremo collegare tanti quanti i byte che costituiscono l'intervallo. Ogni modulo rappresenta una colonna dello spazio di memoria.
- I moduli possono essere considerati alla pari delle RAM statiche viste a Reti logiche, ma dobbiamo tenere conto che le memorie moderne non sono fatte in quel modo. Approfondiremo la struttura circuitale delle memorie centrali moderne ad *Elettronica digitale*.
- Abbiamo in ingresso:
 - n fili di indirizzo (SOLO per il numero di riga, non ho l'indirizzo completo);
 - le variabili *byte enabler*, che non vanno in ingresso a nessun circuito combinatorio (la logica combinatoria viene gestita dal processore);
 - 64 fili di dati.
- I fili di dati sono ottenuti unendo insiemi di fili di dati: abbiamo 8 fili provenienti da ciascun modulo (tanti quanti i bit che compongono il byte).

- Del numero di riga si fanno entrare:
 - i k bit meno significativi in ciascun modulo (dobbiamo dire in ciascun modulo quale elemento della colonna mi interessa, cioè quale riga);
 - i bit rimanenti in una maschera che restituisce 0 se la regione che vogliamo visitare si trova nel modulo RAM (non i sottomoduli, il modulo nel complesso).

Ricordarsi, relativamente alla maschera, che lavoriamo con attivi bassi.

- Il valore per ciascun piedino di select è ottenuto da una porta OR che ha in ingresso l'uscita della maschera e il *byte enabler* relativo².

In sostanza

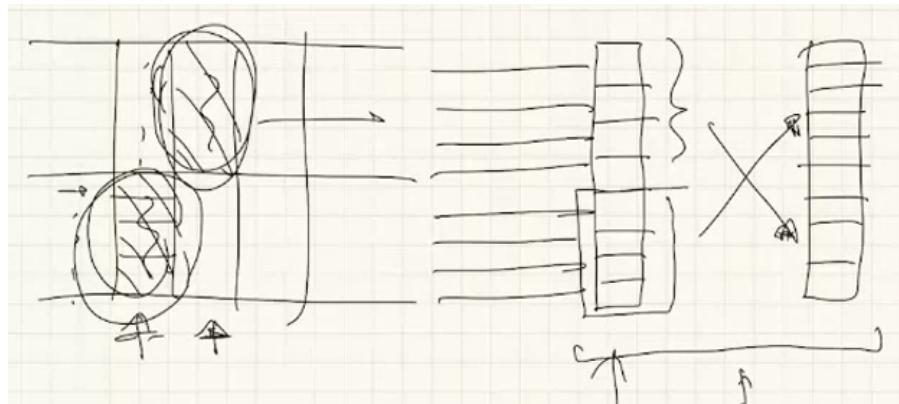
- Non si considerano i tre bit meno significativi, poichè abbiamo già i piedini *byte enabler* a indicare quali posizioni ci interessano.
- Del numero di rigione si prendono i k bit meno significativi per determinare, in una colonna della RAM quale byte effettivamente ci interessa (l'offset dice cosa ci interessa orizzontalmente, i k bit ciò che ci interessa verticalmente).
- I rimanenti bit, quelli più significativi, vanno in una maschera che determina se la RAM è interessata o no dall'operazione di lettura/scrittura (ricordare che tutte le componenti dell'architettura sono connesse sul bus, tutte ricevono e risponde solo chi è chiamato in causa).

6.1.2.3 Osservazioni su lettura disallineata

Prendiamo la seguente operazione

`MOV 0x3ff, %RAX`

Il processore non fa solo due operazioni di lettura, ma deve anche riordinare gli elementi. Nella prima lettura avremo la parte meno significativa, nella seconda la più significativa. Dopo aver eseguito le due operazioni troveremo le parti invertite: la parte meno significativa nei byte più significativi del buffer, e così via. L'operazione è veloce e legata all'hardware (l'azione è eseguita dal processore e intrinseca nella MOV).



²La dispensa di Lettieri pone la versione da me scritta qua (a mio parere la più intuitiva). Durante la spiegazione ha parlato di maschera con a valle una porta NOT, e di porte AND aventi in ingresso l'uscita negata della maschera e il relativo byte enabler.

Modifica di singoli bit Per modificare un singolo bit dobbiamo

- Leggere l'intero byte
- Applicare una maschera con operazione logica (scegliamo la maschera in modo tale che si vada a modificare un solo bit)
- Scrivo il byte aggiornato

```
MOV 0x3ff, %AL  
ORB $0x08, %AL <----- 00001000 OR %AL  
MOV $AL, 0x3ff
```

In questo caso la modifica del singolo bit avviene via software. Per i dettagli ricordarsi gli esempi di operazioni viste a Reti logiche con le istruzioni macchina AND/OR/XOR.

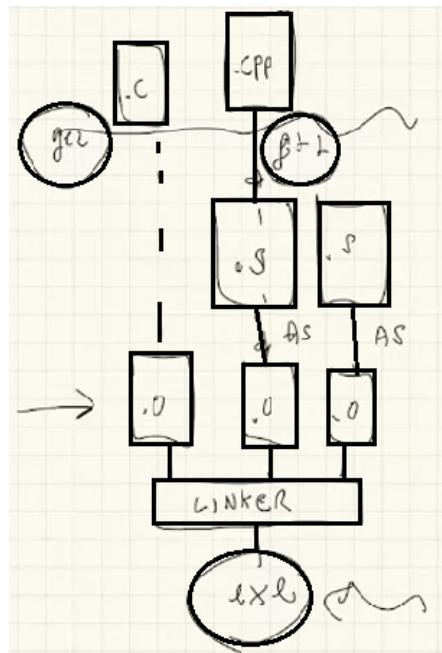
Formato di istruzioni e indirizzamento immediato Il fatto che sorgente e destinatario non possano essere entrambi indirizzati in modo immediato è dovuto al formato di istruzioni: abbiamo spazio soltanto per un offset. L'unica alternativa è l'utilizzo di registri puntatori (nulla di nuovo rispetto a Reti logiche).

Capitolo 7

Martedì 09/03/2021

7.1 Iniziamo ad uccidere il compilatore

- Il file .o è un file di sistema che non dipende dal linguaggio compilato: posso partire da files .s, .cpp, .c...



- Abbiamo due opzioni:
 - scrivere un file in c o in c++, per esempio, passandolo da un compilatore che restituirà un file assembler;
 - scrivere direttamente del codice assembler evitando il compilatore.
- I comandi gcc e g++ non sono veri e propri compilatori, ma dei *front-end*: capiscono quali strumenti devono utilizzare, e in che ordine, per ottenere l'eseguibile finale.

7.1.1 Esercizio dai lucidi del prof.Frosini

Testo Vogliamo realizzare un programma diviso in due parti:

- la prima col programma principale
- la seconda col sottoprogramma *esamina*, usato dal primo file.

Il programma

- legge caratteri fino al fine linea;
- per ogni carattere, oltre a stamparlo, chiama il sottoprogramma *esamina*, e stampa il risultato prodotto da quest'ultimo.

Il sottoprogramma esamina restituisce otto caratteri in codifica ASCII, corrispondenti agli 8 bit della codifica del carattere ricevuto.

Codice di esamina.s

```
.global esamina, alpha, beta

.data
alpha: .byte 0
.align 8
beta: .quad 0

.text
esamina:
    push %rax
    push %rdx
    push %rcx

    mov alpha(%rip), %cl
    movabs beta, %rax
    mov $0, %rdx
prossimo:
    test $0b10000000, %cl
    jz zero

    movb $'1', (%rax, %rdx)
    jmp avanti
zero:
    movb $'0', (%rax, %rdx)
    jmp avanti
avanti:
    shl %cl
    inc %rdx
    cmp $8, %rdx
    jb prossimo

    pop %rcx
    pop %rdx
    pop %rax
    ret
```

Codice di codifica.s

```
.include "ser.s"
.global _start

.data
buffer:
.fill 8,1

.text
_start:
    call tastiera
    cmp $'\n', %al
    je fine
    call video
    mov %al, alpha
    lea buffer, %rax
    mov %rax, beta(%rip)
    call esamina
    mov $' ', %al
    call video
    mov $0, %rcx
ancora:
    mov buffer(%rcx), %al
    call video
    inc %rcx
    cmp $8, %rcx
    jb ancora

    mov $'\n', %al
    call video
    jmp _start
fine:
    call uscita
```

Riflessioni

- **Trasmissione dei dati fra programma e sottoprogramma.** Dobbiamo utilizzare due variabili *alfa* e *beta* definite nel secondo file (*extern* nel primo e *global* nel secondo). La prima contiene il codice del carattere che il sottoprogramma deve esaminare, la seconda contiene l'indirizzo di una variabile array di 8 byte, dove il sottoprogramma deve porre il risultato. Il programma principale pone i dati in *alfa* e *beta*, quindi chiama *esamina*.
- Abbiamo utilizzato una libreria contenuta nel file *ser.s*. Per il momento il contenuto di questi sottoprogrammi non ci interessa (ne ripareremo più avanti).
- **Direttive *global*.** Nel file *esamina.s* le variabili *alfa* e *beta* sono dichiarate globali dalla seguente direttiva

```
.global esamina, alpha, beta
```

se non facciamo questo l'altro file non potrà usare queste etichette. Ricordarsi che il collegatore vede solo ciò che è *global*.

- **Reminiscenza di Reti logiche.** La PUSH e la POP devono essere utilizzate in modo adeguato (eseguo le push all'inizio del sottoprogramma, le pop alla fine del programma, inoltre per determinare l'ordine delle istruzioni POP considero l'ordine delle PUSH).

```
push %rax  
push %rdx  
push %rcx
```

```
[...]
```

```
pop %rcx  
pop %rdx  
pop %rax  
ret
```

- **Cosa succede se non eseguo le istruzioni pop alla fine?** L'istruzione RET alla fine del sottoprogramma *esamina* prende l'indirizzo che sta in cima alla pila: il problema è che abbiamo eseguito la push altre tre volte dopo la chiamata del sottoprogramma, quindi l'elemento in cima alla pila non è quello che dovrebbe usare la RET. Se l'indirizzo considerato (quello che si cerca di trattare come indirizzo) ci porta a un'area non assegnata al programma otterremo l'errore di *segmentation fault*.
- **Cosa succede se poniamo le POP in ordine non consueto?** I registri non vengono riportati al loro valore originario. L'errore non viene segnalato, ma può provocare risultati indesiderati.
- **Sintassi per gli indirizzi.** Attenzione alla rip

```
mov alpha(%rip), %cl
```

Siamo certi che questo indirizzamento ci permette di evitare problemi. La cosa non è necessaria in questo caso: il programma è molto piccolo.

- **Istruzione *test* in *esamina.s*.** L'istruzione

```
test $0b10000000, %cl
```

equivale all'istruzione AND, ma non viene modificato il destinatario. L'unica cosa modificata sono i flag.

- **Sostituzione automatica della *mov* con la *movabs*.** Se poniamo qualcosa che non è rappresentabile su 32 bit

```
mov $0x12345668, %rax
```

l'assemblatore adotta automaticamente la movabs (il destinatario è un registro, nient'altro). Provare per credere!

- **Azzeramento della parte alta di un registro a 64 bit.** Se il registro è a 32 bit la parte alta viene sempre azzerata (novità). Segue che le seguenti istruzioni

```
mov $0, %edx  
mov $0, %rdx
```

avranno lo stesso effetto

- **Natura dell'operando destinatario** (banalità importante). Possiamo scrivere...?

```
cmp %rdx, $8
```

No: la destinazione è per natura un registro o un indirizzo in memoria (sempre, anche se l'istruzione non ci scrive)

- **Domanda da pretest di Reti logiche.** Cosa fa la seguente istruzione..?

```
mov $beta, %rax
```

La stessa cosa della LEA: pone come contenuto del registro rax l'indirizzo relativo all'etichetta *beta*. Possiamo fare la stessa cosa così:

```
lea beta, %rax
```

- **Allineamento e disallineamento.** Attenzione ai dati

```
.data  
alpha: .byte 0  
beta: .quad 0
```

Sono disallineati: *alpha* si trova all'offset 0, *beta* all'offset 1. Segue che un byte di *beta* si troverà in un'altra regione naturale (dunque sono necessari due accessi). L'assemblatore non interviene, a meno che non poniamo la seguente direttiva

```
.data
    alpha: .byte 0
    .align 8
    beta:   .quad 0
```

La direttiva mi permette *di scartare i prossimi elementi in modo tale da arrivare al prossimo multiplo di 8*.

Alfa e beta disallineati

						beta	alfa

Alfa e beta allineati

							alfa
							beta

- **Etichette esterne.** L'assemblatore per default assume che tutte le etichette non definite in un file siano *esterne*, cioè definite in un file esterno. Possiamo indicarlo in modo esplicito con una direttiva

```
.extern alpha
```

Il linker, che ha visione completa (in contrasto all'assemblatore), inchioda se si accorge che certe etichette non sono state definite.

- **Cosa succede se non uso global per segnalare ad altri files ulteriori etichette?** Si segnala errore di *undefined reference* dopo aver avviato il linker. L'etichetta è definita (si veda *nm*), ma non viene vista nell'altro file.
- **Cosa succede se dichiaro un'etichetta beta globale in esamina.s e ne introduco una con lo stesso nome in codifica.s?** Se un file presenta un'etichetta con un certo nome e questa viene usata nel file stesso non c'è motivo per andarla a cercare altrove. Quella che avviene è una *sovraposizione* rispetto alla variabile globale dichiarata altrove.

Capitolo 8

Giovedì 11/03/2021

8.1 Programmi misti C++/Assembler

Oggi vogliamo cominciare a scrivere programmi misti, cioè scritti in parte in C++ (passati dal compilatore) e in parte in Assembler.

8.1.1 g++ e *startfiles*

Il g++ è un front-end per i vari strumenti preconfigurato per programmi scritti in C++. Capisce cosa deve fare e chiama opportunamente compilatore e/o assemblatore (in base ai files che gli passiamo)

```
g++ -o nome_file_output -no-pie file1, file2, ...
```

creiamo un eseguibile con nome *codifica* a partire dai files indicati nel comando. Il parametro *-no-pie* viene posto per evitare degli errori (non approfondiremo il perché).

Proviamolo Proviamo ad eseguire g++ con il programma scritto durante la scorsa lezione

```
g++ -o codifica -no-pie codifica.s esamina.s
```

creiamo un eseguibile con nome *codifica* a partire dai file *codifica.s*, *esamina.s*. Il terminale ci segnala errore:

```
studenti@debian-ce-2:~/Desktop$ g++ -o codifica -no-pie codifica.s esamina.s
/usr/bin/ld: /tmp/ccrwjSuu.o: in function `__start':
(.text+0x64): multiple definition of `__start'; /usr/lib/gcc/x86_64-linux-gnu/8/
. . . . . /x86_64-linux-gnu/crt1.o:(.text+0x0): first defined here
/usr/bin/ld: /usr/lib/gcc/x86_64-linux-gnu/8/ . . . . /x86_64-linux-gnu/crt1.o: i
n function `__start':
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

Come mai? g++ aggiunge, senza dire nulla, degli *start files* che fanno delle inizializzazioni (fatte prima della partenza del programma vero e proprio): in questi file oggetto aggiuntivi viene definito *_start*, che fa un po' di cose e chiama, alla fine, la funzione *_main*. Possiamo evitare questi inserimenti ponendo un ulteriore parametro nell'istruzione

```
g++ .nostartfiles -o codifica -no-pie codifica.s esamina.s
```

Se facciamo questa cosa col problema visto nella scorsa lezione tutto funziona regolarmente.
Non avendo utilizzato cose non nostre non ci sono problemi nell'ignorare gli *startfiles*.

Cosa potrebbe fare la start?

- Definire strutture e classi (se io dichiaro un'istanza globale di una classe dovrò avere già definita la classe al momento dell'esecuzione di main)
- Inizializzazione di oggetti cin e cout (che, va be, sono classi)
- Esecuzione dei distruttori

Standard C++ Definiamo *main* invece di *start* e poniamo RET alla fine del programma.
Abbiamo una chiamata del sottoprogramma *main* all'interno di *start*: dopo che *main* ha restituito il controllo la *start* esegue quanto necessario per concludere l'esecuzione del programma, incluso i distruttori.

8.1.2 g++ e *overloading*

Scriviamo l'analogo in C++ di codifica.s

```
#include <iostream>

extern char alpha;
extern char* beta;
extern void esamina();

char buffer[8];

int main() {
    while(true) {
        char c;
        std::cin.get(c);

        if(c == '\n')
            break;

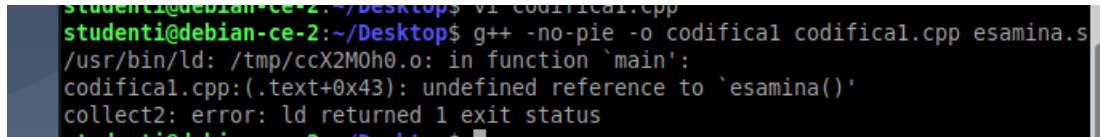
        alpha = c; <--- devo dire al compilatore chi e' alpha           [LO FACCIO SOPRA]
        beta = buffer; <--- devo dire al compilatore chi e' beta

        esamina(); <-- devo dire al compilatore chi e' esamina
        for(int i = 0; i < 8; i++) {
            std::cout << buffer[i];
        }
        std::cout << "\n";
    }
}
```

Se eseguiamo il codice

```
g++ -no-pie -o codifica1 codifica1.cpp esamina.s
```

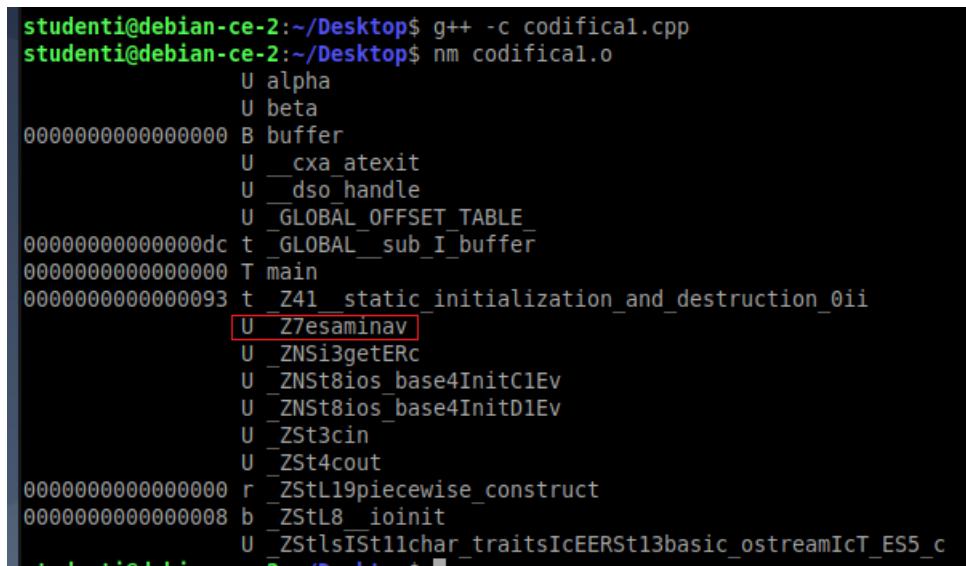
si lamenta perché non trova *esamina*.



```
studenti@debian-ce-2:~/Desktop$ vi codifica1.cpp
studenti@debian-ce-2:~/Desktop$ g++ -no-pie -o codifica1 codifica1.cpp esamina.s
/usr/bin/ld: /tmp/ccX2M0h0.o: in function `main':
codifica1.cpp:(.text+0x43): undefined reference to `esamina()'
collect2: error: ld returned 1 exit status
```

Vediamo cosa succede controllando gli elementi definiti in codifica1.cpp

```
g++ -c codifica1.cpp <---- mi limito ai files oggetto
nm codifica1.o <---- leggo simboli definiti
```



```
studenti@debian-ce-2:~/Desktop$ g++ -c codifica1.cpp
studenti@debian-ce-2:~/Desktop$ nm codifica1.o
U alpha
U beta
0000000000000000 B buffer
    U __cxa_atexit
    U __dso_handle
    U __GLOBAL_OFFSET_TABLE__
00000000000000dc t __GLOBAL_sub_I_buffer
0000000000000000 T main
0000000000000093 t __Z41_static_initialization_and_destruction_0ii
    U __Z7esaminav
    U __ZNSt3getERC
    U __ZNSt8ios_base4InitC1Ev
    U __ZNSt8ios_base4InitD1Ev
    U __ZSt3cin
    U __ZSt4cout
0000000000000000 r __ZStL19piecewise_construct
0000000000000008 b __ZStL8_ioinit
    U __ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_E5_c
```

troviamo *esamina*, ma con un nome strano.

Causa Il compilatore storicamente ha sempre usato gli stessi collegatori del C. Sappiamo che differenza sostanziale tra C e C++ è la presenza dell'overloading nel secondo (cioè funzioni con nome uguale e parametri diversi). Nel nostro caso l'etichetta per *esamina* è la seguente

`_Z7esaminav`

`Z` è un prefisso obbligatorio, 7 è il numero di caratteri del nome della funzione, v significa void (cioè assenza di parametri). Non possiamo gestire più funzioni con lo stesso nome mediante le stesse etichette, dobbiamo differenziarle!

Soluzione

- O modifico l'extern ponendo quel nome strano (vedremo più avanti le convenzione relative ai nomi delle etichette in C++)

```
extern void _Z7esaminav();
```

- O pongo l'extern nel seguente modo

```
extern "C" void esamina();
```

esiste *una funzione esamina che non ha argomenti, posta in un altro file, e scritta in C* (ok, non è scritta in C, ma rispetta lo standard del C). Nel C non esiste l'overloading, quindi le etichette assumono il nome che ci aspettiamo.

Chiaramente non serve questa cosa per le altre extern: non esiste l'overloading per le variabili.

Analogo in C++ di esamina.s

```
char alpha;
char* beta;

extern "C" void esamina() { <-- extern serve solo per poter indicare "C",
                           <-- altrimenti abbiamo il problema di prima
    char c = alpha;
    for(int i = 0; i < 8; i++) {
        if(c & 0x80) {
            beta[i] = '1';
        }
        else {
            beta[i] = '0';
        }
        c << 1;
    }
}
```

E se volessimo solo l'assembler? Poniamo l'istruzione così

