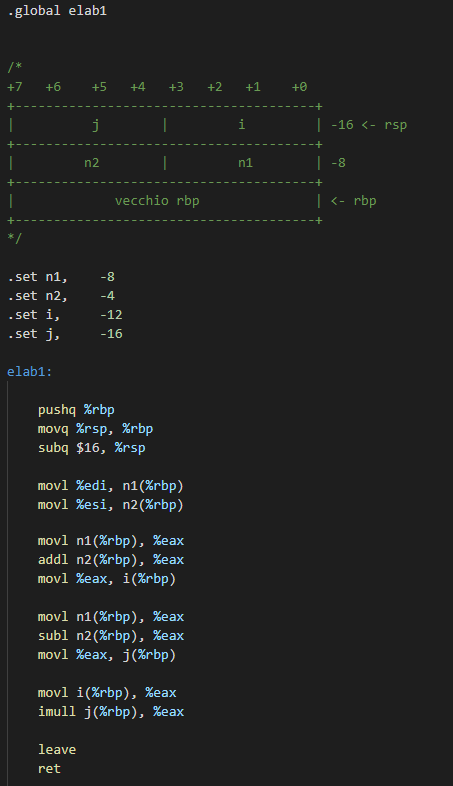
**Programma con parametri passati per valore**

Immagine che contiene testo

Descrizione generata automaticamente

Innanzitutto, il file .cpp ha il nome della funzione preceduto da extern “C”, dunque significa che la funzione rispetta lo standard del C. Questo significa che il nome della funzione va tradotto semplicemente con un’etichetta uguale al nome della funzione. La rendo poi globale, così che il collegatore possa vederla.

Si inizia col prologo, creando il record di attivazione della funzione. Salvo quindi in pila il valore di RBP, metto il valore attuale di RSP in RBP e devo vedere quanto spazio allocare per variabili locali e parametri d’ingresso. Per aiutarsi, **conviene farsi uno schemino** **dello stack**, ricordando che noi rappresentiamo gli offset di riga da dx a sx, e mettendo a fianco ad ogni riga l’offset rispetto a dove punta RBP.

Traduzione file es1b.cpp:

Fatto lo schemino, decidiamo quale spazio riservare ad ogni variabile locale e parametro d’ingresso. Possiamo metterli come vogliamo nello spazio in pila riservato ad essi, rispettando ovviamente il loro allineamento.

Una volta deciso dove metterli, possiamo vedere quanto spazio in pila serve. Servono 2 righe, quindi 16 byte. rsp deve essere allineato a 16, e facendo “sub $16, %rsp” lo sarà.

Fatto questo, **può essere utile definire attraverso la direttiva .set delle etichette dello stesso nome delle variabili locali/parametri d’ingresso e assegnare come valore l’offset di tali variabili locali/parametri d’ingresso rispetto a dove punta RBP**. Questo renderà gli indirizzamenti più semplici.

Copiamo adesso i parametri d’ingresso nella zona scelta da noi (le variabili locali no, non sappiamo ancora cosa metterci). Il primo parametro sta in %rdi, il secondo in %rsi. Visto che abbiamo due int in ingresso, che stanno su 4 byte, si trovano nei 4 byte meno significativi, quindi %esi e %edi**. Grazie ad aver definito le etichette prima, ogni cosa nello stack possiamo indirizzarla facendo *nome\_etichetta*(%rbp), che sarà *offset\_rispetto\_rbp + rbp.***  Così metterò i parametri all’indirizzo corretto.

Ora traduciamo il corpo della funzione vero e proprio. Iniziamo dalla somma. Poiché non possiamo sommare direttamente due operandi in memoria (non posso fare due indirizzamenti di memoria contemporaneamente), metto il primo in un registro, ad esempio eax, ci sommo il secondo, e poi metto il risultato nello spazio riservato alla variabile i, come mi chiede il programma C++ di partenza (anche qui, per indirizzarlo basta fare i(%rbp)).