```
g++ -fno-PIC -S codifica1.cpp <-- genero l'assembler
vi codifica1.s <---- apro l'assembler
```

Il codice ottenuto presenta le istruzioni Assembler che ci aspettiamo, più delle righe di codice contenenti principalmente informazioni (utili, per esempio, al debugger).

8.1.3 Uso dei registri

Vogliamo trovare un compromesso tra funzione chiamante e funzione chiamata: in particolare, vogliamo garantire al chiamante un certo numero di registri.

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of vector registers used; 1 st return register	No
%rbx	callee-saved register	Yes
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12-r14	callee-saved registers	Yes
%r15	callee-saved register; optionally used as GOT base pointer	Yes
%rdi	used to pass 1 st argument to functions	No
%rsi	used to pass 2 nd argument to functions	No
%rdx	used to pass 3 rd argument to functions; 2 nd return register	No
%rcx	used to pass 4 th integer argument to functions	No
%r8	used to pass 5 th argument to functions	No
%r9	used to pass 6 th argument to functions	No

- Alcuni registri sono detti *scratch*, cioè possono essere usati liberamente dalla funzione chiamata senza dover salvare il contenuto pre-esistente:
 - RAX, R10, R11, RCX, RDX, RSI, RDI, R8, R9 (Per ricordare: A,C,D,S,8,9,10,11).
- I registri rimanenti sono registri *non scratch*, garantiti al chiamante (quindi non vengono toccati dalla funzione chiamata): RBP, RBX, R12, R13, R14, R15.

- **Conseguenza:** dobbiamo stare attenti, il contenuto dei registri scratch non viene mantenuto in caso di chiamate di funzione. Noi dobbiamo preoccuparci esclusivamente dei nostri contenuti, dunque ricorrere alle istruzioni PUSH e POP se non vogliamo perdere il valore di alcuni registri scratch a seguito di chiamata di funzione.

8.1.4 Rappresentazione dei dati

La cosa è molto intuitiva, ma c'è uno standard da eseguire.

8.1.4.1 Tipi fondamentali

Sono disponibili molte informazioni su cppreference.com alla voce *Fundamental types*.

Type specifier	Equivalent type	Width in bits by data model				
		C++ standard	LP32	ILP32	LLP64	LP64
<code>short</code>	<code>short int</code>	at least 16	16	16	16	16
<code>short int</code>						
<code>signed short</code>						
<code>signed short int</code>						
<code>unsigned short</code>						
<code>unsigned short int</code>						
<code>int</code>	<code>int</code>	at least 16	16	32	32	32
<code>signed</code>						
<code>signed int</code>						
<code>unsigned</code>						
<code>unsigned int</code>						
<code>long</code>	<code>long int</code>	at least 32	32	32	32	64
<code>long int</code>						
<code>signed long</code>						
<code>signed long int</code>						
<code>unsigned long</code>						
<code>unsigned long int</code>	<code>unsigned long int</code> <code>long long int</code> <code>long long int</code> <code>signed long long</code> <code>signed long long int</code>	at least 64	64	64	64	64
<code>long long</code>						
<code>long long int</code>						
<code>signed long long</code>						
<code>signed long long int</code>						
<code>unsigned long long</code>	<code>unsigned long long int</code> <code>(C++11)</code>					
<code>unsigned long long int</code>						

Il sito mette insieme standard e implementazione: in particolare si osserva che lo standard da margini di libertà all'implementatore (si parla di *almeno*, non un numero ben preciso). Questo significa che altri documenti dovranno specificare per intero come avviene l'implementazione. Sono riportate alcune implementazioni comuni nello standard. In particolare ci interessano LLP64 ed LP64: la prima è per windows, la seconda per i sistemi Unix e Unix-like (Linux, macOs).

char Cosa strana è la presenza di tre tipi diversi:

- `signed char`

tipica rappresentazione dei char in x86, x64.

- `unsigned char`

- `char`

la rappresentazione sarà equivalente a `signed char` o `unsigned char`, ma rimane un tipo diverso.

Approfondiamo l'implementazione Il documento che specifica tutto ciò che lo standard non dice (in Linux) è il *System V Application Binary Interface*. Per quanto riguarda la rappresentazione dei dati abbiamo una tabella con tipi, dimensioni e allineamento. Caratteristica dei tipi fondamentali è avere dimensione e allineamento uguali.

Figure 3.1: Scalar Types

Type	C	<code>sizeof</code>	Alignment (bytes)	AMD64 Architecture
Integral	<code>_Bool[†]</code>	1	1	boolean
	<code>char</code>	1	1	signed byte
	<code>signed char</code>	1	1	
	<code>unsigned char</code>	1	1	unsigned byte
	<code>short</code>	2	2	signed twobyte
	<code>signed short</code>			
	<code>unsigned short</code>	2	2	unsigned twobyte
	<code>int</code>	4	4	signed fourbyte
	<code>signed int</code>			
	<code>enum^{††}</code>			
	<code>unsigned int</code>	4	4	unsigned fourbyte
	<code>long</code>	8	8	signed eightbyte
	<code>signed long</code>			
	<code>long long</code>			
	<code>signed long long</code>			
Pointer	<code>unsigned long</code>	8	8	unsigned eightbyte
	<code>unsigned long long</code>	8	8	unsigned eightbyte
Floating-point	<code>__int128^{††}</code>	16	16	signed sixteenbyte
	<code>signed __int128^{††}</code>	16	16	signed sixteenbyte
	<code>unsigned __int128^{††}</code>	16	16	unsigned sixteenbyte
	<code>any-type *</code>	8	8	unsigned eightbyte
<code>any-type (*)()</code>				
Decimal	<code>float</code>	4	4	single (IEEE-754)
	<code>double</code>	8	8	double (IEEE-754)
	<code>long double</code>	16	16	80-bit extended (IEEE-754)
	<code>__float128^{††}</code>	16	16	128-bit extended (IEEE-754)
<code>Decimal32</code>		4	4	32-bit RDN (IEEE-754R)

8.1.4.2 Tipi derivati

Quale dimensione e allineamento avranno i tipi derivati?

- **Puntatori:** hanno dimensione fissa al di là di cosa puntano, quindi valgono le cose già viste nella pagina precedente.
- **Array:**

`tipo a[DIM]`

- `sizeof`: $\text{DIM} * \text{sizeof}(\text{tipo})$
- `alignof`: `alignof(tipo)`

gli indirizzi crescenti, che partono da zero, mi rendono facile il calcolo dell'indirizzo di ogni elemento dell'array.

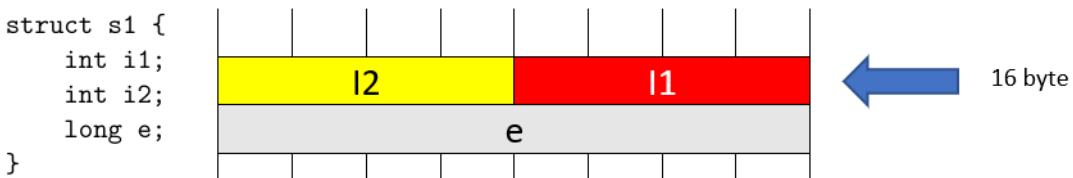
- **Strutture:** la cosa è un tantino più complicata.

```
struct s {
    tipo_1 f_1;
    tipo_2 f_2;
    tipo_3 f_3;
}
```

- `alignof`:

$$\max_i \{\text{alignof}(\text{tipo}_i)\}$$
- `sizeof`: non esiste una formula vera e propria, per prima cosa dobbiamo immaginarcì il *layout* della struttura. Consideriamo che:
 - * ogni elemento della struttura deve rispettare il proprio allineamento;
 - * i campi devono trovarsi in memoria **nell'ordine in cui sono dichiarati** (non esistono forme di ottimizzazione in cui viene alterato l'ordine degli elementi);
 - * la dimensione totale della struttura deve essere un multiplo dell'allineamento della struttura.

8.1.4.3 Primo esempio di struttura

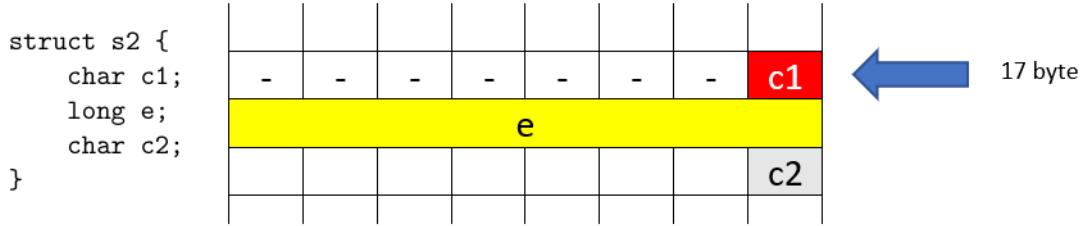


- Gli interi hanno dimensione 4 e allineamento 4. L'ordine in cui sono stati posti mi permette di porre i due interi nella stessa regione naturale. Il long ha dimensione 8 e allineamento 8. Dobbiamo allocarlo al primo indirizzo, multiplo di 8, successivo.

- **Conclusioni:**

- `alignof`: il massimo degli allineamenti è 8.
- `sizeof`: 16 byte

8.1.4.4 Secondo esempio di struttura



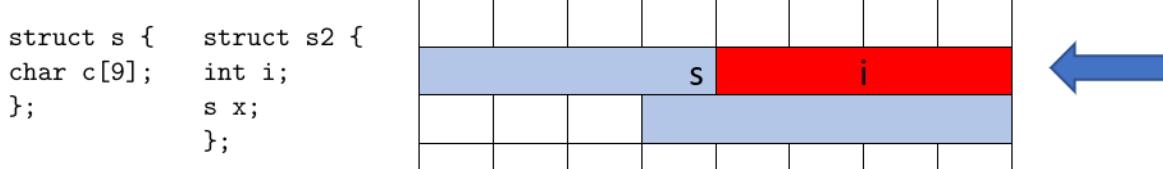
- Contrariamente a prima non possiamo mettere il char assieme a un altro elemento: nè con long (che ha dimensione di 8 byte), nè con char (che si trova dopo long). L'allineamento è stringente e impone che il long sia posto in un indirizzo multiplo di 8. Seguono 7 byte inutilizzati nella regione dove si trova il primo char.
- Dopo long mettiamo l'altro char, che non pone particolari vincoli. Rimangono 7 byte (che non faranno parte della struttura).
- **Conclusioni:**
 - *alignof*: 8 (il massimo tra gli alignof, il più alto è quello di long)
 - *sizeof*: 17 byte, più 7 byte non utilizzati.

perché non si può ottimizzare l'ordine degli elementi? Il compilatore non può ottimizzare l'ordine degli elementi per conto suo.

- Supponiamo di avere due files distinti. Il compilatore lavora distintamente su questi due.
- Se la struttura dipendesse dalle ottimizzazioni i due files potrebbero aver adottato ottimizzazioni diverse, quindi le due strutture non risulterebbero equivalenti.
- Se nei due files ci sono funzioni che comunicano tra di loro e che devono scambiarsi una struttura allora i due files devono concordare sul *layout* della struttura.

8.1.4.5 Terzo esempio di struttura

Prendiamo il seguente esempio



L'unica cosa che conta è l'ordine degli elementi. Si consideri che l'allineamento di ogni singolo char di *s* è 1, quindi l'allineamento della struttura *s* è 1. Possiamo porre i byte di *s* subito, senza passare subito a una nuova regione (niente ce lo vieta).

Capitolo 9

**Venerdì 12/03/2021, Lunedì
15/03/2021 e Martedì 16/03/2021**

9.1 Programmazione mista: funzioni

Per scrivere funzioni in C++ e chiamarle da Assembler, e viceversa, dobbiamo tener conto di una serie di regole, in particolare dobbiamo sapere come avviene il passaggio dei parametri (in ingresso e in uscita) e dove vengono poste le variabili locali. La soluzione adottata è sofisticata (non è la prima cosa che viene in mente a una funzione).

Cosa serve a una funzione? Una qualunque funzione avrà bisogno di:

- uno spazio per i parametri in ingresso, in uscita, e i risultati intermedi;
- uno spazio per le istruzioni.

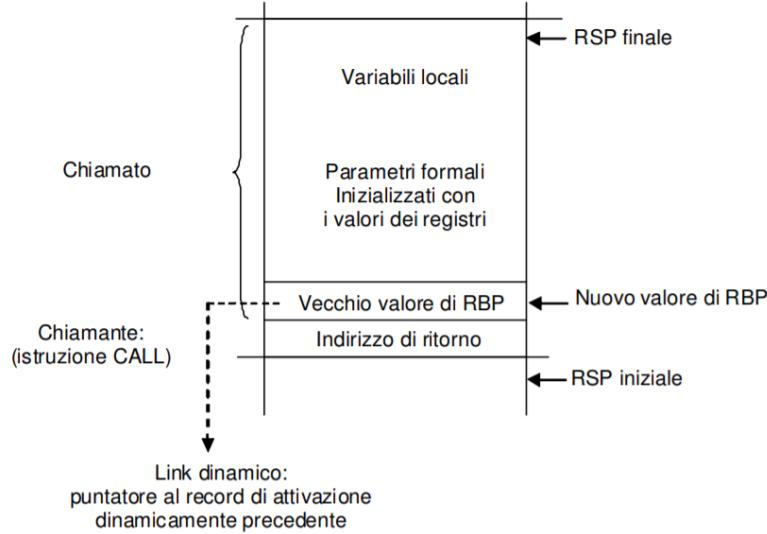
Preallocazione? La preallocazione dello spazio per parametri e risultati intermedi è svantaggiosa:

- si consuma troppo spazio;
- non è possibile fare chiamate ricorsive della stessa funzione;
- non è possibile condividere lo stesso codice in simultanea a più flussi.

Quindi? L'idea è di mettere questi parametri su una pila, ponendo il cosiddetto *record di attivazione*.

9.1.1 Record di attivazione

Ogni volta che si chiama una funzione si alloca nella pila il cosiddetto *record di attivazione della funzione*, che contiene variabili locali, parametri, e l'indirizzo di ritorno.



L'indirizzo di ritorno è cosa già vista: lo inserisco in pila con la CALL e lo rimuovo con la RET. La funzione accede ai vari campi del record di attivazione in modo indiretto. Dobbiamo tener conto che l'indirizzo di questi elementi non è fisso, considerando che potrei avere altre chiamate ricorsive prima di quella attuale che stiamo considerando. Una soluzione è calcolare l'offset rispetto al registro rsp

```
offset(%rsp)
```

rimane il fatto che se durante l'esecuzione del programma alteriamo la RSP allora gli offset dovranno essere ricalcolati. I processori moderni lo sanno fare senza grossi problemi.

Architettura Intel Nell'architettura Intel si tende a usare un registro esplicito che punti al record di attivazione senza usare rsp. Introduciamo il registro RBP (*Register Base Pointer*): per tutta la durata di esecuzione della funzione il valore di questo registro è costante e punta a un punto preciso del record di attivazione, subito sopra l'indirizzo di ritorno. Ad ogni chiamata di funzione viene memorizzato il valore precedente di rbp mediante PUSH.

```
PUSH %rbp <--- salvo il valore vecchio
MOV %rsp, %rbp <--- aggiorno il contenuto del registro
```

alla fine dell'esecuzione, se RSP è stato riportato al suo valore iniziale, diremo

```
POP %rbp <--- recupero il valore vecchio
RET
```

Nel caso in cui ciò non avvenga basta eseguire un'istruzione MOV (non ne avremo bisogno, basta rispettare le regole sull'uso di PUSH e POP). Per terminare il record di attivazione basta riservare spazio per i parametri e le variabili locali. Questo è semplice

```
SUB $spazio, %rsp
```

Sposto rsp in modo che successive push o chiamate di funzione proseguano oltre senza toccare questo spazio. La struttura del record di attivazione (relativamente a parametri e variabili locali) è flessibile: il suo contenuto interessa solo alla funzione.

9.1.1.1 Passaggio di parametri in ingresso

Altra cosa che deve fare il chiamante è passare i parametri. Chiaramente non può scriverli nel record di attivazione direttamente, visto che si muoverà prima che venga riservata l'area di memoria. Osserviamo le differenze tra le due architetture

- **Architettura a 32bit:** i parametri vengono posti nella pila prima di eseguire la call

- **Architettura a 64bit:** sfruttando il numero più alto di registri si passa i parametri direttamente con questi. Possiamo avere varie situazioni

- Pongo i parametri nei registri e li mantengo qua senza spostarli nell'aria di memoria
- Pongo i parametri nei registri e li sposto nell'area di memoria dedicata ai parametri e alle variabili locali. In quali occasioni sono costretto a fare così?
 - * Quando devo chiamare altre funzioni, oppure
 - * quando devo utilizzare istruzioni che ricorrono per forza a certi registri (moltiplicazione, divisione, istruzioni per le stringhe)

se la funzione ha più di 6 argomenti si adotta, per gli argomenti in eccesso, la strategia vista per l'architettura a 32bit. Si usano i seguenti registri, nell'ordine seguente...

- | | | |
|---------|---------|--------|
| 1. %RDI | 3. %RDX | 5. %R8 |
| 2. %RSI | 4. %RCX | 6. %R9 |

(D,S,D,C,8,9) Abbiamo già indicato questi registri come registri scratch: quali registri scratch possiamo usare dipenderà dalle situazioni.

Osservazione Argomenti diversi usano registri diversi. Se io ho, per esempio...

```
f(char c, char b);
```

non posso metterli nello stesso registro: c va in rdi, b in rsi.

Esempio di situazione in cui i parametri non vengono mantenuti nei registri

```
struct s {           int f(s x) {           g(s* p) {  
int i[4];             g(&x);             ...  
}                   }                 }  
}
```

Non possiamo tenere il parametro solo nei registri, visto che questi non hanno indirizzo. Devo spostare il contenuto della struttura in memoria, a quel punto, avrò qualcosa che potrò puntare a partire dalla mia funzione.

9.1.1.2 Parametro in uscita

La funzione termina: dove pongo il risultato? Abbiamo due possibilità

%RAX

%RDX_%RAX <--- (In caso di estensione)

O utilizzo solo il registro rax, oppure utilizzo anche il registro rdx (ponendo in rdx la parte più significativa di ciò che voglio restituire).

Hey Questa cosa dovrebbe accenderci la lampadina.

XOR %EAX, %EAX -----> return 0;

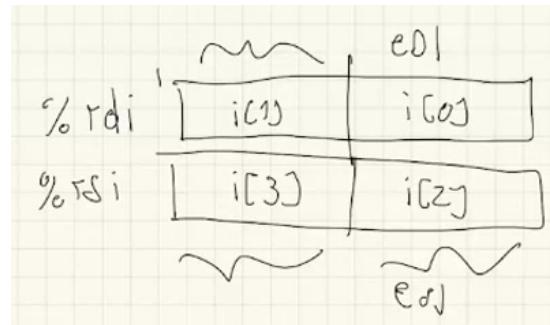
Stea ci ha detto che per convenzione si pone questa istruzione al termine del programma. Adesso capiamo perché!

Osservazione sulla dimensione dei parametri in ingresso e in uscita

- Se io passo in ingresso una struttura e questa è grande al più 16 byte, questa dovrà essere passata con uno o due registri (la prima riga va in rdi, la seconda in rsi).
- Se la struttura in ingresso è più grande di 16 byte la cosa è più complicata, ne ripareremo più avanti.

Stesso ragionamento vale nella restituzione (con i registri citati prima).

9.1.2 Esempio di struttura passata come parametro in ingresso



```
struct s {  
    int i[4];  
}  
int f(s x) {  
    int sum = 0;  
    for(int j = 0; j < 4; j++)  
        sum += x.i[j];  
    return sum;  
}
```

Il contenuto dell'elemento s sarà diviso tra rdi ed rsi. La cosa ottenuta non è molto comoda:

- i[0] e i[2] possono essere recuperati, rispettivamente, coi registri EDI ed ESI
- ma gli altri due elementi?

L'unica cosa possibile è shiftare.

9.1.3 Esercizio dalle diapositive del prof.Frosini

9.1.3.1 Passaggio di parametri per valore (pag.35)

Codice c++

```
// programma sommmaintGlob, file es1a.cpp      return 0;
#include "servi.cpp"
extern "C" int elab1(int n, int m);
int alfa, beta;
int main() {
    int ris;
    alfa = leggiint();
    beta = leggiint();
    ris = elab1(alfa, beta);
    scriviint(ris);
    nuovalinea();
}

// programma sommmaintGlob, file es1b.cpp
extern "C" int elab1(int n1, int n2) {
    int i, j;
    i = n1+n2;
    j = n1-n2;
    return i*j;
}
```

Funzione *elab1* in Assembler

```
.global elab1
elab1:
    push %rbp
    mov %rsp, %rbp
    sub $16, %rsp

    mov %edi, -8(%rbp)
    mov %esi, -4(%rbp)

    # i = n1 + n2
    mov %edi, -16(%rbp)
    add %esi, -16(%rbp)

    # j = n1 - n2
    mov %esi, %eax
    sub %edi, %eax
    mov %eax, -12(%rbp)

    # return i*j
    imull -16(%rbp), %eax

    leave
    ret
```

- Per prima cosa gestiamo il registro RBP

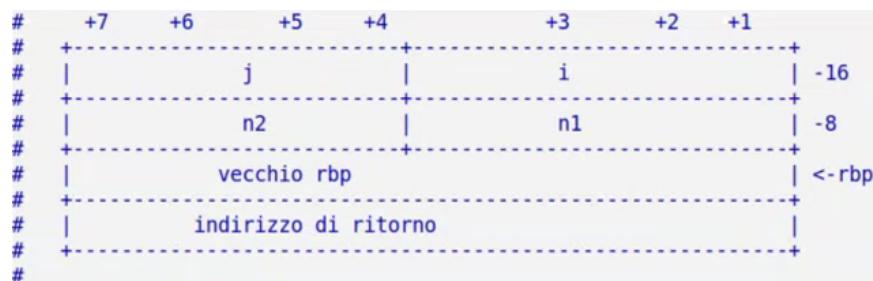
```
push %rbp
mov %rsp, %rbp
```

salviamo in pila il valore vecchio e aggiorniamo il registro con l'RSP attuale.

- Riserviamo lo spazio per le variabili locali e i parametri di ingresso

```
sub $16, %rsp
```

Riempiamo lo spazio appena allocato:



nell'indirizzamento si deve fare riferimento ad RBP (in questo caso potrei fare riferimento anche ad RSP, ma in generale è rischioso farlo). I registri source sono quelli che abbiamo citato, utilizzati nell'ordine indicato (in questo caso RDI ed RSI).

```
mov %edi, -8(%rbp)
mov %esi, -4(%rbp)
```

Abbiamo due *int* in ingresso, dunque bastano le parti meno significative a 32 bit.

- **Domanda.** Come capisco quale sia il destinatario di queste istruzioni? Mi basta sommare il numero a destra con quello in alto. Riguardo la seconda MOV la tentazione potrebbe essere scrivere quanto segue

```
mov %esi, -12(%rbp)
```

Ricordarsi che siamo in una pila, dunque andiamo indietro e non in avanti.



- Rappresentiamo l'addizione tra *n1* ed *n2* così

```
mov %edi, -16(%rbp)
add %esi, -16(%rbp)
```

Nell'indirizzo di *i*, cioè nell'indirizzo della variabile dove finisce la somma, procedo in due step:

1. pongo come contenuto *n1*, che sta in ESI;
2. sommo al contenuto la variabile *n2*, che sta in EDI.

- Rappresentiamo la sottrazione tra *n1* ed *n2*: so che il primo è il minuendo, e il secondo il sottraendo. Pongo il minuendo in EAX

```
mov %edi, %eax
sub %esi, %eax
mov %eax, -12(%rbp)
```

Il passaggio da EAX è necessario per fare la moltiplicazione poco dopo.

- Attenzione all'istruzione utilizzata: *i* e *j* sono due interi, non possiamo utilizzare la istruzione MUL per i numeri naturali (ignorerebbe il segno).

```
imull -16(%rbp), %eax
```

Novità!!!! La IMUL ha due operandi: si moltiplica il sorgente per il destinatario e si pone il risultato nel destinatario.

- Prima di concludere dobbiamo disfare quanto fatto nel prologo. Lo facciamo con la seguente istruzione

```
leave
```

adesso concludiamo con la solita istruzione

```
ret
```

Traduciamo anche il main

```
# int alfa, beta;
.data
alfa:
    .long 0
beta:
    .long 0

.text
.global main
main:
    push %rbp
    mov %rsp, %rbp
    sub $16, %rsp # int ris, tenendo conto delle cose dette (vedere sotto)

    # alfa = leggiint();
    call leggiint # risultato in %rax
    mov %eax, alfa(%rip)

    # beta = leggiint();
    call leggiint # risultato in %rax
    mov %eax, beta(%rip)

    # ris = elab1(alfa, beta);
    mov alfa(%rip), %edi
    mov beta(%rip), %esi
    call elab1
    mov %eax, -8(%rbp)

    # scriviint(ris);
    mov -8(%rbp), %edi
    CALL scriviint

    CALL nuovalinea

    xorl %eax, %eax

    leave
    ret
```

- Ci serve spazio per l'indirizzo di ritorno, il vecchio rbp e per l'intero *ris*.

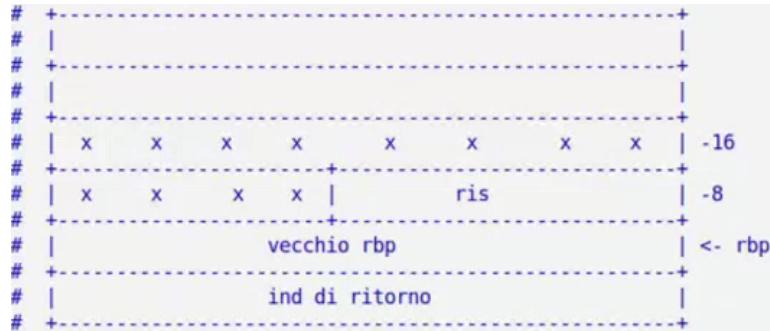
```

push %rbp
mov %rsp, %rbp
sub $X, %rsp

```

Attenzione al sub: che valore mettiamo?

- Non possiamo mettere 4: punterebbe a metà della regione naturale, con conseguente disallineamento di tutto ciò che finisce in pila successivamente. Risulta fondamentale mantenere l'allineamento a 8. La cosa viene fatta implicitamente dalla PUSH e dalla POP: il valore del registro RSP si sposta sempre di 8 byte.
- 8 è una soluzione migliore, tuttavia l'ABI (*System V Application Binary Interface*, la documentazione) richiede un allineamento ancora più stringente: 16! Dobbiamo garantire che prima della chiamata di funzione RSP sia sempre allineato a 16.
 - * Al momento della chiamata poco prima, l'rsp è multiplo di 16.
 - * Dopo aver posto l'indirizzo di ritorno l'RSP è multiplo di 8.
 - * Dopo aver posto il vecchio RBP avremo, nuovamente, RSP come multiplo di 16.
 - * Se vogliamo rispettare la regola dovremo mettere $X = 16$. Solitamente non si hanno errori, ma esistono istruzioni del processore che potrebbero lamentarsi in caso di RSP non multiplo di 16.
 - * *Sprecare un po' di memoria è considerato in genere meno importante rispetto al mantenere l'allineamento (cit.)*



- Attenzione alle variabili *alfa* e *beta*, che non solo sono presenti, ma anche globali (quindi vanno dichiarate).

```

.data
alfa: .long 0
beta: .long 0

```

- Chiamo la leggiint e sposto in *alfa* il risultato, che sappiamo essere nel registro rax

```

call leggiint
mov %eax, alfa(%rip)

```

- Stessa cosa con *beta*

```

call leggiint
mov %eax, beta(%rip)

```

- Metto in rdi il primo argomento e in rsi il secondo. Pongo inoltre il risultato in *ris*

```

mov alfa(%rip), %edi
mov beta(%rip), %esi
call elab1
mov %eax, -8(%rbp)

```

- Eseguo la scriviint ponendo *ris* nel registro edi (secondo lo standard)

```

mov -8(%rbp), %edi
CALL scriviint

```

- La funzione *main* restituisce 0 se tutto è andato per il verso giusto (un numero diverso da zero consiste nell'identificativo di un errore). Per restituire 0 utilizziamo l'istruzione XOR sul registro EAX (RAX, bastano 32 bit visto che si restituisce un *int*)

```
xorl %eax, %eax
```

- Concludiamo sbarazzandoci del prologo e restituendo il controllo

```

leave
ret

```

Osservazione Possiamo includere in un file assembler servi.cpp?

```
#include "servi.cpp"
```

Il compilatore si lamenta (*junk*). Non ha senso includere un file di un certo linguaggio in uno con un altro linguaggio.

Ottimizzazione L'ottimizzazione non si ottiene dalla riduzione del numero di istruzioni, ma da altre cose: scelta di istruzioni accurate, scritture in memoria solo quando necessario.

9.1.3.2 Passaggio di parametri per riferimento (pag.41)

Proviamo a fare una versione simile dell'esercizio precedente, ma con la presenza di un tipo di riferimento.

Codice c++

```

// programma sommmaintRif, file es3a.cpp
#include "servi.cpp"
extern "C" void elab3(int& tot, int n1, int n2);
int main() {
    int a, b; int ris; <----- a frosini e' sfuggita una &
    a = leggiint();
    b = leggiint();
    elab3(ris, a, b);
    scriviint(ris);
    nuovalinea();
}
// programma sommmaintRif, file es1b.cpp
extern "C" int elab3(int& tot, int n1, int n2) {
    int i, j;
    i = n1+n2;
    j = n1-n2;
    tot = i*j;
}

```

Traduzione Assembler di elab3

```
.text
.global elab3
elab3:
    push %rbp
    mov %rsp, %rbp
    sub $32, %rsp

    # copia tot al suo posto
    mov %rdi, -8(%rbp)

    # copia n1 ed n2 al suo posto
    mov %esi, -16(%rbp)
    mov %edx, -12(%rbp)

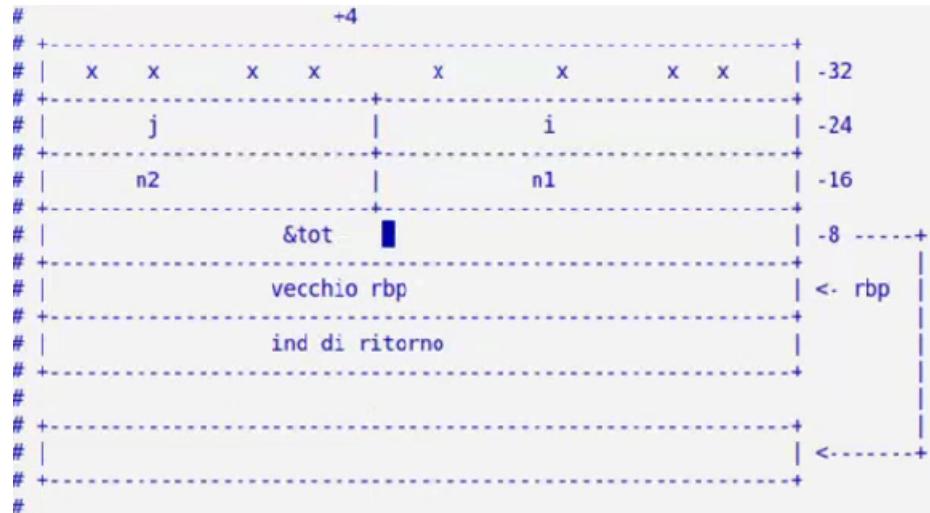
    # i = n1 + n2
    mov -16(%rbp), %eax
    add -12(%rbp), %eax
    mov %eax, -24(%rbp)

    # j = n1 - n2
    mov -16(%rbp), %eax
    add -12(%rbp), %ecx
    sub %eax, %ecx
    mov %ecx, -20(%rbp)

    # tot = i*j
    mov -24(%rbp), %eax
    imull -20(%rbp), %eax

    mov -8(%rbp), %rdx
    mov %eax, (%rdx)

    leave
    ret
```



- Pongo il vecchio rbp nella pila, aggiorno il valore di rbp e riservo 32 byte (32 invece di 24 per rispettare rsp multiplo di 16)

```
push %rbp
mov %rsp, %rbp
sub $32, %rsp
```

- Copio *tot*, così come tutte le altre variabili (secondo lo standard) *n1* ed *n2*.

```
mov %rdi, -8(%rbp)
mov %esi, -16(%rbp)
mov %edx, -12(%rbp)
```

- Sulla somma e sulla differenza non ci sono differenze importanti rispetto all'esercizio precedente.
- Recupero *i* ponendolo in eax. Teniamo conto che in &tot ci sarà l'indirizzo della variabile: sposto il contenuto della parte in un registro sporcabile, utilizzo l'indirizzamento indiretto per aggiornare il contenuto dell'indirizzo posto in rdx.

```
mov -24(%rbp), %eax
imull -20(%rbp), %eax

mov -8(%rbp), %rdx
mov %eax, (%rdx)
```

- La funzione non restituisce niente, quindi mi limito a terminarla con *leave* e *ret*.

```
leave
ret
```

Traduciamo anche il main

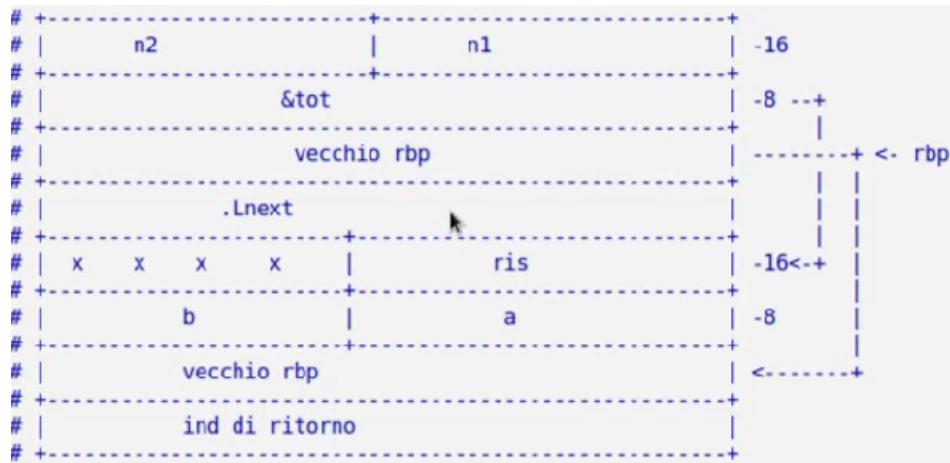
```
.text
.global main
main:
    push %rbp
    mov %rsp, %rbp

    call leggiint
    mov %eax, -8(%rbp)

    call leggiint
    mov %eax, -4(%rbp)

    lea -16(%rbp), %rdi
    mov -8(%rbp), %esi
    mov -4(%rbp), %edx
    call elab3

.Lnext:
    mov -16(%rbp), %edi
    call scrivint
    call nuovalinea
    leave
    ret
```



- Pongo il vecchio rbp nella pila, aggiorno il valore di rbp e riservo un certo numero di byte per *b*, *a* e *ris*.
- Le chiamate di leggiint non sono tanto diverse rispetto all'esercizio precedente
- Sposto nel registro rdi l'indirizzo della variabile *tot* usando la LEA. Sistemo anche le variabili *n1* e *n2*.

```
lea -16(%rbp), %rdi
mov -8(%rbp), %esi
mov -4(%rbp), %edx
call elab3
```

- Eseguo le funzioni rimanenti all'interno del *main*, nulla di particolare da segnalare.

9.1.3.3 Passaggio di parametro array (pag.50)

Codice c++

```
// programma array, file es6a.cpp          // programma array, file es6b.cpp
#include "servi.cpp"                      extern "C" void raddoppia(int a[], int n){
extern "C" void raddoppia(int a[], int n);    int i;
int main() {                                for(i = 0; i < n; i++)
    int ar[5];                            a[i] = 2*a[i];
    int i;                               }
    for(i = 0; i < 5; i++)
        ar[i] = leggiint();
    raddoppia(ar, 5);
    for(i = 0; i < 5; i++)
        scriviint(ar[i]);
    nuovalinea();
}
```

Traduzione assembler di raddoppia

```

.text
.global raddoppia
raddoppia:
    push %rbp
    mov %rsp, %rbp
    sub $16, %rsp

    mov %rdi, -8(%rbp)
    mov %esi, -16(%rbp)

    # inizializzazione (i = 0)
    mov $0, -12(%rbp)

.Lfor:   # Verifica della condizione (i < n)
    cmp %esi, -12(%rbp)
    jl .Lcorpo
    jmp .Lfine

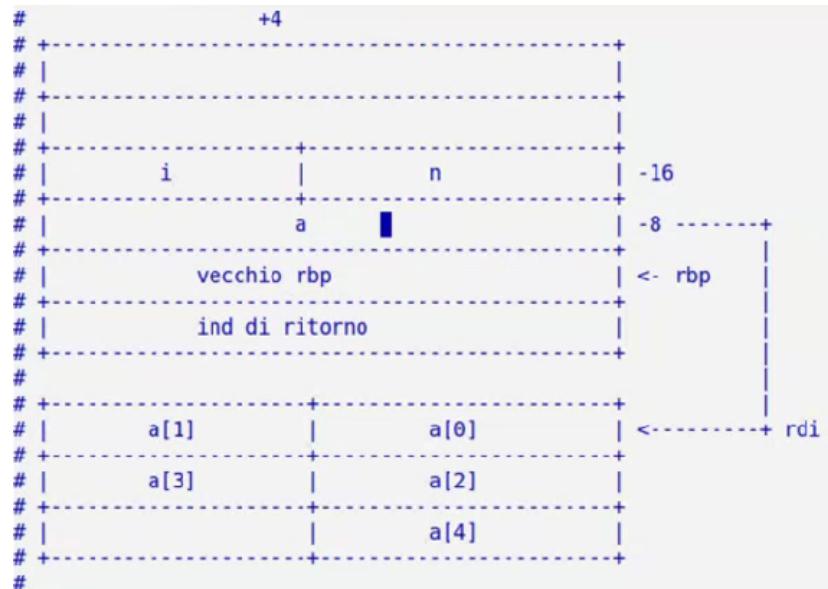
.Lcorpo:
    # caricare %rdx
    movslq -12(%rbp), %rdx

    mov (%rdi, %rdx, 4), %eax
    sal $1, %eax <--- (stesso OPCODE di SHL)
    mov %eax, (%rdi, %rdx, 4)

    # i++
    incl -12(%rbp)
    jmp .Lfor

.Lfine:  leave
        ret

```



- Pongo il vecchio rbp nella pila, aggiorno il valore di rbp e riservo un certo numero di byte per a , n ed i .

```

push %rbp
mov %rsp, %rbp
sub $16, %rsp

```

- Pongo i valori degli argomenti in memoria

```

mov %rdi, -8(%rbp)
mov %esi, -16(%rbp)

```

per quanto riguarda l'array *a* ricordiamoci che poniamo in memoria l'indirizzo del primo elemento dell'array. Ricordarsi che il primo elemento sta per forza negli indirizzi più bassi.

- Novità rispetto agli esercizi precedenti è l'introduzione di un for. Ricordiamo, a tal proposito, le spiegazioni sui cicli dalle dispense di Stea. In particolare, dobbiamo ricordare l'ordine delle operazioni in un for

1. inizializzazione del counter;

```
mov $0, -12(%rbp)
```

2. verifica della validità della condizione;

3. esecuzione del corpo del for (vedere dopo);

4. ritorno al secondo punto.

```
jmp .Lfor
```

- Dobbiamo caricare il registro rdx col valore del contatore, in modo da poterlo utilizzare per prendere l'elemento dell'array

```
movslq -12(%rbp), %rdx
```

con questa istruzione teniamo conto del segno nel passaggio da *long* a *quad* (se utilizzo solo la MOV emergono problemi con un numero elevato di iterazioni). perché dobbiamo dire questo? **Nell'indirizzamento coi registri dobbiamo utilizzare registri a 64 bit** (il compilatore fa storie in caso contrario), tuttavia

- **il numero coinvolto nello spostamento è un numero a 32 bit,**
- ed è anche intero.

Segue che **l'estensione di campo non è quella dei naturali** (in quel caso mi sarebbe bastato una semplice MOV con registri a 32 bit, considerato che i bit più significativi vengono azzerati)

- Sposto nel registro eax il valore da moltiplicare per due. Il prodotto è per una potenza della base due, quindi posso fare la cosa con uno shift a sinistra. Concludo riportando il numero shiftato in memoria.

```

mov (%rdi, %rdx, 4), %eax
sal $1, %eax <-- va bene anche shl (stesso opcode)
mov %eax, (%rdi, %rdx, 4)

```

per quanto riguarda l'indirizzamento bimodificato

$$\text{Indirizzo} = |\text{rdi} + \text{rdx} \times 4|_{\text{modulo}_{2^{64}}}$$

rdi è il registro avente per contenuto l'indirizzo del primo elemento dell'array (primo argomento della funzione), rdx è il registro dove poniamo ogni volta il contenuto della variabile contatore. Sapendo che ogni intero occupa 4 byte si capisce al volo che incrementando rdx passeremo all'elemento immediatamente successivo dell'array.

Traduciamo anche il main

```
.set ar, -24
.set i, -32

.text
.global main
main:
    push %rbp
    mov %rsp, %rbp
    sub $32, %rsp

    movl $0, i(%rbp) # <---- i = 0
.Lfor1:
    cmp $5, i(%rbp),
    jl .Lcorpo1 # <--- i < 5
    jmp .Lfine1
.Lcorpo1:
    call leggiint
    movslq i(%rbp), %rcx
    mov %eax, ar(%rbp, %rcx, 4)
    incl i(%rbp)
    jmp .Lfor1 # ritorno alla condizione
.Lfine1:
    lea -24(%rbp), %rdi
    mov $5, %esi
    call raddoppia

    movl $0, i(%rbp)
.Lfor2:
    cmp $5, i(%rbp)
    jl .Lcorpo2
    jmp .Lfine2
.Lcorpo2:
    movslq i(%rbp), %rcx
    mov ar(%rbp, %rcx, 4), %rdi
    call scrivint
    incl i(%rbp)
    jmp .Lfor2
.Lfine2:
    call nuovalinea

    leave
    ret
```

# +-----+	i	-32
# x x x x		
# ar[1]	ar[0]	-24
# ar[3]	ar[2]	
# x x x x	ar[4]	
# vecchio rbp		<- rbp
# ind ritorno		

- Nella creazione dell'array trattiamo ogni elemento come se fosse un qualcosa di indipendente. Li poniamo in modo tale che l'allineamento sia rispettato. Un modo per ricordarsi che l'array deve essere posto in un certo modo sono le istruzioni per visitare, in un ciclo, tutti gli elementi dell'array (somma sul puntatore ad array): se poniamo in fondo alla pila (negli indirizzi più alti) il primo elemento allora il meccanismo non funziona.

- **Direttive set.** Con le direttive *set* andiamo a creare delle costanti (già visto a Reti logiche, ricordarsi *pioppo* e la domanda del pretest).

```
.set ar, -24
.set i, -32
```

Possiamo assegnare SOLO delle costanti, non stiamo lavorando con delle macro.

```
.set i, -32(%rbp) <----- SBAGLIATO!
```

- Come al solito memorizziamo il vecchio rbp e allochiamo lo spazio necessario

```
push %rbp
mov %rsp, %rbp
sub $32, %rsp
```

- Poniamo in rdi l'indirizzo dell'array, che è l'indirizzo del primo elemento dell'array.

```
lea -24(%rbp), %rdi
```

non si utilizzi la mov invece della lea

```
mov -24(%rbp), %rdi
```

che sposta in memoria il contenuto del primo elemento dell'array, E NON L'INDIRIZZO DELLO STESSO!

- Poniamo anche il secondo argomento e chiamiamo la funzione *raddoppia*

```
mov $5, %esi
call raddoppia
```

- Il for per la stampa dell'array (legata alla funzione *scriviint*) non è strutturato in modo diverso dal primo.

- **Osservazione sui registri scratch:** Uno studente si è accorto di un errore del prof nell'uso del registro rcx. Il codice inizialmente era così

```
movslq i(%rbp), %rcx
call leggiint
mov %eax, ar(%rbp, %rcx, 4)
```

La cosa è un problema: rcx, essendo scratch, potrebbe non essere preservato nella funzione *leggiint*. Se proviamo ad eseguire otterremo un *Bus error*: questo perché la modifica dell'rcx da parte di *leggiint* ci porta nel cosiddetto *buco della memoria* (la parte di memoria non utilizzata introdotta nelle lezioni precedenti). Abbiamo due soluzioni possibili:

- scegliere un registro *non scratch*
- invertire le prime due righe della parte incriminata (soluzione adottata dal prof per questo caso)

9.1.3.4 Passaggio di parametro struttura (pag.64)

Codice c++

```
// programma struttura, file es10a.cpp      int main() {
#include "servi.cpp"                      s st1, st2;
struct s { int n1; char c; int n2; };       st1 = leggis();
extern "C" s leggis() {                     st2 = fai(st1);
    s ss;                                scrivis(st2);
    ss.n1 = leggiint();                   return 0;
    ss.c = leggichar();                  }
    ss.n2 = leggiint();                  }
    return ss;                           }

extern "C" void scrivis(s ss) {           // programma struttura, file es10b.cpp
    scriviint(ss.n1);                  struct s { int n1; char c; int n2; };
    scrivichar(ss.c);                 extern "C" s fai(s st) {
    scriviint(ss.n2);                  s ss;
    nuovalinea();                     ss.n1 = st.n1 + 5;
}                                         ss.c = st.c + 1;
                                         ss.n2 = st.n2 + 10;
                                         return ss;
}

extern "C" s fai(s st);
```

Traduzione assembler di es10b

```
.text
.global fai
fai:
    push %rbp
    mov %rsp, %rbp
    sub $32, %rsp
    mov %rdi, -16(%rbp)
    mov %esi, -8(%rbp)

    # ss.n1 = st.n1 + 5
    mov -16(%rbp), %eax
    add $5, %eax
    mov %eax, -32(%rbp)
    # ss.c = st.c + 1
    mov -12(%rbp), %al
    add $1, %al
    mov %al, -28(%rbp)
    # ss.n2 = st.n1 + 10
    mov -8(%rbp), %eax
    add $10, %eax
    mov %eax, -24(%rbp)
    # return ss
    mov -32(%rbp), %rax
    mov -24(%rbp), %edx

    leave
    ret
```

#		+4	
# +-----+ +-----+			+
# x x x ss.c	ss.n1		-32
# +-----+ +-----+			+
# x x x x	ss.n2		-24
# +-----+ +-----+			+
# x x x st.c	st.n1		-16
# +-----+ +-----+			+
# x x x x	st.n2		-8
# +-----+-----+			+
#	vecchio rbp		<- rbp
# +-----+-----+			+
#	ind di ritorno		
# +-----+-----+			+

- Nel riservare lo spazio agli elementi della struttura ricordiamo lo stesso ragionamento visto per gli array: se io incremento devo passare all'elemento successivo. Segue che **in cima alla pila (indirizzi più bassi) ci saranno i primi elementi della struttura.**
- Memorizziamo l'rbp vecchio e riserviamo lo spazio di memoria necessario, in luce delle considerazioni precedenti

```
push %rbp
mov %rsp, %rbp
sub $32, %rsp
```

- Sistemiamo la struttura posta in ingresso come parametro. Abbiamo 16byte divisi in due blocchi, seguono due operazioni di mov. Si consideri che finchè il parametro sta in 16byte non è un problema. Segue

```
mov %rdi, -16(%rbp)
mov %esi, -8(%rbp)
```

Nel primo registro vanno i primi elementi della struttura, i rimanenti nel secondo registro.

- Per le operazioni di somma
 - spostiamo l'operando non costante dalla memoria a un registro
 - eseguiamo l'addizione su quel registro
 - spostiamo il contenuto modificato nella variabile indicata nell'assegnamento

segue

```
# ss.n1 = st.n1 + 5
mov -16(%rbp), %eax # ----- sposto st.n1 in eax
add $5, %eax # ----- sommo 5 ad eax
mov %eax, -32(%rbp) # ---- sposto il contenuto addizionato in ss.n1

# ss.c = st.c + 1
mov -12(%rbp), %al # ----- sposto st.c in al
add $1, %al # ----- sommo 1 ad al
mov %al, -28(%rbp) # ----- sposto il contenuto incrementato in ss.c

# ss.n2 = st.n1 + 10
mov -8(%rbp), %eax # ----- sposto st.n1 in eax
add $10, %eax # ----- sommo 10 ad eax
mov %eax, -24(%rbp) # ----- sposto il contenuto addizionato in ss.n2
```

- La struttura è di 16byte, divisa in due blocchi. Dobbiamo eseguire due spostamenti di memoria per la struttura restituita dalla funzione

```
mov -32(%rbp), %rax
mov -24(%rbp), %edx
```

Come parte meno significativa poniamo la prima riga della struttura (quella a indirizzo più basso, col primo intero e il char), l'altra come parte più significativa (quella col secondo intero).

- Chiudiamo come al solito

```
leave
ret
```

- **Osservazione sulla dichiarazione delle strutture.**

```
struct s { int n1; char c; int n2; };
```

Ricordiamoci che il compilatore lavora singolarmente su ogni file.

- Nel file col main abbiamo la dichiarazione della struttura *s* (la utilizziamo nel main, è fondamentale perché il compilatore deve sapere quanto spazio allocare per la struttura - non tanto per gli elementi interni)
- Nel file con la funzione *fai* si potrebbe pensare che non è necessario definire la struttura *s*: FALSO, altrimenti il compilatore non potrà conoscere gli offset della struttura (la funzione *fai* lavora su singoli elementi delle strutture)

- **Proviamo a cambiare la struttura *s* in uno dei due files: se ne accorge qualcuno?**

Il compilatore no, visto che lavora sui singoli files. Il collegatore vede tutto insieme, ma non si preoccupa di queste cose. Quindi, se siamo fortunati, non se ne accorge nessuno. Se modifico la definizione nel primo file non è un problema: sposto gli elementi ma la dimensione complessiva della struttura è la stessa. Nel secondo file, invece, la cosa genera problemi: non vengono segnalati errori, ma il risultato ottenuto non è quello desiderato.

Traduzione assembler del main

#		+4
# +-----+ +-----+		
# x x x	st2.c	ss.n1 -32
# +-----+ +-----+		
# x x x x		st2.n2 -24
# +-----+ +-----+		
# x x x	st1.c	st1.n1 -16
# +-----+ +-----+		
# x x x x		st1.n2 -8
# +-----+ +-----+		
#	vecchio rbp	<- rbp
# +-----+ +-----+		
#	ind di ritorno	
# +-----+ +-----+		

```
.text
.global main
main:
```

```
push %rbp
mov %rsp, %rbp

call leggis
mov %rax, -16(%rbp)
mov %edx, -8(%rbp) # <---- memorizzo la struttura restituita in st1

mov -16(%rbp), %rdi
mov -8(%rbp), %edx # <--- pongo la struttura st1 come argomento della funzione fai
call fai
mov %rax, -32(%rbp)
mov %edx, -24(%rbp) # <--- memorizzo la struttura restituita in st2

mov -32(%rbp), %rdi
mov -24(%rbp), %esi # <--- pongo la struttura st2 come argomento della funzione scrivis
call scrivis

xorl %eax, %eax # <--- restituisco 0
leave
ret # <---- concludo come al solito
```

Capitolo 10

Venerdì 19/03/2021

10.1 Traduzione dei nomi da C++ ad Assembler

Cosa abbiamo visto fino ad ora? Fino ad ora abbiamo affrontato questioni valide sia per il C che per il C++.

Overloading in C Fino ad ora abbiamo introdotto le funzioni in questo modo

```
extern "C" elab(...)
```

indicare *C* significa dire che la funzione rispetta lo standard del C. Nel C, ricordiamo, non è presente l'overloading e i nomi delle funzioni vengono tradotti esattamente come indicati nel codice.

Overloading in C++ L'overloading è presente in C++ e deve essere trattato. Possiamo avere funzioni con lo stesso nome, ma parametri in ingresso diversi (sia nel numero che nei tipi).

```
elab(int)
elab(int, int)
elab(miastuct)
elab(int*)
```

queste sono funzioni diverse! Il compilatore, quando vede una chiamata del tipo

```
elab(5)
```

deve capire quale funzione chiamare. Ricordiamoci che due funzioni non si distinguono per il valore restituito: segue che non posso avere funzioni del tipo

```
char elab(int)
int elab(int)
```

Se il compilatore sta compilando un file e nello stesso è presente la definizione di ciascuna di queste funzioni la cosa è semplice, mentre diventa più complessa se alcune funzioni (o tutte le funzioni) sono definite in altri files. Non posso indicare l'indirizzo della funzione, visto che non lo conosco: lascerò al collegatore le informazioni necessarie per capire quale funzione ci interessa. Il collegatore guarderà le tabelle dei simboli di tutti i files.

In base a cosa stabilisco se l'etichetta è la stessa che sto cercando? L'etichetta è una stringa, quindi mediante corrispondenza tra stringhe.

Problema Non posso definire più funzioni con la stessa etichetta *elab*. Il collegatore vedrebbe più definizioni della stessa etichetta. Il problema nasce perché vogliamo utilizzare lo stesso collegatore del C.

Soluzione Modifichiamo i nomi posti delle etichette codificando all'interno gli argomenti della funzione. Dobbiamo individuare un algoritmo che produce la stessa stringa data la stessa funzione (ricordiamoci che il compilatore agisce sui singoli files in modo distinto).

10.1.1 Regole per le etichette delle funzioni

Osservazione L'algoritmo non è standard. Esistono compilatori diversi con regole diverse.

Regole Prendiamo una funzione

```
funz(tipo1, tipo2, ..., tipoN)
```

1. La stringa inizia sempre così

`_Z`

2. Segue il nome della funzione, preceduto dal numero di caratteri del nome (devo sapere quando si inizia a parlare dei tipi nella stringa)

`4funz`

3. Segue la traduzione di ognuno dei tipi

`_Z4funzS1S2...SN`

Quali sono i possibili tipi nel C++? Il numero di tipi è piuttosto alto: abbiamo

- i tipi base (char, bool, int, long, unsigned/signed)
- i tipi definiti dall'utente (enum, class, struct...)
- i tipi composti (*,&, const, array...)

10.1.1.1 Tipi base

Per ogni tipo base c'è una lettera identificativa

<i>tipo C++</i>	<i>Sottoidentificatore Assembler</i>
void (o argomenti assenti)	v
short int	s
int	i
long int	l
unsigned short int	t
unsigned int	j
unsigned long int	m
char	c
unsigned char	h
bool	b
...	

notare che in assenza di argomenti è presente la lettera *v* (*void*). In presenza di più argomenti relativi a tipi base si pongono più caratteri in sequenza

```
funz() ---> _Z4funzx
funz(int) --->_Z4funzi
funz(long int) ---> _Z4funzl
funz(short int) ---> _Z4funzs
funz(int, int) ---> _Z4funzii
```

10.1.1.2 Struttura

Supponiamo di voler trovare la stringa relativa a questa funzione

```
funz(mastrutt)
```

Trattiamo la cosa in modo simile al nome della funzione. *mastrutt* ha nove caratteri, quindi

```
_Z4funz9mastrutt
```

Vediamo un ulteriore esempio che combina una struttura e un tipo base

```
funz(mias, int, char) ----> _Z4funz4miasic
```

10.1.1.3 Puntatori

Poniamo P seguito dal tipo

```
tipo* ----> PStipo
```

Vediamo alcuni esempi

```
int* ---> Pi
mias* ---> P4mias
int** ---> PPi
```

10.1.1.4 Riferimenti

Poniamo R seguito dal tipo

`tipo& ---> RStipo`

Vediamo alcuni esempi

`int& ---> Ri`
`mias*& ---> RP4mias`

10.1.1.5 Costanti

Sulle costanti dobbiamo fermarci un attimo

- (1) `const int*`
- (2) `int const* ---> Pki`
- (3) `int* const`

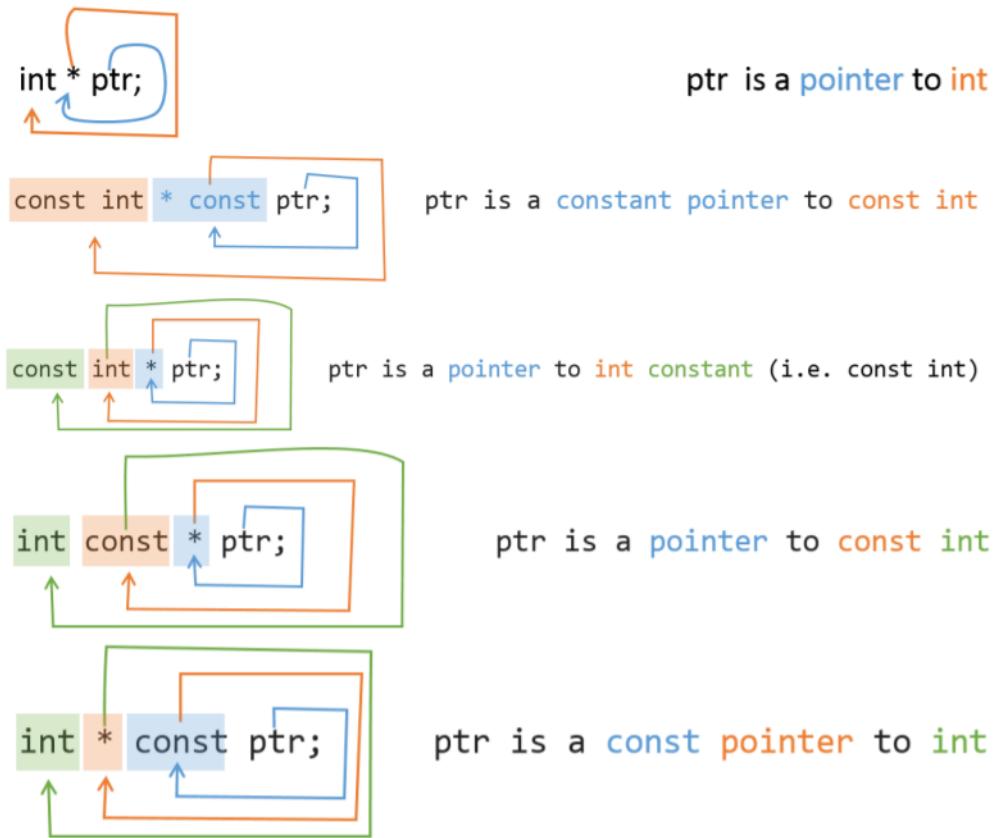
Quanti tipi diversi ho dichiarato? Due:

- Puntatore a intero costante (1 e 2, posso modificare l'indirizzo puntato ma non il valore dell'intero puntato)
- Puntatore costante a intero (3, posso modificare il valore dell'intero ma non l'indirizzo puntato)

const si traduce con una K maiuscola. Nel capire di quale puntatore stiamo parlando dobbiamo ricordarci che **il *const* fa sempre riferimento a ciò che è prima, tranne quando è in cima (in quel caso fa riferimento a ciò che viene dopo).**

Regola della spirale La regola della spirale (suggerimento mio, non di Lettieri) permette di capire in modo più facile di quale puntatore stiamo parlando:

- parto dal nome dell'elemento;
- mi sposto indietro facendo un giro in senso orario (se becco la keyword *const* devo contare fino a tre, e guardare a chi fa riferimento usando la regola di prima);
- ripeto fino al termine dell'espressione.



Confronto tra tipi e tipi costanti nei parametri Attenzione alle seguenti funzioni

```
funz(int)
funz(const int)
```

non sono due funzioni diverse. Entrambe generano come stringa

`_Z4funzi`

Nel secondo caso dico che non posso modificare la cosa che mi viene passata, ma succede la stessa cosa anche nel primo caso visto che si ha un passaggio per valore. Il ragionamento è valido anche parlando in modo generico.

Conseguenza In *KPi* la lettera K non dice sostanzialmente niente, può essere tolto.

Se nel costruire l'etichetta pongo un *K* all'inizio allora possiamo toglierlo

10.1.1.6 Array

Prendiamo i seguenti esempi

```
f(int a[10])
f(int a[100])      -----> _Z1fPi
f(int a[])
```

sono le stesse funzioni, il numero non entra a far parte del nome della funzione. Inoltre, la seguente funzione è uguale alle precedenti

```
f(int* a)           -----> _Z1fPi
```

Ricordiamo che passare un array come parametro significa passare un puntatore al primo elemento dell'array. Il fatto che un array decada in puntatore è una delle cose più criticate del C, conseguentemente anche del C++.

Osservazione: non affronteremo la regola per scrivere le etichette a riguardo, ma dobbiamo tenere a mente che funzioni di questo tipo non sono equivalenti

```
f(int a[] [10])  
f(int a[] [20])
```

possiamo mantenere implicita solo la prima dimensione, quella successiva no. Devo conoscere quanti interi sono presenti in ogni elemento dell'array, altrimenti non saprei come raggiungerli. Raggiungo l'elemento $a[i][j]$ con la seguente formula

```
i*secondaDim+j
```

10.1.1.7 Esempi di etichette con costanti

- Per ogni esempio
 - Metto il solito inizio
 - Per il nome pongo *1f*
- `f(int const*) -----> _Z1fPKi`
 - Per il tipo pongo *PKi*: abbiamo un puntatore a un intero costante
- `f(const int*) -----> _Z1fPKi`
 - Ricordiamoci la regola detta prima. L'etichetta ottenuta è la stessa della funzione precedente
- `f(int* const) -----> _Z1fKPi -----> _Z1fPi`
 - Per il tipo pongo *Pi*: abbiamo un puntatore costante a intero, abbiamo già detto che non ha senso mettere prima *K*
- `f(int const* const) -----> _Z1fKPKi -----> _Z1fPKi`
 - Per il tipo pongo *PKi*: abbiamo un puntatore costante a intero costante. La *K* prima non ha senso, quella dopo *P* invece è necessaria.
- `f(int* const*) -----> _Z1fPKPi`

– Per il tipo pongo *PKPi*: abbiamo un puntatore a puntatore costante di intero.

- `f(int*)` ----> `_Z1fPi`

– Per il tipo pongo *Pi*: abbiamo un puntatore a intero

Leggilo indietro (come guidato da [Regola in senso orario/a spirale](#)):

- `int*` - puntatore a int
- `int const *` - puntatore a const int
- `int * const` - const puntatore a int
- `int const * const` - const puntatore a const int

Ora il primo `const` può essere su entrambi i lati del tipo, quindi:

- `const int * == int const *`
- `const int * const == int const * const`

Se vuoi diventare veramente pazzo puoi fare cose del genere:

- `int **` - puntatore al puntatore su int
- `int ** const` - un puntatore const a un puntatore a un int
- `int * const **`: un puntatore a un puntatore const su un int
- `int const ***` - un puntatore a un puntatore a un const int
- `int * const * const` - un puntatore const a un puntatore const su un int

Curiosità sui puntatori. L'elemento `a[i]` viene tradotto nel seguente modo

`*(a+i)`

questa cosa è talmente vera in C, e quindi in C++, che possiamo dire

`*(i+a)`

segue che dire

`i[a]`

equivale a dire `a[i]`. La notazione è utilizzabile senza alcun problema in C e in C++.

Capitolo 11

Lunedì 22/03/2021

11.1 Corrispondenza C++-Assembler: classi

```
class c {
    int i;
    long j;
    ....
    c();
    c(int);
    ~c();
};
```

Parliamo di classi: esse permettono di raggruppare insieme dati e codice, definire un nuovo tipo (astratto) e la sua rappresentazione interna (sottotipi), definire quali operazioni è possibile eseguire su questo sottotipo. Possiamo definire *funzioni membro*, distinguere la visibilità dei vari elementi definiti in una classe, definire funzioni particolari come costruttori (anche più di uno in base ai tipi) e i distruttori.

11.1.1 Prima implementazione (da C++ a C): compilatore *cfront*

La prima implementazione è avvenuta con un compilatore chiamato *cfront* in cui si passa da C++ a C. Il linguaggio verso cui si traduce non deve essere per forza a basso livello.

Come traduciamo una chiamata di funzione membro? Prendiamo il seguente codice

```
int main() {
    c miac;
    miac.f(...);
}
```

- Il compilatore osserva la definizione della classe: questa non produce di per sé alcun codice, ma determina una struttura dati interna al compilatore. Il compilatore saprà che esiste una struttura dati che presenta certi dati e certe funzioni membro, con costruttore e distruttore.
- Quando vede dichiarata un'istanza di quell'oggetto il compilatore si preoccupa di allocare spazio per quell'istanza.

- Quando vede un'invocazione della funzione membro di un oggetto verifica che la si possa chiamare in quel punto (elemento public o no?). Solo dopo questo step traduce la chiamata.

Allocazione dell'oggetto Lo spazio allocato per l'oggetto è lo spazio delle sue variabili membro. La sua parte dati viene tradotta in uno struct:

```
struct c {
    int i;
    long j;
};
```

Le funzioni membro diventano funzioni globali, definite però all'interno di una classe. I nomi non coincideranno con quelli delle funzioni: due funzioni globali non possono avere stesso nome, significherebbe impedire a due classi diverse di assumere lo stesso nome per una funzione membro.

Argomento implicito nelle funzioni Le funzioni membro ricevono, in modo implicito, un puntatore all'oggetto su cui devono lavorare

```
void c_f(c* this, ...)
```

Dereferenziazione Sappiamo che scrivendo una cosa del genere

```
void c_f(c* this, ...) {
    i = 10;
}
```

il compilatore intenderà `i` come il campo dell'oggetto su cui la funzione membro è stata chiamata.

```
void c_f(c* this, ...) {
    this->i = 10;
}
```

Abbiamo una dereferenziazione隐式 (nulla di nuovo rispetto a quanto visto a Fondamenti di programmazione). Segue una traduzione del main di questo tipo

```
int main() {
    c miac;
    c_f(&miac, ...);
}
```

11.1.2 Implementazione attuale (da C++ ad Assembler direttamente)

Adesso *cfront* non viene più utilizzato. Abbiamo compilatori che effettuano un passaggio diretto da C++ ad Assembler. Non si hanno grandi differenze concettuali rispetto al *cfront*.

Cosa dobbiamo sapere?

- Le regole per la traduzione dei nomi delle funzioni membro.
- Come viene passato il puntatore *this*.

11.1.2.1 Traduzione dei nomi delle funzioni membro

```
class c {  
    tipoR funz(tipo1, ..., tipoN)  
}
```

Il nome della funzione viene tradotto in Assembler nello stesso modo in cui faceva *cfront*. In particolare, il nome della funzione deve contenere il nome della classe. Vediamo le regole:

- Si parte con

_Z

- Tutti i caratteri che servono a definire di quale funzione membro stiamo parlando sono posti tra due lettere: N ed E

_ZN ... E

Indichiamo il nome della classe e il nome della funzione negli stessi modi visti nella scorsa lezione

_ZN1c4funzE

Prima il nome della classe, dopo il nome della funzione

- Segue la traduzione dei tipi della funzione con i modi già visti, **SENZA** contare il parametro隐式的 *this*. Unica differenza: se uno di questi tipi si rifà alla classe associata alla funzione membro il nome della classe non viene ripetuto, ma si pone

S_

Esempio Vediamo un esempio con la seguente classe

```
class miac {  
    void funz1(); <---- _ZN4miac5funzEv  
    void funz2(int); <---- _ZN4miac5funzEi  
    void funz2(miac*) <---- _ZN4miac5funzEPS_  
}
```

La regola è un po' più complicata ma ci limiteremo a questo. Il compilatore cerca di ridurre al minimo le ridondanze, per evitare problemi relativi alla lunghezza delle stringhe nel collegatore.

Costruttori I costruttori si indicano col seguente nome

C1

La cosa non è fonte di equivoci: tutti i nomi definiti da un utente iniziano sempre con un numero.

Distruttori I distruttori si indicano in modo simile

D1

11.2 Esercizio dalle diapositive di Frosini

11.2.1 Primo esercizio

Codice C++

```
// programma classe1, file clai1.cpp

#include "servi.cpp"
class clai {
    int ix, iy, iz;
public:
    clai();
    clai(int, int, int);
    // ...
    void stampa();
    clai somma(int, int, int);
};

clai::clai() {
    ix = 0; iy = 0; iz = 0;
}

clai::clai(int a, int b, int c) {
    ix = a; iy = b; iz = c;
}

void clai::stampa() {
    scriviint(ix);
    scriviint(iy);
    scriviint(iz);
    nuovalinea();
}

clai clai::somma(int a, int b, int c) {
    clai temp;
    temp.ix = ix+a;
    temp.iy = iy+b;
    temp.iz = iz+c;
    return temp;
}
```

Traduzione di *stampa*

```
.global _ZN4clai6stampaEv
_ZN4clai6stampaEv:
    push %rbp
    mov %rsp, %rbp
    sub $16, %rip
    mov %rdi, -8(%rbp)

    mov -8(%rbp), %rax # this
    mov (%rax), %edi # this->ix
    call scriviint
```

```

mov -8(%rbp), %rax # this
mov 4(%rax), %edi # this->iy
call scriviint

mov -8(%rbp), %rax # this
mov 8(%rax), %edi # this->iz
call scriviint

call nuovalinea # Non ho argomenti in ingresso

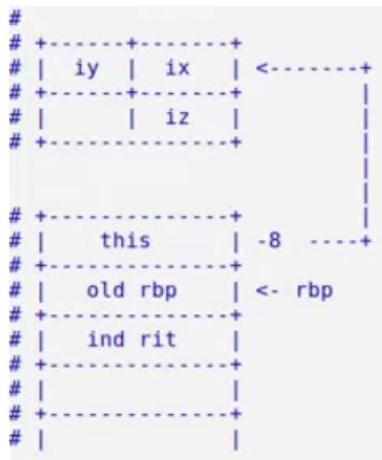
leave
ret

```

- *stampa* è una funzione globale che ha in ingresso un solo parametro, quello implicito *this*.
- Scriviamo il nome della funzione

`_ZN4clai6stampaEv`

- Inizio con i soliti caratteri: `_Z`.
- Abbiamo una funzione membro, dunque poniamo nome della classe e nome della funzione membro tra le lettere *N* ed *E*.
- Concludo con la lettera *v* (la funzione non presenta parametri in ingresso, *this* non si considera)
- Per quanto riguarda il record di attivazione nulla di strano (l'unico elemento da considerare è il parametro *this*)



solito prologo

```

push %rbp
mov %rsp, %rbp
sub $16, %rip
mov %rdi, -8(%rbp)

```

- La funzione si limita semplicemente a chiamare quattro funzioni esterne alla classe. I valori posti come parametri di queste funzioni sono sempre dati membro dell'istanza della classe. Segue che dovrò usare il puntatore *this* per recuperare questi dati, mediante indirizzamento indiretto.

```

mov -8(%rbp), %rax # this
mov (%rax), %edi # this->ix
call scriviint

mov -8(%rbp), %rax # this
mov 4(%rax), %edi # this->iy
call scriviint

mov -8(%rbp), %rax # this
mov 8(%rax), %edi # this->iz
call scriviint

call nuovalinea # Non ho argomenti in ingresso

```

- Concludiamo allo stesso modo con *leave* e *ret*.

Traduzione dei costruttori I costruttori hanno una sintassi particolare, non restituiscono niente all'apparenza (lo standard suggerisce che siano void). Si adotta una convenzione: la restituzione dell'indirizzo dell'oggetto costruito.

<pre>.global _ZN4claiC1Eiii _ZN4claiC1Eiii: push %rbp mov %rsp, %rbp sub \$32, %rip mov %rdi, -8(%rbp) # mov %esi, -24(%rbp) # mov %edx, -28(%rbp) # mov %ecx, -16(%rbp) mov %esi, (%rdi) mov %edx, 4(%rdi) mov %ecx, 8(%rdi) mov %rdi, %rax leave ret </pre>	<pre>.global _ZN4claiC1Ev _ZN4claiC1Ev: push %rbp mov %rsp, %rbp sub \$16, %rip mov %rdi, -8(%rbp) movl \$0, (%rdi) # this->ix = 0 movl \$0, 4(%rdi) # this->iy = 0 movl \$0, 8(%rdi) # this->iz = 0 mov %rdi, %rax leave ret </pre>
--	--

- Primo costruttore (quello senza argomenti):

- Scriviamo il nome della funzione
- Solito prologo per il record di attivazione (stessa struttura vista per la funzione *stampa*)
- Poniamo tutte le variabili della classe uguali a zero, usando un indirizzamento indiretto con displacement

```
movl $0, (%rdi) # this->ix = 0  
movl $0, 4(%rdi) # this->iy = 0  
movl $0, 8(%rdi) # this->iz = 0
```

- Restituiamo l'indirizzo dell'oggetto costituito

```
mov %rdi, %rax
```

- Concludiamo allo stesso modo

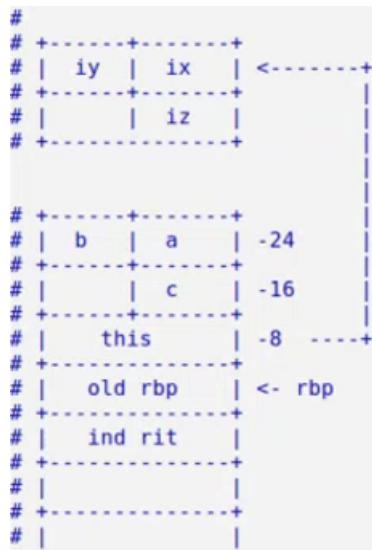
leave
ret

- Secondo costruttore (quello con argomenti):

- Scriviamo il nome della funzione

_ZN4claiC1Eiii

- Il record di attivazione conterrà al suo interno anche i parametri in ingresso a , b , c .



scriviamo il prologo

```
push %rbp  
mov %rsp, %rbp  
sub $32, %rip  
  
mov %rdi, -8(%rbp)  
mov %esi, -24(%rbp)  
mov %edx, -28(%rbp)  
mov %ecx, -16(%rbp)
```

in aggiunta ad rdi spostiamo anche il contenuto dei parametri in ingresso.

- Aggiorniamo i valori delle variabili. Facciamo la cosa velocemente, passando direttamente i valori dai registri

```
mov %esi, (%rdi)  
mov %edx, 4(%rdi)  
mov %ecx, 8(%rdi)
```

chiaramente il compilatore, se chiediamo ottimizzazione, farà così e non allocherà quanto allocato prima nel record di attivazione.

- Restituiamo l'indirizzo dell'oggetto costituito

```
    mov %rdi, %rax
```

- Concludiamo allo stesso modo

```
    leave  
    ret
```

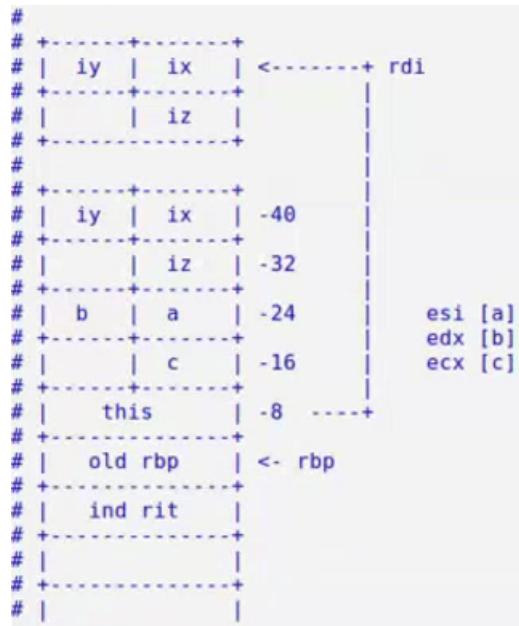
Traduzione della funzione *somma* La funzione somma presenta al suo interno un'istanza temporanea della classe, e la restituisce!

```
.global _ZN4clai5sommaEiii  
_ZN4clai5sommaEiii:  
    push %rbp  
    mov %rsp, %rbp  
    sub $48, %rsp  
  
    mov %rdi, -8(%rbp)  
    mov %esi, -24(%rbp)  
    mov %edx, -28(%rbp)  
    mov %ecx, -16(%rbp)  
  
    lea -40(%rbp), %rdi  
    call _ZN4claiC1Ev  
  
    mov -8(%rbp), %rdi # sposto l'indirizzo dell'oggetto temp in rdi  
  
    mov -24(%rbp), %eax  
    add (%rdi), %eax  
    mov %eax, -40(%rbp)  
  
    mov -20(%rbp), %eax  
    add 4(%rdi), %eax  
    mov %eax, -36(%rbp)  
  
    mov -16(%rbp), %eax  
    add 8(%rdi), %eax  
    mov %eax, -32(%rbp)  
  
    mov -40(%rbp), %rax  
    mov -32(%rbp), %edx  
  
    leave  
    ret
```

- Scriviamo il nome della funzione

```
_ZN4clai5sommaEiii
```

- Il record di attivazione includerà le variabili relative all'istanza *temp*



Abbiamo incluso anche i parametri, nonostante non sia cosa necessaria. Poniamo in memoria i parametri passati mediante registri

```
mov %rdi, -8(%rbp)
mov %esi, -24(%rbp)
mov %edx, -28(%rbp)
mov %ecx, -16(%rbp)
```

- La creazione di un'istanza *temp* non si traduce in una semplice allocazione di spazio: dobbiamo creare l'oggetto, cioè chiamare il costruttore!

```
lea -40(%rbp), %rdi
call _ZN4claiC1Ev
```

il compilatore ottiene l'etichetta partendo dalla dichiarazione (non dalla definizione).

- Eseguiamo le addizioni

```
mov -8(%rbp), %rdi # sposto l'indirizzo dell'oggetto temp in rdi

mov -24(%rbp), %eax
add (%rdi), %eax
mov %eax, -40(%rbp)

mov -20(%rbp), %eax
add 4(%rdi), %eax
mov %eax, -36(%rbp)

mov -16(%rbp), %eax
add 8(%rdi), %eax
mov %eax, -32(%rbp)
```

- Dobbiamo restituire l'oggetto, ma non possiamo limitarci a passare a un indirizzo. In questo momento l'istanza *temp* è allocata nel record di attivazione, dobbiamo renderla disponibile. Quello che dobbiamo fare in teoria è chiamare il *costruttore di copia*. In questo caso non lo abbiamo,

quindi ci limitiamo a restituire l'oggetto temporaneo copiandolo nei registri (è un caso particolare, si traduce il tutto nel ritorno di una struttura)

```
mov -40(%rbp), %rax
mov -32(%rbp), %edx
```

- Dopo aver fatto ciò dovrei chiamare il distruttore di *clai* per l'istanza *temp*. Non esiste un distruttore, quindi non abbiamo da fare altro.
- Concludiamo allo stesso modo

```
leave
ret
```

11.2.2 Secondo esercizio

Codice C++

```
// programma classe2, file main2.cpp

class clai {
    int ix, iy, iz;
public:
    clai();
    clai(int, int, int);
    clai somma(int, int, int);
    void stampa();
};

int main() {
    clai c1, c2(1, 2, 3), c3;
    c1 = c2.somma(5, 10, 15);
}
```

Traduzione del main

```
.global main
main:
    push %rbp
    mov %rsp, %rbp
    sub $48, %rsp

    lea -12(%rbp), %rdi
    call _ZN4claiC1Ev

    lea -24(%rbp), %rdi
    mov $1, %esi
    mov $2, %edx
    mov $3, %ecx
    call _ZN4claiC1Eiii

    lea -36(%rbp), %rdi
    call _ZN4claiC1Ev

    lea -24(%rbp), %rdi
    mov $5, %esi
    mov $10, %edx
    mov $15, %ecx
    call _ZN4clai5sommaEiii

    mov %rax, -12(%rbp)
    mov %edx, -4(%rbp)

    leave
    ret
```

- La parte relativa alla classe non genera codice Assembler: ciò avviene col main, con le definizioni di istanze e le chiamate di funzioni.
- Nella struttura del record di attivazione giochiamo a *Tetris*

```

# +-----+-----+
# | c3.ix|      | -40
# +-----+-----+
# | c3.iz| c3.iy | -32
# +-----+-----+
# | c2.iy| c2.ix | -24
# +-----+-----+
# | c1.ix| c2.iz | -16
# +-----+-----+
# | c1.iz| c1.iy | -8
# +-----+-----+
# | old rbp   | <- rbp
# +-----+-----+
# | ind rit   |
# +-----+-----+

```

solito prologo dove riserviamo una parte della pila

```

push %rbp
mov %rsp, %rbp
sub $48, %rsp

```

- Inizializziamo le istanze *c1*, *c2* e *c3* della classe *clai*

```

lea -12(%rbp), %rdi # pongo l'argomento implicito this
call _ZN4claiC1Ev # Chiamo il costruttore default

lea -24(%rbp), %rdi # Argomento implicito this
mov $1, %esi # Primo argomento esplicito
mov $2, %edx # Secondo argomento esplicito
mov $3, %ecx # Terzo argomento esplicito
call _ZN4claiC1Eiii # Chiamo il costruttore con tre argomenti int

lea -36(%rbp), %rdi # pongo l'argomento implicito this
call _ZN4claiC1Ev # Chiamo il costruttore default

```

- Traduciamo l'operazione di assegnamento

```

lea -24(%rbp), %rdi # Argomento implicito this
mov $5, %esi # Primo argomento esplicito
mov $10, %edx # Secondo argomento esplicito
mov $15, %ecx # Terzo argomento esplicito
call _ZN4clai5sommaEiii # Chiamata della funzione c2.somma(int, int, int)

# assegnamento del contenuto restituito dalla funzione membro c2.somma a c1
mov %rax, -12(%rbp)
mov %edx, -4(%rbp)

```

Ricordiamo che abbiamo un caso particolare che ci permette di restituire l'istanza *temp* come una struttura.

- Concludiamo allo stesso modo

```

leave
ret

```

Capitolo 12

Martedì 23/03/2021

12.1 Punti su cui stare attenti nella prova pratica

Negli esercizi di traduzione della prova d'esame ci sono due punti ostici:

1. uso della MOV/LEA (quando uso una cosa e quando un'altra);
2. capire più nel dettaglio cosa succede quando una funzione restituisce valori di tipo struttura/classe/unione.

Il primo punto lo abbiamo già affrontato, vediamo il secondo.

12.1.1 Funzioni che restituiscono strutture/classi per valore

Esempio 1 Supponiamo di voler fare la seguente somma (dove a e b sono interi)

$$a + b$$

sappiamo che in C++ ogni espressione ha un *tipo* e un *valore*. In questo caso il tipo è *intero* e il valore è il risultato della somma. Questo valore di tipo intero va messo da qualche parte, l'espressione non ci dice cosa ne dobbiamo fare. Se invece indico

$$c = a + b$$

sappiamo che il risultato dovrà essere messo nella variabile c . Precisamente:

1. memorizzo da qualche parte l'unico risultato intermedio, $a + b$;
2. copio il risultato intermedio dal luogo in cui l'abbiamo salvato all'indirizzo di c .

Esempio 2 Consideriamo la seguente espressione...

$$e = (a + b) \times (c + d)$$

che non può essere calcolata in un colpo solo. Abbiamo tre valori temporanei:

- $a + b$
- $c + d$
- $(a + b) \times (c + d)$

Potremo sbarazzarci di questi valori dopo il loro utilizzo.

Se avessi oggetti? Supponiamo che a e b non siano interi, ma istanze di una presunta classe *matrice*. Operator+ è una funzione che ha in ingresso due istanze della classe *matrice* e ne restituisce una terza.

$$C = A + B$$

Tutti i ragionamenti fatti prima rimangono validi, soprattutto per quanto riguarda i valori intermedi: ho istanze intermedie che devono essere collocate da qualche parte.

Quali sono le situazioni che possono manifestarsi con questi risultati intermedi?

- Il risultato intermedio ha dimensione inferiore a 16byte: bastano semplicemente i registri.
- Il risultato intermedio è più grande di 16byte: dobbiamo creare un vero e proprio *oggetto temporaneo* e allocarlo in memoria. L'oggetto in questione è creato ed eliminato automaticamente dal compilatore, senza esplicita indicazione del programmatore

12.1.2 Esercizio dalle diapositive di Frosini con struttura superiore a 16byte

Codice C++

```
// file es12a.cpp
#include "servi.cpp"
struct s { int n1; int n2; char a[10]; }; // la struttura non viene letta,
                                            # ma restituita da fstruct()
};

extern "C" s fstruct(int a, char c);
extern "C" void scriviris(s& ss) {
    int i;
    scriviint(ss.n1); scriviint(ss.n2);
    for (i=0; i<10; i++)
        scrivichar(ss.a[i]);
    nuovalinea();
}

int main() {
    s sa;
                                            // file es12b.cpp
                                            struct s { int n1; int n2; char a[10]; };
                                            extern "C" s fstruct(int a, char c) {
                                                int i; s st;
                                                st.n1 = a; st.n2 = 2*a;
                                                for (i=0; i<10; i++)
                                                    st.a[i] = c+i;

                                                return st;
                                            }
}
```

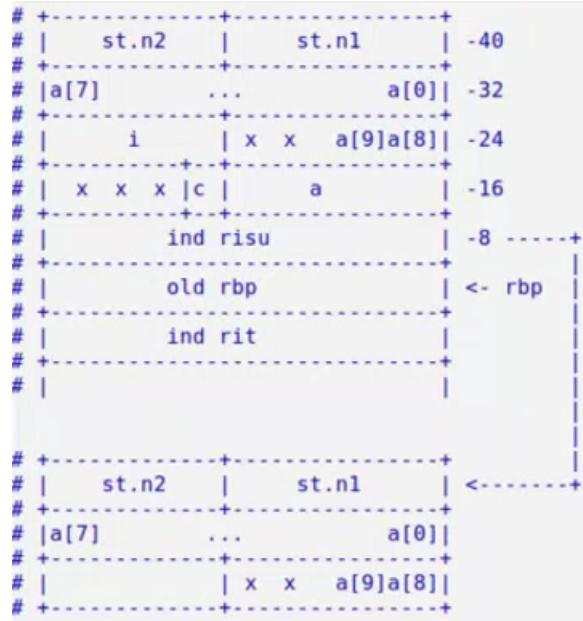
Traduzione della funzione *fstruct*

```
.global _Z7fstructic
_Z7fstructic:
    push %rbp
    mov %rsp, %rbp
    sub $48, %rsp
    mov %rdi, -8(%rbp)
    mov %esi, -16(%rbp)
    mov %dl, -12(%rbp)
    # ...
    lea -40(%rbp), %rsi
    mov -8(%rbp), %rdi
    mov $5, %rcx
    rep movsl
    mov -8(%rbp), %rax
    leave
    ret
```

- Scriviamo il nome della funzione

`_Z7fstructic`

- Dobbiamo restituire un valore più grande di 16byte, quindi siamo costretti a introdurre un nuovo parametro implicito. Questo parametro, come l'altro già visto (*this*), sarà memorizzato nel primo registro disponibile. Seguono i parametri di ingresso e le variabili locali. Vediamo il record di attivazione



solito prologo

```

push %rbp
mov %rsp, %rbp
sub $48, %rsp

```

- Copiamo i parametri

```

mov %rdi, -8(%rbp) # indirizzo che dicevamo
mov %esi, -16(%rbp)
mov %dl, -12(%rbp)

```

- Andiamo direttamente alla *return*, questione bollente. Il valore da restituire sono le ultime due righe e mezzo in cima alla pila, più di 16byte. La soluzione migliore è utilizzare le istruzioni per le stringhe viste a Reti logiche (ricordiamoci che fare una cosa con una singola istruzione invece che con più istruzioni è sinonimo di efficienza).

```

lea -40(%rbp), %rsi
mov -8(%rbp), %rdi
mov $5, %rcx
rep movsl

```

- Poniamo nel registro *rsi* la source (l'indirizzo dell'area di memoria relativa ad *st*)
- Poniamo nel registro *rdi* la destination (l'indirizzo dell'area di memoria dove porre il valore restituito, in questo caso l'indirizzo è il contenuto di un'area di memoria).

- Chiamiamo la *movsl* accompagnandola col prefisso di replicazione *rep*. Quest'ultima decrema *rcx*, posto uguale a 5, e garantisce l'esecuzione ripetuta della *movsl* finchè *rcx* non sarà uguale a zero. La *movsl*, ad ogni iterazione, incrementa l'indirizzo posto in *rdi* e in *rsi*.

- **Convenzione:** si restituisce mediante *rax* l'indirizzo di ritorno (quello che abbiamo ricevuto in ingresso)

```
mov -8(%rbp), %rax
```

L'indirizzo non può essere preso da *rdi*, visto che è stato sporcatto dalla *movsl*.

- Concludiamo allo stesso modo

```
leave  
ret
```

Traduzione della funzione *main* Svestiamo i panni del chiamato e indossiamo quelli del chiamante

```
.global main  
main:  
    push %rbp  
    mov %rsp, %rbp  
    sub $48, %rsp  
  
    # fstruct(5, 'a') <--- temp  
    lea -48(%rbp), %rdi  
    mov $5, %esi  
    mov '$a', %dl  
    call _Z7fstructic  
  
    # sa = temp  
    lea -48(%rbp), %rsi  
    lea -24(%rbp), %rdi  
    mov $5, %rcx  
    rep movsl  
  
    # ...  
    xorl %eax, %eax  
    leave  
    ret
```

# +-----+-----+	
# tmp.n2 tmp.n1 -48	
# +-----+-----+	
# a[7] ... a[0] -40	
# +-----+-----+	
# x x a[9]a[8] -32	
# +-----+-----+	
# sa.n2 sa.n1 -24	
# +-----+-----+	
# a[7] ... a[0] -16	
# +-----+-----+	
# x x a[9]a[8] -8	
# +-----+-----+	
# old rbp <- rbp	
# +-----+-----+	
# ind rit	
# +-----+-----+	

- Per quanto riguarda il record di attivazione dobbiamo considerare lo spazio non solo per l'istanza *sa* ma anche per l'istanza *temp* di cui abbiamo detto a inizio lezione.
- Solito prologo

```
push %rbp  
mov %rsp, %rbp  
sub $48, %rsp
```

- Scriviamo l'assegnamento

```
sa = fstruct(5, 'a')
```

La cosa si articola in due parti:

- la creazione dell'istanza *temp*

```
# fstruct(5, 'a') <--- temp
lea -48(%rbp), %rdi
mov $5, %esi
mov $'a', %dl
call _Z7fstructic
```

chiama la funzione e ottengo *temp*. Si ricordi che quanto restituito ha dimensione superiore a 16byte, quindi dobbiamo agire in modo diverso rispetto al solito.

- l'assegnamento di *sa*

```
# sa = temp
lea -48(%rbp), %rsi
lea -24(%rbp), %rdi
mov $5, %rcx
rep movsl
```

pongo come *sa* il contenuto restituito dalla funzione (cioè *temp*). Gestisco l'assegnamento utilizzando la funzione per stringhe *movsl*, con prefisso di replicazione.

- Andiamo direttamente al *return*. Concludiamo allo stesso modo, utilizzando lo *xor* per restituire zero.

```
xorl %eax, %eax
leave
ret
```

12.1.3 Situazioni su cui riflettere

12.1.3.1 Singolo oggetto

```
C c, x;
// ...
x = c
```

In questo caso non abbiamo bisogno di oggetti temporanei. L'oggetto *c* funge da valore dell'espressione. Non invocheremo il costruttore di copia, se presente, ma l'operatore di assegnamento.

12.1.3.2 Chiamata di funzione

```
C g();
```

```
void f() {
    C x;
    // ...
    x = g();
}
```

Supponiamo di avere una funzione g che restituisce un oggetto di classe C per valore. In questo caso è necessario allocare un oggetto temporaneo, per contenere il valore dell'espressione restituita dalla funzione. L'oggetto che andiamo a restituire, come tutti gli oggetti, deve essere

- allocato (svolta dalla funzione f , che alloca l'oggetto come se fosse una variabile locale), e
- inizializzato (svolta dalla funzione g , che restituisce il valore).

Abbiamo visto nell'esercizio di Frosini che in caso di oggetti di dimensione superiore a 16byte è necessario introdurre un parametro implicito (come primo parametro, nel registro *rdi*): un puntatore all'oggetto risultato, cioè dove vogliamo porre l'oggetto restituito.

Come traduciamo l'istruzione *return* sicuramente presente in g ?

1. Per prima cosa dobbiamo calcolare il valore dell'espressione contenuta nel *return*, ottenendo un altro oggetto di tipo C .
2. La funzione g dovrà invocare il costruttore di copia, se presente, sull'oggetto risultato allocato da f . Si passa in ingresso il riferimento all'oggetto creato allo step precedente.
3. Nel caso in cui il costruttore di copia non sia definito la funzione g dovrà copiare, membro a membro, y nell'oggetto creato da f (con le istruzioni per copiare le stringhe viste a Reti logiche).
4. Si ricordi che la funzione dovrà lasciare l'indirizzo dell'oggetto risultato in *rax*.

Al ritorno da g la funzione f dovrà completare l'assegnamento tra oggetto risultato e x , e poi distruggere l'oggetto risultato. In questo caso va invocato il distruttore, se presente.

Tempo di vita degli oggetti temporanei Il tempo di vita degli oggetti temporanei *si estende fino al punto e virgola che si trova alla fine dell'istruzione*.

12.1.3.3 Costruttore

Supponiamo di avere un costruttore con parametro *int*, relativo alla classe C . Supponiamo di avere x , oggetto di tipo C

```
x = C(100);
```

un assegnamento di questo tipo comporta la creazione di un oggetto temporaneo, e l'assegnamento ad x . Prendiamo un altro esempio

```
return C(100);
```

Anche in questo caso viene creato un oggetto temporaneo. Lo useremo per inizializzare, mediante costruttore di copia, l'oggetto temporaneo allocato dal chiamante della funzione. Il costruttore di copia viene chiamato dalla funzione chiamata relativamente all'oggetto temporaneo allocato dalla funzione chiamante.

12.1.3.4 Costruttore di copia

Cosa succede in presenza di un costruttore di copia? Negli esercizi dobbiamo stare attenti alla presenza o meno del costruttore di copia. Viene invocato in situazioni dove si inizializza un oggetto di tipo *C* a partire da un altro oggetto dello stesso tipo. Se il costruttore di copia è presente non possiamo fare come visto nell'esercizio di Frosini: dobbiamo demandare la copia al costruttore di copia. Il costruttore di copia va invocato nei seguenti casi...

1. **Assegnamento in caso di dichiarazione dell'oggetto** (costruzione di una nuova istanza a partire da un oggetto già esistente).

```
C c = e;  
C c(e);
```

Altri assegnamenti comportano l'uso dell'operatore di assegnamento, non del costruttore di copia.

2. **Passaggio di parametri per valore** (cioè poniamo in ingresso, in una funzione, un'istanza della classe)

```
f(e) <---- parametro di tipo C
```

3. **Restituzione di valore.** Cosa copiamo? L'espressione che si trova nel return viene copiata nell'oggetto temp.

```
return e;
```

Questo significa che nella *fstruct*, in presenza di un costruttore di copia, sarebbe stato necessario chiamarlo, e demandare a lui l'esecuzione della *movsl*. L'allocazione di memoria la fa il chiamante, l'inizializzazione il chiamato, partendo dall'espressione calcolata.

12.1.3.5 Elisione dei costruttori di copia (ottimizzazione, RVO ed NRVO)

Se rispettiamo queste regole otterremo un gran numero di oggetti temporanei e di invocazioni dei costruttori di copia e dei distruttori. Vediamo il seguente caso:

```
C g() {  
    return C(100);  
}
```

```
void f() {  
    C x = g();  
}
```

- Tra le variabili locali della funzione *f* poniamo *x*, istanza di *C*, e un oggetto temporaneo. Questo mi servirà per valutare l'espressione di *g*
- Passo a *g* il solito puntatore隐式 (l'indirizzo puntato è quello dell'oggetto temporaneo)

- Nella funzione g dobbiamo valutare l'espressione nel *return*. Per fare ciò dobbiamo allocare spazio in g per un secondo oggetto temporaneo. Su questo oggetto invochiamo il costruttore avente come argomento il parametro intero.
- La funzione g invoca il costruttore di copia sul primo oggetto temporaneo creato (con riferimento al secondo oggetto) e distrugge il secondo oggetto.
- La funzione f invoca il costruttore di copia su x con riferimento al primo oggetto temporaneo, e infine lo distrugge.

Attenzione allo standard Lo standard prevede che la chiamata dei costruttori di copia possa essere eliminata facendo in modo che si utilizzi, al posto di oggetti temporanei, direttamente le destinazioni.

Ottimizzazione strana L'ottimizzazione dovrebbe rendere il programma più veloce, ma qui si hanno dei *side effects*: pensiamo alle variabili statiche, si ha output diverso in base alla presenza o meno delle ottimizzazioni. Lo standard permette queste ottimizzazioni perché sono troppo importanti: segue che il programmatore deve esserne consapevole.

Ottimizzazioni svolte. Lo standard C++ 17 ha reso obbligatorie tutte le ottimizzazioni che hanno a che fare col costruttore di copia

- `C c2 = g();`
non copio in $c2$ un oggetto temporaneo, ma costruisco direttamente $c2$.
- *Return Value Optimization* (RVO):

```
return C(100);
```

non creo un oggetto temporaneo.

Ulteriore ottimizzazione, non obbligatoria, è la *Named Return Value Optimization* (NRVO):

```
C g() {
    C c(100);
    c.do_something();
    return c;
}
```

questo tipo di ottimizzazione è sofisticata (contrariamente alla RVO non guardo solo il return, ma tutta la funzione). Non alloco c , ma utilizzo direttamente l'oggetto temporaneo allocato dal chiamante (indicato in ingresso come parametro implicito, in rdi).

Traduzione della *fstruct* con la *Named Return Value Optimization*

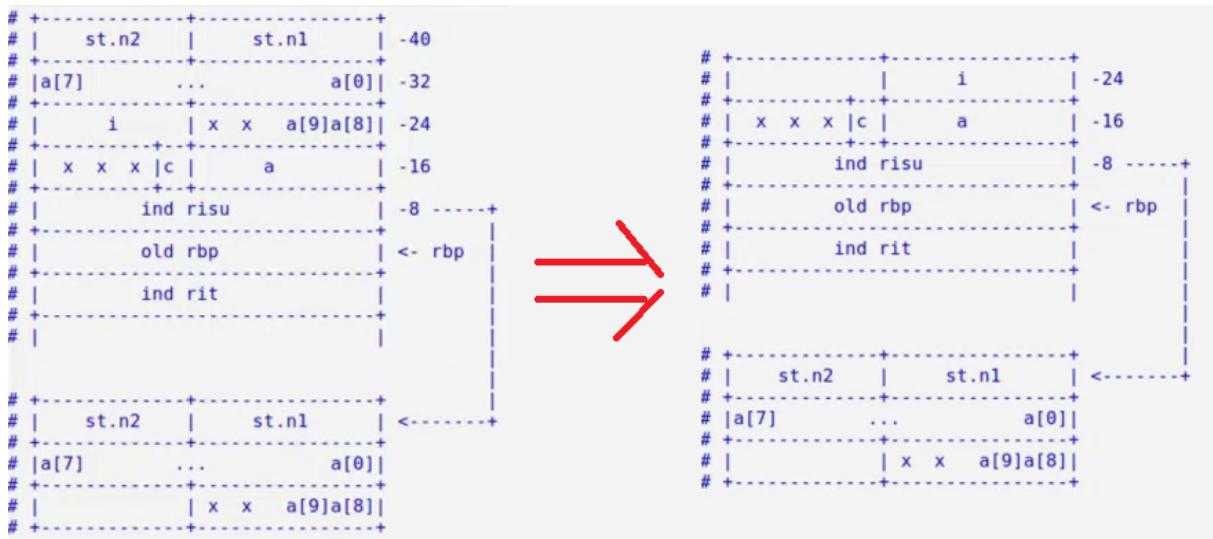
```
.global _Z7fstructic
_Z7fstructic:
push %rbp
mov %rsp, %rbp
sub $32, %rsp
mov %rdi, -8(%rbp)
mov %esi, -16(%rbp)
mov %dl, -12(%rbp)

# mov st.n1 = a ottimizzato
mov %esi, (%rdi)
```

```
# mov st.n 2= 2*a
mov %esi, %eax
sal $1, %eax
mov %eax, 4(%rdi)

# for ...
# return st
mov -8(%rsp), %rax
leave
ret
```

- Il record di attivazione è di dimensione minore, non si alloca spazio per la variabile temporanea



- Copiamo i parametri

```
mov %rdi, -8(%rbp) # indirizzo che dicevamo
mov %esi, -16(%rbp)
mov %dl, -12(%rbp)
```

- Al posto di allocare *st* utilizziamo direttamente l'oggetto in cui poi *st* andrà copiato.

```
# mov st.n1 = a ottimizzato
mov %esi, (%rdi)
# mov st.n2 = 2*a
mov %esi, %eax
sal $1, %eax
mov %eax, 4(%rdi)
```

- Concludiamo restituendo, secondo convenzione, l'indirizzo dell'oggetto allocato dal chiamante

```
mov -8(%rsp), %rax
leave
ret
```

Capitolo 13

Giovedì 28/03/2021 e Venerdì 26/03/2021

13.1 Comandi per la lettura di esami passati

Decompressione dei files	Apertura di un file pdf	Lettura di un file
tar xf percorso/nome_file	evince percorso/nome_file	cat percorso/nome_file

13.2 Esercizio di traduzione [Scritto del 26/02/2020]

Codice C++ da utilizzare

```
struct st1 { char vi[4]; };
struct st2 { char vd[4]; };
class cl {
    char v1[4]; int v3[4]; long v2[4];
public:
    cl(st1& ss);
    cl elab1(char ar1[], st2 s2);

    void stampa() {
        for (int i = 0; i < 4; i++) cout << (int)v1[i] << ' '; cout << endl;
        for (int i = 0; i < 4; i++) cout << (int)v2[i] << ' '; cout << endl;
        for (int i = 0; i < 4; i++) cout << (int)v3[i] << ' '; cout << endl << endl;
    }
};
```

Codice C++ da tradurre

```
cl::cl(st1& ss) {
    for (int i = 0; i < 4; i++) {
        v1[i] = ss.vi[i];
        v2[i] = v1[i] / 2;
        v3[i] = 2 * v1[i];
```

```

    }
}

cl cl::elab1(char ar1[], st2 s2) {
    st1 s1;
    for (int i = 0; i < 4; i++) s1.vi[i] = ar1[i] + i;

    cl cla(s1);
    for (int i = 0; i < 4; i++) cla.v3[i] = s2.vd[i];

    return cla;
}

```

Traduzione del costruttore

```

.global _ZN2clC1ER3st1
_ZN2clC1ER3st1:
    push %rbp
    mov %rsp, %rbp
    sub $32, %rsp
    mov %rdi, -8(%rbp)
    mov %rsi, -16(%rbp)

    # inizializzazione
    movl $0, -24(%rbp)
.Lfor: #confronto
    cmpl $4, -24(%rbp)
    jl .Lcorpo
    jmp .Lfinefor
.Lcorpo:
    # v1[i] = ss.vi[i]
    movslq -24(%rbp), %rcx
    mov -16(%rbp), %rdi
    mov (%rdi, $rcx), %al
    mov -8(%rbp), %rsi
    mov %al, (%rsi, %rcx)

    # v2[i] = v1[i] / 2
    mov (%rsi, %rcx), %al
    sar $1, %al
    movsbq %al, %rax
    mov %rax, 24(%rsi, %rcx, 8)

    # v3[i] = 2*v1[i]
    mov (%rsi, %rcx), %al
    shl $1, %al
    movsbl %al, %eax
    mov %eax, 4(%rsi, %rcx, 4)

    # incremento
    inc -24(%rbp)

```

```

        jmp .Lfor
.Lfinefor
    mov -8(%rbp), %rax
    leave
    ret

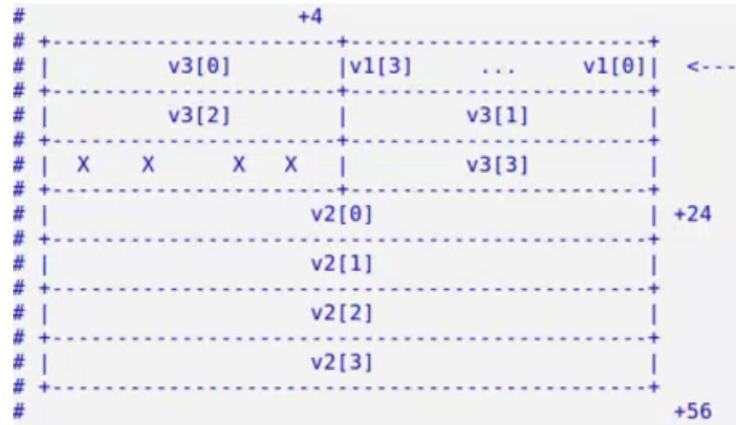
```

- Scriviamo il nome della funzione

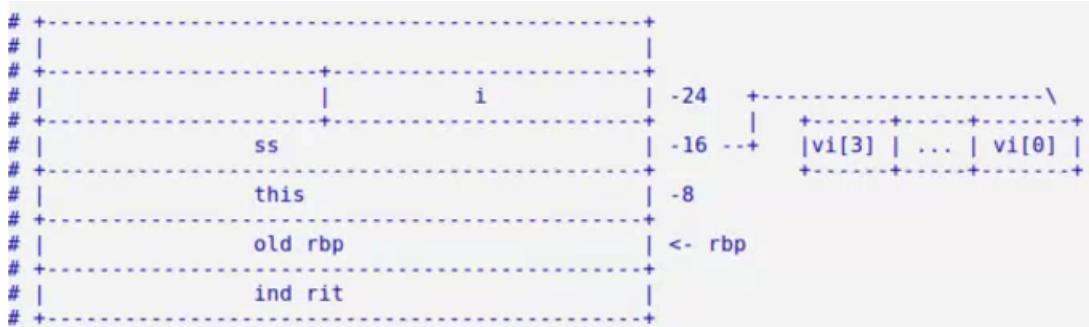
`_ZN2c1C1ER3st1`

consideriamo che si ha una funzione membro, il suo nome, che è un costruttore, che l'unico parametro esplicito presente è un riferimento alla struttura `st1`.

- Si consideri la struttura di un qualunque spazio di memoria riservato a un'istanza della classe (considerando, come al solito, `sizeof` e `alignof` dei vari elementi).



Si consideri la struttura del record di attivazione



Presente un puntatore `this` (la funzione che stiamo traducendo è una funzione membro) e un puntatore riguardo il riferimento `ss` (ricordarsi che i riferimenti vengono implementati come puntatori). Seguono i parametri, e dopo le variabili locali (si ricordi che non abbiamo un ordine ben preciso da rispettare, il programma può fare quello che vuole nello spazio che si è riservato nella pila).

- Solito prologo

```

push %rbp
mov %rsp, %rbp
sub $32, %rsp

```

inizializziamo anche l'unica variabile locale presente ($i = 0$)

```
movl $0, -24(%rbp)
```

- Dobbiamo tradurre un *for* in Assembler. Il codice si articola in
 - Inizializzazione, eseguita una volta soltanto
 - Confronto (etichetta .Lfor), La prima jump, condizionata, mi manda al corpo se la condizione è vera. La seconda jump viene raggiunta se la condizione non è vera, e ci permette di saltare a dopo il for (etichetta .Lfinefor).
 - Corpo del for (etichetta .Lcorpo), che eseguo ogni volta che la condizione risulta vera. All'interno incremento la variabile contatore *i*.
- v1[i] = ss.vi[i];
 - Attenzione alla variabile contatore *i*: è un intero. Le regole di indirizzamento ci pongono il passaggio a un registro a 64bit. Utilizziamo l'istruzione movslq per estendere con segno l'intero nel passaggio al registro

```
movslq -24(%rbp), %rcx
```
 - Subito dopo spostiamo l'indirizzo in *ss* in un registro, sempre per questioni di indirizzamento

```
mov -16(%rbp), %rdi
```
 - A questo punto possiamo spostare nel registro al (al visto che stiamo lavorando con caratteri)

```
mov (%rdi, $rcx), %al
```

abbiamo recuperato *ss.vi[i]*.
 - Abbiamo trovato *ss.vi[i]*, adesso possiamo svolgere l'operazione di assegnamento. Stiamo parlando di un campo della classe. Spostiamo *this* in un registro

```
mov -8(%rbp), %rsi
```

poniamo, infine, il contenuto del registro al nel vettore da noi detto

```
mov %al, (%rsi, %rcx)
```
- v2[i] = v1[i] / 2;
 - Il valore è quello nel registro al, se ce ne dimentichiamo non è un problema un'istruzione in più

```
mov (%rsi, %rcx), %al
```
 - La divisione per due si ottiene con uno shift aritmetico a destra

```
sar $1, %al
```
 - Ci manca l'assegnamento, ma dobbiamo stare attenti alle differenze di tipo: *v2* è un array di *long*, *v1* un'array di *int*. Nel passaggio da *int* a *long* dobbiamo fare estensione con segno

```
movsbq %al, %rax
```

```
mov %rax, 24(%rsi, %rcx, 8)
```

il passaggio da *rax* è obbligato, non esiste l'istruzione *movsbq* con operando destinatario diverso da registro.
- v3[i] = 2 * v1[i];
 - Recuperiamo *v1[i]*

- ```

 mov (%rsi, %rcx), %al
 – La moltiplicazione per due si ottiene con uno shift logico a sinistra
 shl $1, %al
 – Dobbiamo passare da byte a long, con estensione di segno. Si fa in modo simile a come
 abbiamo gestito il passaggio da byte a quad.

 movsbl %al, %eax
 mov %eax, 4(%rsi, %rcx, 4)

• Incremento la variabile i memorizzata in memoria

 inc -24(%rbp)
 e ritorno alla condizione

 jmp .Lfor

• Concludiamo restituendo l'indirizzo dell'oggetto, e con le solite istruzioni

 mov -8(%rbp), %rax <--- restituisco this
 leave
 ret

```

### Traduzione della funzione membro *elab1*

```

.global _ZN2cl5elab1EPc3st2
_ZN2cl5elab1EPc3st2:
 push %rbp
 mov %rsp
 sub $96, %rsp
 mov %rdi, -8(%rbp)
 mov %rsi, -16(%rbp)
 mov %rdx, -24(%rbp)
 mov %ecx, -32(%rbp)

 # inizializzazione
 movl $0, -28(%rbp)
.Lfor1: # confronto
 cmpl $4, -28(%rbp)
 jl .Lcorpo1
 jmp .Lfinefor1
.Lcorpo1:
 movslq -28(%rbp), %rcx
 mov -24(%rbp), %rdi
 movsbl (%rdi, %rcx), %eax
 add %ecx, %eax
 mov %al, -40(%rbp, %rcx)

 # inc
 incl -28(%rbp)
 jmp .Lfor1
.Lfinefor1:
 # &cla -> %rdi

```

```

lea -96(%rbp), %rdi
&s1 -> %rsi
lea -40(%rbp), %rsi

secondo for
inizializzazione
movl $0, -28(%rbp)

1: # confronto
 cmpl $4, -28(%rbp)
 jl 2f
 jmp 3f
2:
 # trovo s2.vd[i]
 movslq -28(%rbp), %rcx
 mov -32(%rbp, %rcx), %al

 # assegnamento cla.v3[i] = s2.vd[i]
 # attenzione, si assegna un char a un intero ... dobbiamo estendere
 movsbl %al, %eax
 mov %eax, -92(%rbp, %rcx, 4)

 # inc
 incl -28(%rbp)
 jmp 1b
3:
 # return cla
 lea -96(%rbp), %rsi
 mov -8(%rbp), %rdi
 mov $7, %rcx
 rep movsq

 leave
 ret

```

- Scriviamo il nome della funzione

`_ZN2c15elab1EPc3st2`

si consideri che *elab1* è una funzione membro, che riceve in ingresso un array di *char* (puntatore al primo elemento) e un oggetto di tipo *st2* passato per valore (ci allarmiamo, ma vediamo che è una struttura banale, di dimensione inferiore a 16 byte, senza costruttori e distruttori, segue un passaggio mediante registri).

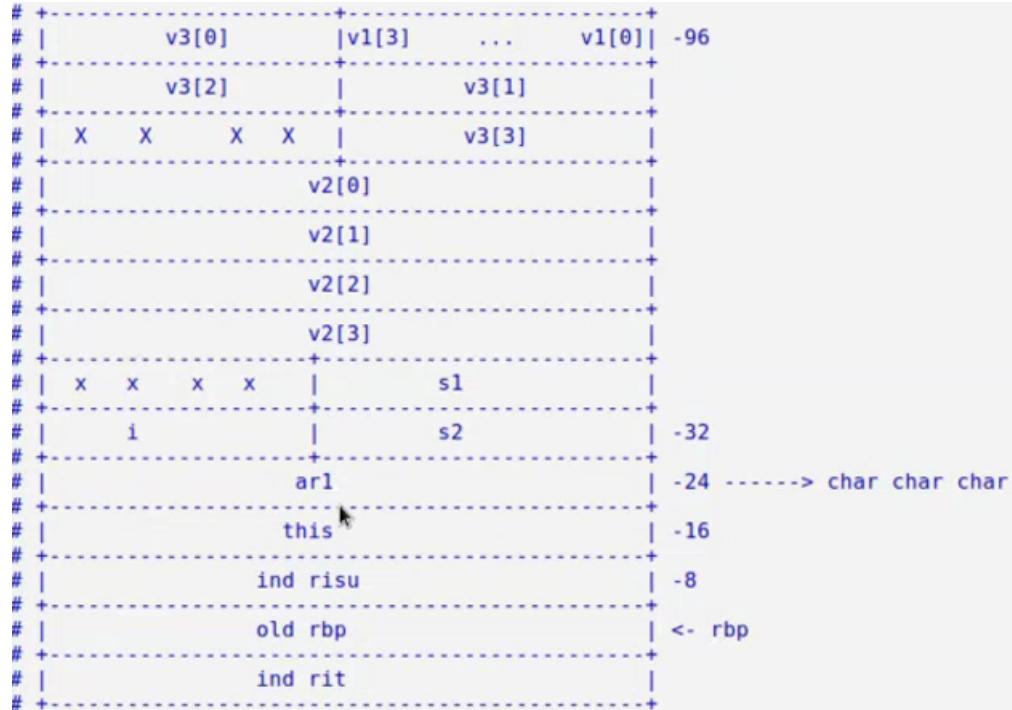
- La funzione restituisce un oggetto di tipo *cl* per valore. Purtroppo qua abbiamo un oggetto di dimensione di 56byte: non possiamo passare per registri. Non abbiamo costruttori di copia o distruttori. Il chiamante ci dice dove vuole il risultato (parametro *this*), il chiamato (*elab*) si occupa di inizializzare il risultato (con l'espressione che si trova nel *return*). Non abbiamo un costruttore di copia definito dall'utente, quindi poniamo il costruttore di copia di default (istruzione stringa).

- **Importante.** Attenzione ai parametri impliciti in ingresso. In questo caso dobbiamo inserire sia il puntatore *this* (funzione membro) che il puntatore all'oggetto risultato (poichè l'istanza restituita ha dimensione superiore ai 16byte). Come facciamo?

- In rdi va l'indirizzo dell'oggetto risultato (primo parametro, implicito)
- In rsi va *this* (secondo parametro, implicito)

seguono gli altri parametri.

- **Ci sono ottimizzazioni obbligatorie da fare in tutto questo?** No. Potrei utilizzare la NRVO, non allocando *cla* (utilizzo direttamente l'oggetto risultato allocato dal chiamante), ma non è una cosa obbligatoria secondo lo standard C++ 17.
- In considerazione di tutto quanto vediamo la struttura del record di attivazione:



Ricordarsi che *arl* è puntatore al primo elemento di un array (lo riscrivo perché sono duro come una pigna). Facciamo il solito prologo

```
push %rbp
mov %rsp
sub $96, %rsp
```

e inizializziamo i vari elementi

```
mov %rdi, -8(%rbp)
mov %rsi, -16(%rbp)
mov %rdx, -24(%rbp)
mov %ecx, -32(%rbp)
```

anche la variabile *i*

```
movl $0, -28(%rbp)
```

- Per gestire il for dobbiamo utilizzare delle etichette, come già visto prima. Ricordiamoci che il compilatore vede solo etichette, non ha in testa il concetto di funzione. Segue che le etichette dovranno avere nomi diversi da quelli utilizzati in altre funzioni.

- Corpo del primo for:
  - Ad ogni iterazione dobbiamo eseguire la seguente operazione
 

```
s1.vi[i] = ar1[i] + i;
```
  - Per prima cosa prendiamoci il contenuto del contatore *i*

```
movslq -28(%rbp), %rcx
```
  - Prendiamo l'indirizzo dell'array *ar1*

```
mov -24(%rbp), %rdi
```
  - Prendiamo l'elemento dell'array *i*-esimo, estendendolo con segno
 

```
movsbl (%rdi, %rcx), %eax
```
  - Aggiungiamo *i* all'elemento *ar1[i]* trovato prima
 

```
add %ecx, %eax
```
  - Eseguo l'operazione di assegnamento
 

```
mov %al, -40(%rbp, %rcx)
```
  - Concludo alla solita maniera, incremento il contatore *i* e salto alla condizione
 

```
incl -28(%rbp)
jmp .Lfor1
```
- Necessario chiamare il costruttore di copia per inizializzare l'istanza *cla* di *c1*. Passiamo l'indirizzo dell'oggetto risultato (che contiene i dati dell'istanza)
 

```
lea -96(%rbp), %rdi
```

non conosciamo l'indirizzo di *cla*, ma sappiamo come calcolarlo. L'unico parametro in ingresso esplicito è un riferimento a un'istanza della struttura *st1*.

```
lea -40(%rbp), %rsi
```

segue la chiamata del costruttore

```
call _ZN2c1C1ER3st1
```

- Corpo del secondo for

- Inizializziamo nuovamente la variabile contatore, la stessa utilizzata nel primo for

```
movl $0, -28(%rbp)
```

- Abbiamo posto le etichette del for in modo diverso rispetto a quello tradizionale. Precisamente:
  - \* Abbiamo posto come "etichette" i numeri 1,2,3. Etichette tra virgolette perché non rimarranno nel file oggetto: servono soltanto al compilatore per calcolare gli indirizzi verso cui fare jump.
  - \* Nelle istruzioni jump abbiamo indicato uno dei tre numeri e una lettera. In particolare
    - `jl 2f`  
indichiamo di voler saltare, con condizione soddisfatta, alla prima "etichetta" 2 che troviamo andando avanti (f per *forward*).

· **jmp 1b**  
 indichiamo di voler saltare, in qualunque caso, alla prima "etichetta" 1 che troviamo andando indietro (b per *backward*).

- \* La comodità principale sta nell'indicare, in ogni for, le "etichette" 1,2,3 senza dover inventare, ogni volta, etichette diverse per ogni step del for.

- Ad ogni iterazione dobbiamo eseguire la seguente operazione

```
cla.v3[i] = s2.vd[i];
```

- Poniamo *s2.vd[i]* nel registro al

```
movslq -28(%rbp), %rcx # Sposto il valore di i in rcx (necessario reg. a 64bit)
mov -32(%rbp, %rcx), %al
```

- Eseguiamo l'assegnamento

```
movsbl %al, %eax
mov %eax, -92(%rbp, %rcx, 4)
```

si consideri che avviene il passaggio da un char a un intero, quindi dobbiamo fare l'estensione con segno (per fare ciò siamo costretti a scrivere un ulteriore istruzione coinvolgendo il reg. eax)

- Concludo alla solita maniera, incremento il contatore *i* e salto alla condizione

```
incl -28(%rbp)
jmp 1b
```

- Andiamo all'istruzione *return*:

- La dimensione dell'oggetto restituito è superiore ai 16byte, quindi non utilizzeremo i registri per passare l'oggetto restituito. Ricordarsi che abbiamo posto nel registro rdi l'indirizzo dell'oggetto risultato. Il chiamante ha allocato lo spazio, noi dobbiamo limitarci ad inizializzarlo.
- L'inizializzazione dipende dall'espressione nel return, cioè *cla*. Il costruttore di copia non c'è, quindi si ricorre al costruttore di default (copia membro a membro).
- Per copiare tutto *cla* dentro l'oggetto allocato dal chiamante dobbiamo utilizzare le istruzioni stringa

```
lea -96(%rbp), %rsi
mov -8(%rbp), %rdi
mov $7, %rcx
rep movsq
```

ponendo, come al solito, *source*, *destination* e numero di volte in cui la copia deve essere ripetuta.

- Concludiamo come al solito

```
leave
ret
```

**Introduzione del costruttore di copia** Modifichiamo il codice C++ iniziale introducendo un costruttore di copia. La domanda che sorge spontanea è: cosa cambia nelle traduzioni scritte fino ad ora?

```
struct st1 { char vi[4]; };
struct st2 { char vd[4]; };
class cl {
 char v1[4]; int v3[4]; long v2[4];
public:
 cl(st1& ss);
 cl(const cl& c); # <-----
 ...
 ...
}
```

**Modifica della elab1** Dobbiamo sostituire la parte relativa al costruttore di copia di default limitandoci a chiamare il costruttore di copia da noi definito.

```
...
...
3:
 mov -8(%rbp), %rdi
 lea -96(%rbp), %rsi
 call _ZN2clC1ERKS_

 leave
 ret
```

Si osservi che non è necessario porre prima della *leave* il parametro da restituire

```
mov -8(%rbp), %rax
```

ci pensa il costruttore di copia!

**Proviamo a scrivere il costruttore di copia** Scriviamo il codice Assembler

```
global _ZN2clC1ERKS_
ZN2clC1ERKS:
 push %rbp
 mov %rsp, %rbp
 mov $rdi, %rdx

 mov $7, %rcx
 rep movsq
 mov -8(%rbp), %rax

 movb $42, (%rdx)
 leave
 ret
```

- Abbiamo scritto un costruttore di copia che non si limita a copiare come il costruttore di copia di default, ma fa anche qualcos'altro. Poniamo per comodità la traduzione in C++ del costruttore appena scritto

```
cl:cl(const cl&c) {
 for(int i = 0; i < 4; i++) {
 v1[i] = c.v1[i];
```

```

 v2[i] = c.v2[i];
 v3[i] = c.v3[i];
 }
 v1[0] = 42; # <----- ecco
}

```

- A questo punto ci chiediamo: l'output sarà quello che ci aspettiamo? No, a causa della NRVO. Non viene proprio chiamato il costruttore di copia, poichè non alloco *cla* e utilizzo direttamente lo spazio indicato dal chiamante attraverso l'apposito parametro implicito.
- L'ottimizzazione determina un output diverso. Ponendo nell'istruzione

`-fno-elide-constructors`

diciamo a g++ di non elidere i costruttori. A questo punto l'output dovrebbe essere quello immaginato.

### 13.3 Esercizio di traduzione [Scritto del 19/09/2018]

#### Codice C++ da utilizzare

```

struct st1 { char vc[4]; }; struct st2 { int vd[4]; };
class cl {
 st1 c1; st1 c2; long v[4];
public:
 cl(char c, st2& s);
 void elab1(st1 s1, st2 s2);
 void stampa() {
 for (int i=0; i < 4; i++) cout << c1.vc[i] << ' '; cout << "\n";
 for (int i=0; i < 4; i++) cout << c2.vc[i] << ' '; cout << "\n";
 for (int i=0; i < 4; i++) cout << v[i] << ' '; cout << "\n\n";
 }
};

```

#### Codice C++ da tradurre

```

cl::cl(char c, st2& s2) {
 for (int i = 0; i < 4; i++) {
 c1.vc[i] = c; c2.vc[i] = c++;
 v[i] = s2.vd[i] + c2.vc[i];
 }
}
cl& cl::elab1(st1 s1, st2 s2) {
 cl cla('a', s2);
 for (int i = 0; i < 4; i++) {
 if (c2.vc[i] <= s1.vc[i]) {
 c1.vc[i] = i + cla.c2.vc[i];
 v[i] = i - cla.v[i];
 }
 }
 return *this;
}

```

## Traduzione della funzione *elab1*

```

global _ZN2cl5elab1E3st13st2
_ZN2cl5elab1E3st13st2:
 push %rbp
 mov %rsp, %rbp
 sub $96, %rsp

 mov %rdi, -8(%rbp)
 mov %esi, -16(%rbp)
 mov %rdx, -32(%rbp)
 mov %rcx, -24(%rbp)

 # cl cla('a', s2)
 lea -80(%rbp), %rdi
 mov $'a', %sil
 lea -32(%rbp), %rdx
 call _ZN2clC1EcR3st2

 # init
 movl $0, -40(%rbp)
.Lfor: # confronto
 cmpl $4, -40(%rbp)
 jl .Lcorpo
 jmp .Lfinefor
.Lcorpo:
 # test
 mov -8(%rbp), %rdi
 movslq -40(%rbp), %rcx
 mov 4(%rdi, %rcx), %al
 cmpb %al, -16(%rbp, %rcx)
 jle .Lcorpoif
 jmp .Lfineif # se falso salto a fineif
.Lcorpoif:
 # c1.vc[i] = i + cla.c2.vc[i];
 movsbl -78(%rbp, %rcx), %eax
 add %ecx, %eax
 mov %al, (%rdi, %rcx)

 # v[i] = i - cla.v[i];
 mov -72(%rbp, %rcx, 8), %rax
 sub %rcx, %rax
 mov %rax, 8(%rdi, %rcx, 8)
.Lfineif:
 # incrmeento
 incl -40(%rbp)
 jmp .Lfor
.Lfinefor:
 mov -8(%rbp), %rax

 leave
 ret

```

- Consideriamo i parametri in ingresso: abbiamo un'istanza *s1* della struttura *st1*, e un'istanza *s2*

della struttura *st2*. Si consideri il parametro implicito *this* posto nel registro *rdi*, visto che *elab1* è una funzione membro della classe *cl*. Si consideri la struttura delle due classi: al loro interno sono presenti, rispettivamente, un array di *char* e un array di *int*.

- Scriviamo il nome dell'etichetta della funzione

`_ZN2cl5elab1E3st13st2`

- La funzione restituisce un riferimento a un oggetto di tipo *cl*, quindi non si ha il solito passaggio per valore. Non dobbiamo costruire un oggetto che la funzione dovrà inizializzare. L'implementazione avviene sottoforma di puntatore, quindi le dimensioni non sono problematiche e l'oggetto viene restituito via registri (non c'è bisogno di un ulteriore parametro隐式的).
- All'interno abbiamo una variabile locale, precisamente un'istanza *cla* della classe *emphcl*. Si considerino i dati appartenenti alla classe: due istanze della struttura *st1* e un array di *long*.
- In considerazione di quanto detto vediamo la struttura del record di attivazione

|    |          |   |          |        |
|----|----------|---|----------|--------|
| #+ | cla.c2   | + | cla.cl   | -80    |
| #  |          |   |          |        |
| #+ | cla.v[0] | + |          |        |
| #  |          |   |          |        |
| #  | cla.v[1] | + |          |        |
| #  |          |   |          |        |
| #  | cla.v[2] | + |          |        |
| #  |          |   |          |        |
| #  | cla.v[3] | + |          |        |
| #  |          |   |          |        |
| #  | x x x x  |   | i        | -40    |
| #  |          |   |          |        |
| #  | s2.vd[1] | + | s2.vd[0] | -32    |
| #  |          |   |          |        |
| #  | s2.vd[3] | + | s2.vd[2] |        |
| #  |          |   |          |        |
| #  | x x x x  |   | s1       | -16    |
| #  |          |   |          |        |
| #  | this     |   |          | -8     |
| #  |          |   |          |        |
| #  | old rbp  |   |          | <- rpb |
| #  |          |   |          |        |
| #  | ind rit  |   |          |        |
| #  |          |   |          |        |

Abbiamo il puntatore *this* (necessario, *elab1* è una funzione membro), i parametri locali (*s1* ed *s2*), le variabili locali (*cla*). Nell'allineamento degli elementi di *s1* ed *s2* abbiamo un certo margine di movimento, stessa cosa non possiamo dire nell'allineamento degli elementi di *cla* (due istanze di *st1*, ciascuna di 4byte, e un array di 4long, ciascuno di dimensione di 8byte). Scriviamo il solito prologo

```
push %rbp
mov %rsp, %rbp
sub $96, %rsp
```

- Inizializziamo le variabili locali

```
mov %rdi, -8(%rbp) # this
mov %esi, -16(%rbp) # s1

mov %rdx, -32(%rbp) # s2.vd[0], s2.vd[1]
mov %rcx, -24(%rbp) # s2.vd[3], s2.vd[2]
```

la variabile *i* sarà inizializzata più avanti, quando gestiremo il for. Per quanto riguarda l'istanza *cla* della classe *cl* dobbiamo chiamare l'unico costruttore definito nel codice C++.

```

lea -80(%rbp), %rdi
mov $'a', %sil
lea -32(%rbp), %rdx
call _ZN2clC1EcR3st2

```

Prima di chiamare il costruttore abbiamo indicato i parametri in ingresso: con le istruzioni `lea` abbiamo sistemato il puntatore *this* e il riferimento *s*, con la `mov` abbiamo indicato il valore della variabile *char*.

- Gestiamo l'unico `for` presente. A differenza di altri esercizi dobbiamo occuparci anche di un `if`, presente nel corpo del `for`.

- Inizializziamo la variabile contatore *i*

```
 movl $0, -40(%rbp)
```

- Gestiamo l'`if` introducendo ulteriori etichette.

\* I jump legati al controllo della condizione dell'`if` vengono posti come nel `for`, in modo da non dover verificare la condizione opposta (il programma è più intuitivo per chi legge).

```

 mov -8(%rbp), %rdi
 movslq -40(%rbp), %rcx
 mov 4(%rdi, %rcx), %al
 cmpb %al, -16(%rbp, %rcx)
 jle .Lcorpoif
 jmp .Lfineif # se falso salto a fineif

```

Utilizziamo le prime due righe per porre in registri il puntatore *this* e la variabile contatore *i* (che va in *rdx*, a 64bit, per questioni di indirizzamento). Per quanto riguarda il confronto dobbiamo paragonare due cose che stanno in memoria: non possiamo porre due indirizzi di memoria in una stessa istruzione, segue che uno dei due elementi parte del confronto dovrà essere spostato in un registro.

- \* Gestiamo le due operazioni di assegnamento

```

 · # c1.vc[i] = i + cla.c2.vc[i];
 movsbl -78(%rbp, %rcx), %eax
 add %ecx, %eax
 mov %al, (%rdi, %rcx)

```

Per il primo assegnamento: sposto nel registro *eax* *cla.c2.vc[i]*, sommo ad esso il registro *rcx* (dove si trova il valore di *i*), pongo il valore sommato in *c1.vc[i]*.

```

 · # v[i] = i - cla.v[i];
 mov -72(%rbp, %rcx, 8), %rax
 sub %rcx, %rax
 mov %rax, 8(%rdi, %rcx, 8)

```

Per il secondo assegnamento: sposto nel registro *rax* *cla.v[i]*, sottraggo a *rax* il contenuto del registro *rcx* (dove si trova il valore di *i*), pongo il risultato della sottrazione in *v[i]*.

- Per quanto riguarda l'oggetto da restituire consideriamo che dobbiamo ottenere un riferimento.

```
return *this;
```

l'operatore di dereferenziazione è necessario per creare un riferimento all'oggetto puntato da *this*. La cosa si traduce in Assembler passando mediante registro l'indirizzo dell'oggetto puntato da *this*.

```
mov -8(%rbp), %rax
```

- Concludiamo come al solito con *leave* e *ret*.

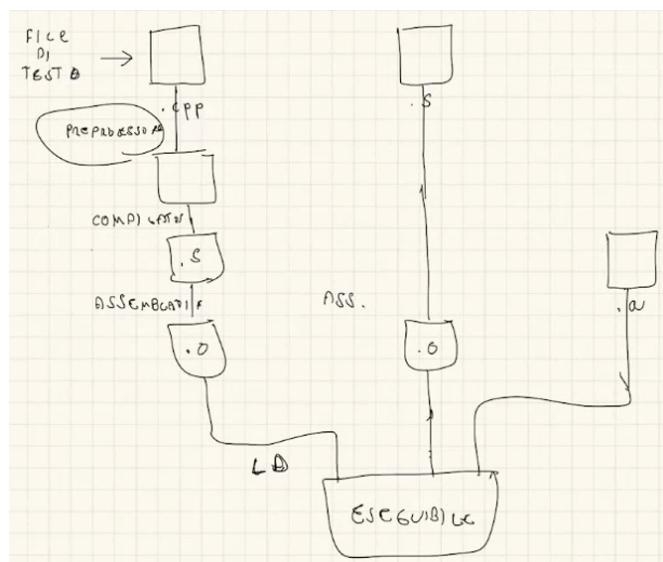
# Capitolo 14

Lunedì 29/03/2021 e Martedì  
30/03/2021

## 14.1 Concludiamo su compilatore, assemblatore e collegatore

Oggi concludiamo la parte sul compilatore esaminando in modo più organico alcune questioni.

- Per creare un programma si può partire da vari sorgenti, che possono essere scritti in C++, ma anche in Assembler. Si aggiungono librerie *statiche* (vedremo a breve) con formato *a*.
- Tutti questi files, da noi introdotti, sono files di testo che passano da vari strumenti. Per quanto riguarda il C++
  - preprocessore (ottengo un file .cpp in cui le direttive sono state processate)
  - compilatore (il risultato è un file Assembler)
  - assemblatore (il risultato è un file oggetto)
- I files oggetto e le librerie vanno in ingresso nel collegatore, che produce l'eseguibile.
- In Linux si utilizza un unico formato per librerie, files oggetto ed eseguibili.



### 14.1.1 Preprocessore

Prendiamo un programma C++ (la funzione *foo* è definita in un altro file assembler, che non ci interessa)

```
#include "lib.h"

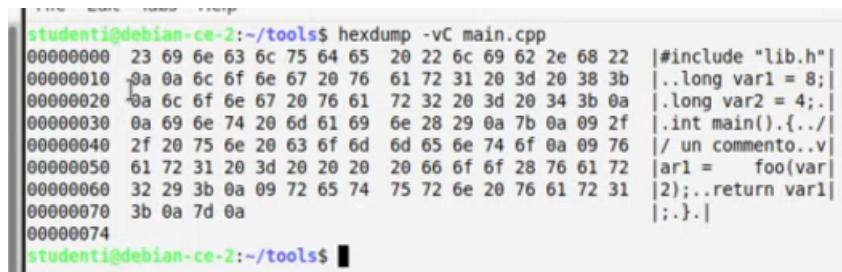
long var1 = 0;
long var2 = 4;

int main() {
 // UN COMMENTO
 var1 = foo(var2);
 return var1;
}
```

Tutti i file in UNIX sono semplicemente sequenze di byte, l'interpretazione di un file dipende dallo strumento utilizzato per esaminarlo. Col seguente strumento

```
hexdump -vC main.cpp
```

esaminiamo il file leggendone i corrispondenti esadecimali (la cosa più vicina all'analisi del file nel suo formato binario).



```
studenti@debian-ce-2:~/tools$ hexdump -vC main.cpp
00000000 23 69 6e 63 6c 75 64 65 20 22 6c 69 62 2e 68 22 |#include "lib.h"
00000010 0a 0a 6c 6f 6e 67 20 76 61 72 31 20 3d 20 38 3b |..long var1 = 8;|
00000020 0a 6c 6f 6e 67 20 76 61 72 32 20 3d 20 34 3b 0a |.long var2 = 4;.|.
00000030 0a 69 6e 74 20 6d 61 69 6e 28 29 0a 7b 0a 09 2f |.int main().{..}|
00000040 2f 20 75 6e 20 63 6f 6d 6d 65 6e 74 6f 0a 09 76 |/ un commento..v|
00000050 61 72 31 20 3d 20 20 20 20 66 6f 6f 28 76 61 72 |var1 = foo(var|
00000060 32 29 3b 0a 09 72 65 74 75 72 6e 20 76 61 72 31 |[2]);..return var1|
00000070 3b 0a 7d 0a |;..|.|
00000074
studenti@debian-ce-2:~/tools$
```

- La prima colonna da sinistra rappresenta l'offset rispetto al file
- Con l'opzione -vC chiedo la stampa a fianco del contenuto del file (se i caratteri sono stampabili, solitamente pone un punto se il carattere non è stampabile<sup>1</sup>).

#### 14.1.1.1 Analisi di quanto fatto dal preprocessore

Possiamo chiedere a g++ di fermarsi subito dopo il preprocessore

```
g++ -E main.cpp
```

L'output risultante, dal programma precedente è il seguente...

```
1 "main.cpp"
1 "<built-in>"
1 "<command-line>"
1 "/usr/include/stdc-predef.h" 1 3 4
```

<sup>1</sup>Tra i caratteri non stampabili c'è quello per andare a capo.

```

1 "<command-line>" 2
1 "main.cpp"
1 "lib.h" 1
long foo(long);
void bar();
2 "main.cpp" 2

long var1 = 0;
long var2 = 4;

int main() {
 var1 = foo(var2);
 return var1;
}

```

- Le prime righe sono informazioni su quanto fatto dal preprocessore, il compilatore le ignorerà.
- Osserviamo che la riga di inclusione di *lib.h* è stata sostituita col contenuto del file stesso.
- Sono stati rimossi i commenti, che servono solo a noi.
- Ha eliminato lunghe sequenze di spazi bianchi
- Le seguenti righe

```

1 "lib.cpp" 1
...
2 "main.cpp" 2
...

```

indicano che si sono inclusi questi due files. i numeri in fondo indicano che il preprocessore ha incontrato il file per la prima e la seconda volta, rispettivamente. Col primo numero, invece, indichiamo da quale riga stiamo ripartendo con le successive righe di codice.

Il formato è più canonico: sono state rimosse cose che non servono al compilatore, inoltre si è obbedito alle direttive indicate. Un tempo il preprocessore era effettivamente indipendente dal compilatore: questa cosa va bene per il C, ma risulta una cosa molto costosa nel C++. Poichè il preprocessore deve fare un parsing per considerare il file, come il compilatore, si è preferito integrare il preprocessore nel compilatore: un solo parsing per tutto.

#### 14.1.1.2 Inclusione di files

Attenzione all'inclusione di files: certe volte viene fatta con virgolette, altre volte con parentesi angolari.

```
#include "lib.h"
#include <lib.h>
```

- Nel primo caso si indica di cercare il file nella stessa directory del file che stiamo preprocessorando, e solo dopo guardare una serie di directories predefinite.
- Nel secondo caso si indica di cercare il file solo nelle directory predefinite (vengono usate per evitare il problema di nomi di file uguali a quelli di librerie presenti nelle directories predefinite).

#### 14.1.1.3 Definizione di macro

Se noi scriviamo una macro denominata *PIPPO*

```
#include "lib.h"
#define PIPPO var1
...
int main() {
 ... var1 = foo(PIPPO); ...
}
```

il preprocessore interviene restituendo quanto segue

```
long foo(long);
void bar();
...
int main() {
 ... var1 = foo(var1); ...
}
```

Si pone *var1* ovunque sia presente *PIPPO* (in questo caso si sostituisce ponendo *var1* come nome di una variabile, e non come una stringa).

#### 14.1.1.4 Inclusione condizionale di parti di testo

Possiamo includere o meno parti di codice secondo due criteri:

1. l'aver definito o no una macro

```
#include "lib.h"
#ifndef DEBUG
void debug(const char*);
#else
#define debug(msg);
#endif
...
int main() {
 ...
 var1 = foo(PIPPO);
 debug("questo e' un messaggio di debug");
 ...
}
```

- **Cosa succede se non ho definito la macro DEBUG?**

- La funzione *debug* non esiste.
- La direttiva

```
#define debug(msg)
```

sostituisce la chiamata di funzione *debug* con il niente (ricordarsi la sintassi della direttiva con cui si definisce una macro).

- **Cosa succede se definisco la macro DEBUG?**

- Poniamo in cima

```
#define DEBUG 1
```

- Viene inclusa la riga con cui viene dichiarata la funzione.
- La macro con cui si eliminano le chiamate di funzione non viene applicata.

2. avere una macro con un certo valore o no

```
#if DEBUG == 1
...
#endif
```

#### 14.1.1.5 Macro predefinite

Col seguente comando possiamo caricare una lista di macro predefinite

```
g++ -E -dM main.cpp
```

sono tante e definite in base a tanti fattori.

#### Esempi di macro

- *\_linux\_*, se mi trovo su linux o meno;
- *\_x86\_64\_*, se viene utilizzata l'omonima architettura...

#### 14.1.1.6 Curiosità: eredità dell'indipendenza del preprocessore

Un esempio di eredità dell'indipendenza del preprocessore dal compilatore l'abbiamo con la seguente riga

```
#define DEBUG 1;
```

il punto e virgola, che è parte della sintassi del C++, viene considerato parte del valore della macro *DEBUG*.

### 14.1.2 Assemblatore

L'assemblatore, essenzialmente, prepara il contenuto di sezioni di memoria:

- *data* e *text*, che già conosciamo;
- *bss*, sezione dove saranno collocate le variabili globali che hanno come valore zero. Nel file viene indicata la sola dimensione, non il contenuto (ovviamente).

**Come lavora l'assemblatore?** L'assemblatore lavora concettualmente con due passate: diciamo concettualmente perché non è detto che la cosa avvenga, fisicamente parlando. Parlare in questo modo è un'eredità del passato, quando il calcolatore doveva leggere per forza il programma due volte (pensare alle schede perforate, si dovevano fare due letture per estrarre delle informazioni, cosa necessaria visto che la memoria all'epoca era poca).

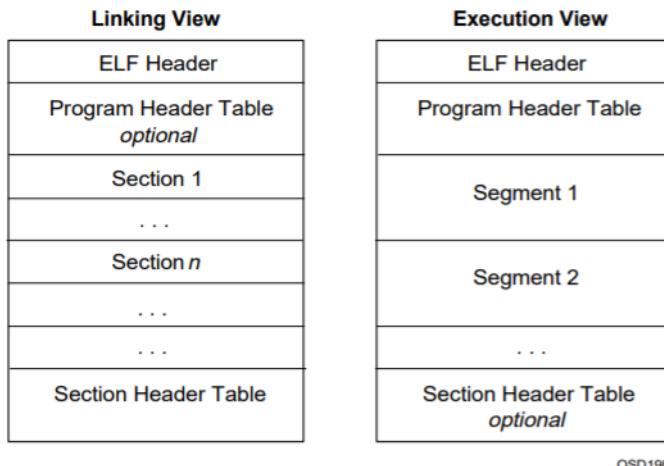
- Nella prima "passata concettuale" crea una tabella dei simboli, ogni volta che trova una nuova etichetta aggiunge una nuova entrata. Per ogni sezione mantiene un contatore, per ricordarsi il numero di byte effettivamente utilizzati in quella sezione. Per ogni etichetta individuata memorizza a quanto era arrivato il contatore: l'offset del corrispondente byte all'interno della sezione in cui l'etichetta è stata definita.
- Nella seconda passata traduce effettivamente il programma, crea i byte da porre nelle varie sezioni, utilizzando la tabella dei simboli.

Questa divisione concettuale in due passate permette di utilizzare etichette definite successivamente: contrariamente al C++ non dobbiamo dichiarare un'etichetta prima di usarla, è un qualcosa che è necessario per dare senso alle istruzioni JMP.

### 14.1.3 Formato ELF (*Executable and Linking Format*)

Abbiamo un file binario<sup>2</sup> in formato ELF (*Executable and Linking Format*). Il formato è comune per files oggetto, librerie statiche ed eseguibili.

#### Struttura stabilita dal formato ELF



<sup>2</sup>Se facciamo l'*hexdump* del formato oggetto individuiamo sequenze esadecimali molto lunghe, e poche cose stampabili.

- Intestazione ELF, unica parte fissa
- Parte rimanente, variabile e interpretabile in base al contenuto dell'intestazione. Il file ELF permette una doppia visione in base all'uso che si deve fare (ricordiamo che funge come file oggetto o eseguibile).
  - Gli strumenti interessati alla rilocazione (collegatore) prendono una **tabella delle sezioni**, che permette di recuperare le sezioni all'interno del file.
  - Gli strumenti interessati all'esecuzione del file prendono una **tabella del programma** (o **tabella dei segmenti**), che indica i segmenti presenti nel file, come devono essere caricati e utilizzati.
  - Attenzione all'*optional*: per i primi è facoltativa la cosa richiesta dai secondi, e viceversa.

**Esaminare un file ELF** Esistono degli strumenti che permettono di esaminare i file ELF in maniera più comoda. Uno di questi lo abbiamo già utilizzato: l'*objdump*.

```
objdump -o foo.o
```

Il programma non è specifico per i files ELF, quindi alcune cose non è in grado di interpretarle, o le interpreta in modo strano. Uno strumento più adatto è *readelf*. Scrivendo il seguente comando

```
readelf -h foo.o
```

siamo in grado di leggere, in modo chiaro, le informazioni codificate nell'intestazione ELF.

```
studenti@debian-ce-2:~/tools$ readelf -h foo.o
ELF Header:
 Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
 Class: ELF64
 Data: 2's complement, little endian
 Version: 1 (current)
 OS/ABI: UNIX - System V
 ABI Version: 0
 Type: REL (Relocatable file)
 Machine: Advanced Micro Devices X86-64
 Version: 0x1
 Entry point address: 0x0
 Start of program headers: 0 (bytes into file)
 Start of section headers: 472 (bytes into file)
 Flags: 0x0
 Size of this header: 64 (bytes)
 Size of program headers: 0 (bytes)
 Number of program headers: 0
 Size of section headers: 64 (bytes)
 Number of section headers: 8
 Section header string table index: 7
```

- Attenzione a quel *magic number*: è l'inizio di ogni file ELF, un identificativo dei files ELF.
- Tra le cose presenti abbiamo:
  - Informazioni sul sistema per cui è realizzato il programma.
  - Punto di partenza della tabella delle sezioni e della tabella dei programmi: il fatto che si abbia una tabella all'offset 0 ne stabilisce la sua assenza (all'offset zero è presente, per forza, il *magic number* detto prima).

- Numero di sezioni presenti e dimensione di ciascuna sezione (sostanzialmente è come un'array). Il formato, con lo scopo di essere il più generico possibile, non fissa numero di elementi presenti e dimensione degli stessi. In questo caso abbiamo 8 entrate, ciascuna da 64bit.

#### 14.1.3.1 Tabella delle sezioni

Possiamo chiedere a `readelf` di mostrarcici la tabella delle sezioni, col seguente comando

```
readelf -WS foo.o
```

```
studenti@debian-ca-2:~/tools$ readelf -WS foo.o
There are 8 section headers, starting at offset 0x1d8:

Section Headers:
[Nr] Name Type Address Off Size ES Flg Lk Inf Al
[0] NULL 0000000000000000 000000 000000 00 0 0 0 0
[1] .text PROGBITS 0000000000000000 000040 000022 00 AX 0 0 1
[2] .rela.text RELA 0000000000000000 000158 000048 18 I 5 1 8
[3] .data PROGBITS 0000000000000000 000062 000010 00 WA 0 0 1
[4] .bss NOBITS 0000000000000000 000072 000008 00 WA 0 0 1
[5] .syntab SYMTAB 0000000000000000 000078 0000C0 18 6 7 8
[6] .strtab STRTAB 0000000000000000 000138 00001d 00 0 0 1
[7] .shstrtab STRTAB 0000000000000000 0001a0 000031 00 0 0 1
Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
l (large), p (processor specific)
```

La sezione 0 è sempre nulla. Quali informazioniabbiamo per ciascuna sezione?

- Il nome, che è una stringa (le entrate con lunghezza fisica non danno problemi, contrariamente a quelle con lunghezza variabile). Per le stringhe si è deciso di riservare un'area dove queste vengono poste in sequenza (*strtab*, *shstrtab*), una dopo l'altra. Le entrate in dimensione fissa sono identificate immediatamente da un offset.
- Il tipo della sezione è un codice numerico, decodificato da `readelf` con una stringa. *PROGBITS* sono aree definite dal programmatore, altri tipi come *SYMTAB* e *STRTAB* sono definiti dallo standard.
- L'indirizzo dove la sezione viene caricata (l'assemblatore non lo sa, quindi tutto uguale a zero per ora).
- L'offset all'interno del file, dove si trova la sezione.
- Altre informazioni non significative per tutte le sezioni. Una di queste è il *flag*, che in un certo senso dice cosa si dovrà fare con quella sezione. **Sotto la tabella delle sezioni è presente una legenda** che suggerisce il significato dei vari flag. Vediamo alcuni esempi
  - **AX**: dobbiamo allocare spazio (A), e quanto allocato dovrà essere eseguibile (X).
  - **WA**: dobbiamo allocare spazio (A), e quanto allocato dovrà essere scrivibile (W).

#### 14.1.3.2 Sezione *syntab*: Tabella dei simboli

Possiamo chiedere a `readelf` di mostrarcici la tabella delle sezioni (*syntab*)

```
readelf -s foo.o
```

```
studenti@debian-ce-2:~/tools$ readelf -s foo.o

Symbol table '.symtab' contains 8 entries:
Num: Value Size Type Bind Vis Ndx Name
 0: 0000000000000000 0 NOTYPE LOCAL DEFAULT UND
 1: 0000000000000000 0 SECTION LOCAL DEFAULT 1
 2: 0000000000000000 0 SECTION LOCAL DEFAULT 3
 3: 0000000000000000 0 SECTION LOCAL DEFAULT 4
 4: 0000000000000000 0 NOTYPE LOCAL DEFAULT 3 foovar1
 5: 0000000000000008 0 NOTYPE LOCAL DEFAULT 3 foovar2
 6: 0000000000000000 0 NOTYPE LOCAL DEFAULT 4 foovar3
 7: 0000000000000000 0 NOTYPE GLOBAL DEFAULT 1 foo
```

Quali informazioni abbiamo in ciascuna entrata?

- Il nome del simbolo (nelle sequenze di bit relative alla tabella non si pone di nuovo il nome, ma l'offset relativo a *syntab* dove possiamo trovare il nome).
- Il valore, che consiste (in questo momento della compilazione) nell'offset del simbolo all'interno della sezione in cui è definito.
- *Ndx*, identificativo della sezione in cui si trova il simbolo.
- Dimensione e tipo del simbolo, non rilevanti per quanto ci riguarda.
- *Bind*, se il dato ha visibilità locale o globale.

#### 14.1.3.3 Sezione *rela.section\_name*: tabella di rilocazione

Abbiamo già visto col seguente comando

```
objdump -d foo.o
```

che l'assemblatore non conosce il valore del simboli. Si riservano aree mettendo come valori zero, e si lascia la palla al linker. Queste cose sono gestite mediante *tabelle di rilocazione*: ne produce una per ogni sezione che ne ha bisogno. La sezione è di tipo *RELA* (*Rilocazione con addendo*), attenzione al flag *I*, la tabella contiene informazioni per il linker). Nel nostro esempio la tabella è stata creata solo per la sezione *text*: si dice al linker come modificare la sezione *text* quando il valore di tutti i simboli sarà conosciuto. Leggiamo il contenuto col seguente comando

```
readelf -r foo.o
```

```
Relocation section '.rela.text' at offset 0x158 contains 3 entries:
 Offset Info Type Sym. Value Sym. Name + Addend
000000000009 000300000001 R_X86_64_64 0000000000000000 .bss + 0
000000000015 00020000000b R_X86_64_32S 0000000000000000 .data + 0
00000000001c 000200000002 R_X86_64_PC32 0000000000000000 .data + 4
studenti@debian-ce-2:~/tools$ cat foo.s
.data
foovar1:
 .quad 5
foovar2:
 .quad 6
.bss
foovar3:
 .quad 0
.text
Disassembly of section .text:
0000000000000000 <foo>:
 0: 55 push %rbp
 1: 48 89 e5 mov %rsp,%rbp
 4: 48 89 f8 mov %rdi,%rax
 7: 48 a1 00 00 00 00 00 00 movabs 0x0,%rax
 e: 00 00 00
11: 48 03 04 25 00 00 00 00 add 0x0,%rax
```

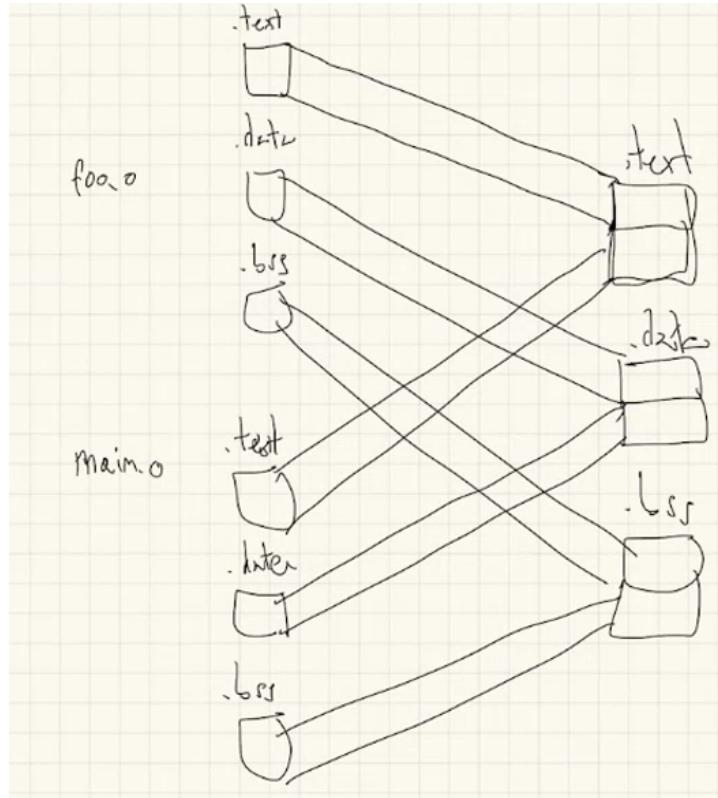
- All'offset 9 della sezione *text* c'è da fare l'operazione *64* (la parte iniziale del tipo è un prefisso legato al processore che stiamo utilizzando), cioè mettere un numero da 64bit a partire da questo offset. Quale numero? Un'operazione, in genere, fa riferimento a un simbolo. Nel caso delle rilocazioni con addendo si procede sommando una costante a un simbolo. In questo caso abbiamo *.bss + 0* (cioè *foovar3*): si chiede al linker di scrivere a partire dall'offset 9 della sezione *text* quando conoscerà il valore presente in *.bss + 0*.

#### 14.1.4 Collegatore

Cosa fa il collegatore? Scriviamo il seguente comando

```
ld -o main main2.o foo.o
```

Abbiamo due file oggetto, ciascuno con una sezione *text*, *data*, *bss*. La prima cosa che deve fare il linker è creare un qualcosa di unico: otterremo un'unica sezione *text*, un'unica sezione *data*, un'unica sezione *bss* (che saranno le sezioni del programma finale). La cosa viene fatta prendendo le varie sezioni nell'ordine in cui le abbiamo poste, in base al tipo.



- A questo punto il collegatore conosce la dimensione di ogni cosa, dunque può decidere gli indirizzi di partenza di ogni sezione. Pone le sezioni nell'ordine visibile nell'immagine, e ne decide gli indirizzi.
- Se io conosco l'indirizzo di partenza di ogni sezione sarò in grado di raggiungere ogni simbolo. Il linker, tenendo conto dei vari *syntab* (uno per file), deve intervenire aggiustando i vari *offset* relativi alle sezioni successive alla prima. Ottengo una tabella dei simboli finale.

- Tra questi simboli potrebbero esserci simboli non definiti. Il file finale, affinchè sia eseguibile, non deve avere simboli non definiti (ciò che non è stato definito in un file deve essere per forza definito da qualche altra parte).
- A questo punto possono essere eseguite tutte le istruzioni di rilocazione seguendo le informazioni presenti nella *tabella di rilocazione*.

**Informazioni nell'intestazione ELF dopo aver collegato tutto** Scriviamo

```
readelf -h main
```

```
student@debian-cw:~/tools$ readelf -h main
ELF Header:
 Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
 Class: ELF64
 Data: 2's complement, little endian
 Version: 1 (current)
 OS/ABI: UNIX - System V
 ABI Version: 0
 Type: EXEC (Executable file)
 Machine: Advanced Micro Devices X86-64
 Version: 0x1
 Entry point address: 0x401022
 Start of program headers: 64 (bytes into file)
 Start of section headers: 8736 (bytes into file)
 Flags: 0x0
 Size of this header: 64 (bytes)
 Size of program headers: 56 (bytes)
 Number of program headers: 4
 Size of section headers: 64 (bytes)
 Number of section headers: 8
 Section header string table index: 7
```

- Il tipo è cambiato: abbiamo un eseguibile (*EXEC*).
- La tabella dei programmi stavolta è presente (la tabella dei segmenti)!
- Cosa curiosa: continua ad essere presente la tabella delle sezioni. Non servono ma vengono lasciate: se proviamo a usare il comando già citato per leggere questa tabella non troveremo cose radicalmente diverse. Gli indirizzi e le dimensioni, stavolta, sono definite.
- Risulta presente anche la tabella dei simboli: differenza sostanziale è l'assenza di elementi non definiti (ricordiamo cosa abbiamo fatto prima). Sono presenti ulteriori simboli definiti dal linker, volendo possiamo utilizzarli nei nostri programmi.

Tutte queste cose potrebbero essere rimosse col seguente comando

```
strip main
```

non vengono rimosse proprio tutte le cose, ma ci si rende conto dell'irrilevanza della tabella delle sezioni e della tabella dei simboli nell'esecuzione del programma.

#### 14.1.4.1 Tabella dei segmenti

Concentriamoci sulla cosa nuova, cioè la *tabella dei segmenti*.

```
studenti@debian-te-2:~/tools$ readelf -Wl main
Elf file type is EXEC (Executable file)
Entry point 0x401022
There are 4 program headers, starting at offset 64

Program Headers:
Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align
LOAD 0x000000 0x0000000000400000 0x0000000000400000 0x000140 0x000140 R 0x1000
LOAD 0x001000 0x0000000000401000 0x0000000000401000 0x000045 0x000045 R E 0x1000
LOAD 0x002000 0x0000000000402000 0x0000000000402000 0x000020 0x000028 RW 0x1000
NOTE 0x000120 0x0000000000400120 0x0000000000400120 0x000020 0x000020 R 0x8

Section to Segment mapping:
Segment Sections...
00 .note.gnu.property
01 .text
02 .data .bss
03 .note.gnu.property
```

`readelf -Wl main`

Interpreto ogni riga nel seguente modo: carica a partire dall'offset *Offset* del file all'indirizzo *VirtAddr*, carica *FileSiz* byte, azzera ulteriori byte arrivando fino *memSiz* byte (se *FileSiz* = *memSiz* non si azzera nulla di più rispetto alla dimensione del file), successivamente faccio quanto indicato dai *Flg*.

## 14.2 Librerie statiche

In UNIX una libreria consiste in un archivio di file oggetto.

**Riflessione** Prendiamo tre files assembler: *foo.s*, *bar.s*, *main2.s*. Il contenuto del secondo file non viene utilizzato negli altri due. Per prima cosa creiamo i files oggetto

`g++ -c foo.s bar.s main2.s`

successivamente colleghiamo col linker

`ld -o main main2.o foo.o bar.o`

Tutti i files vengono uniti, indipendentemente dall'utilizzo delle funzioni. Cioè, posso unire a un set di files oggetto un ulteriore file oggetto con funzioni che non utilizziamo negli altri files.

### 14.2.1 Come si crea una libreria?

Col seguente comando

`ar cr mialib.a foo.o bar.o`

andiamo a creare un archivio, un unico file che contiene al suo interno altri files e presenta un'intestazione che permette di capire cosa è effettivamente presente nell'archivio. Il file creato (in GNU) è già pronto per essere usato.

### 14.2.2 Comandi utilizzati fino ad ora e librerie

I files archivio *a* sono ben noti dai programmi che abbiamo già visto. Vediamo alcuni esempi.

#### Lettura dell'intestazione ELF

```
readelf -h mialib.a
```

#### objdump

```
objdump -d mialib.a
```

```
studenti@debian-ce-2:~/tools$ readelf -h mialib.a
File: mialib.a(foo.o)
ELF Header:
 Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
 Class: ELF64
 Data: 2's complement, little endian
 Version: 1 (current)
 OS/ABI: UNIX - System V
 ABI Version: 0
 Type: REL (Relocatable file)
 Machine: Advanced Micro Devices X86-64
 Version: 0x1
Entry point address: 0x0
Start of program headers: 0 (bytes into file)
Start of section headers: 480 (bytes into file)
Flags: 0x0
Size of this header: 64 (bytes)
Size of program headers: 0 (bytes)
Number of program headers: 0
Size of section headers: 64 (bytes)
Number of section headers: 8
Section header string table index: 7

File: mialib.a(bar.o)
ELF Header:
 Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
 Class: ELF64
 Data: 2's complement, little endian
 Version: 1 (current)
 OS/ABI: UNIX - System V
 ABI Version: 0
 Type: REL (Relocatable file)
 Machine: Advanced Micro Devices X86-64
 Version: 0x1
Entry point address: 0x0
Start of program headers: 0 (bytes into file)
Start of section headers: 248 (bytes into file)
Flags: 0x0
Size of this header: 64 (bytes)
Size of program headers: 0 (bytes)
Number of program headers: 0
Size of section headers: 64 (bytes)
Number of section headers: 7
Section header string table index: 6

studenti@debian-ce-2:~/tools$ objdump -d mialib.a
In archive mialib.a:
foo.o: file format elf64-x86-64

Disassembly of section .text:
0000000000000000 <_Z3fool>:
 0: 55 push %rbp
 1: 48 89 e5 mov %rsp,%rbp
 4: 48 89 f8 mov %rdi,%rax
 7: 48 a1 00 00 00 00 movabs $0x0,%rax
 e: 00 00 00
11: 48 03 04 25 00 00 00 add 0x0,%rax
18: 00
19: 48 03 05 00 00 00 00 add 0x0(%rip),%rax # 20 <_Z3fool+0x20>
20: c9 leaveq %
21: c3 retq %

bar.o: file format elf64-x86-64

Disassembly of section .text:
0000000000000000 <_Z3bar>:
 0: 55 push %rbp
 1: 48 89 e5 mov %rsp,%rbp
 4: c9 leaveq %
 5: c3 retq %
```

**nm** Possiamo fare di più

```
nm -s mialib.a
```

```
studenti@debian-ce-2:~/tools$ nm -s mialib.a
Archive index:
_Z3fool in foo.o
_Z3bar in bar.o

foo.o:
0000000000000000 d foovar1
0000000000000008 d foovar2
0000000000000000 b foovar3
0000000000000000 T _Z3fool

bar.o:
0000000000000000 T _Z3bar
```

Con l'opzione *-s* possiamo vede l'indice che è stato creato nell'archivio: possiamo distinguere simboli globali e simboli locali, e vedere in che file sono stati definiti i simboli globali.

**ld** A questo punto posso collegare la mia libreria con il file *main.o*

```
ld -o main main2.o mialib.a
```

Ricerca nella libreria soltanto i simboli non definiti: per ogni simbolo non definito si porta dentro il corrispondente file oggetto. Il resto è uguale a prima. Se noi proviamo a usare *objdump* vedremo che sono state prese solo le cose necessarie (*bar*, l'intruso, non è più presente). Attenzione: conseguenza di quanto detto è che se noi invertiamo nel comando l'ordine dei file assemblati avremo errore. Convenzione vuole che le librerie vadano SEMPRE in fondo, in modo tale da considerare solo le cose che effettivamente ci servono (possiamo sapere cosa ci serve solo se processiamo gli altri files oggetto prima delle librerie).

**Directories per le librerie** Il linker è in grado di andare a cercare automaticamente le librerie in una serie di cartelle, in un certo senso come il preprocessore. Vediamo alcuni esempi

```
ls /usr/include/
ls /usr/lib/
ls /usr/local/lib/
```

La convenzione UNIX richiede che il nome cominci per *lib*. Inoltre, per copiare in quelle cartelle dobbiamo diventare amministratori. Copiamo utilizzando un comando leggermente diverso (viene richiesta la password, svolgiamo l'operazione come amministratore)

```
sudo cp mialib.a /usr/local/lib/libmia.a
```

nel comando per collegare richiameremo la libreria così

```
ld -o main main2.o -lmia
```

Attenzione a *lib* abbreviato con *l*. Per quanto riguarda *g++*, che tra tante cose utilizza *ld*, dobbiamo scrivere il seguente comando

```
g++ -no-pie -nostdlib -o main main2.o -lmia
```