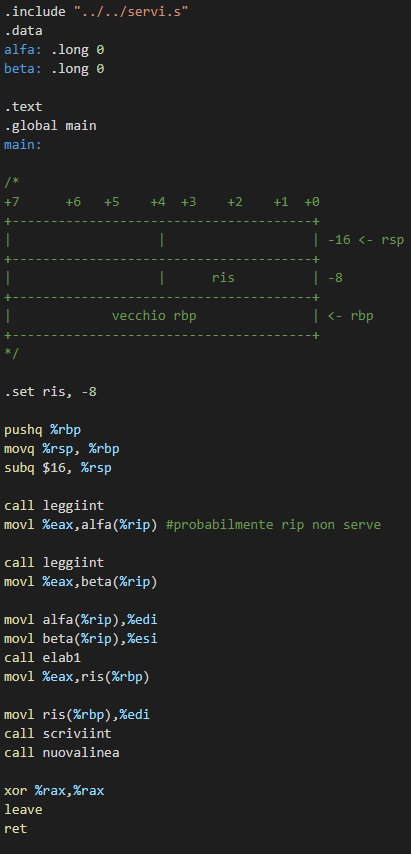
Ora traduciamo la differenza. Per lo stesso motivo di prima, metto il primo operando in un registro, ad esempio eax, ci sottraggo il secondo, e metto il risultato nello spazio riservato alla variabile j, come mi chiede il programma C++ di partenza.

Ora il programma mi chiede di ritornare i\*j. i e j sono due interi, cioè possono essere positivi o negativi, quindi per moltiplicarli uso la imul.

**Novità rispetto all’assembly a 32 bit**: la **imul** ora può accettare **due operandi**, e fa   
**operando\_sorgente \* operando\_destinatario**, mettendo il **risultato nel destinatario**. Per fare questo, serve che entrambi gli operandi abbiano stessa dimensione (qui c’è %eax che vincola l’operando sorgente a 32 bit, e per di più sta il suffisso l che non lascia dubbi, ma molti suffissi qui sono inutili).

Faccio allora il prodotto tra il contenuto della variabile i e il contenuto di eax, che contiene al momento il contenuto della variabile j, e il risultato finisce in eax. Il parametro d’uscita deve stare in %rax; ci sta già, poiché sta nei 4 byte più bassi (%eax).

Allora concludo con l’epilogo della funzione. Non ho usato registri preservati (a parte RBP, che gestisco con la leave quindi non lo considero), quindi concludo direttamente con leave e ret.

Traduzione file es1a.cpp:

Innanzitutto, nel file .cpp veniva incluso il file “servi.cpp”. #include è una direttiva per il preprocessore, che prima della compilazione sostituirà questa riga con il contenuto del file stesso. Dovremmo quindi tradurre in Assembly il file “servi.cpp” e mettere la sua traduzione dentro il file es1a.s. Dando per buona che sia stata già fatta e che si trova nella directory “../../servi.s” (quindi devo tornare indietro rispetto alla directory corrente due volte e lì sta il file servi.s, “..” significa questo), la includo attraverso la direttiva Assembly “.include”. **NB**: va inclusa fuori da ogni sezione, poiché anche questo file al suo interno avrà parti che saranno nella sezione .data, altre parti nella sezione .text. Se scrivessi .include dentro una sezione, l’assemblatore non capirebbe cosa va messo in una certa sezione e cosa no.

Abbiamo poi la dichiarazione delle variabili globali alfa e beta, e questo si traduce dichiarando due variabili con stesso nome nella sezione .data. Sono due int, quindi devo riservare 4 byte, e si fa con la direttiva .long. Non sono inizializzate nel file .cpp, e il compilatore traduce questo inizializzandole a 0. Il loro allineamento è 4. Ricordando che la data section verrà caricata sempre a partire da un indirizzo che ha allineamento 8, entrambe le variabili saranno messe in memoria rispettando il loro allineamento (alfa ad indirizzo “0 + indirizzo\_di\_partenza”; beta ad indirizzo “4 + indirizzo\_di\_partenza”, quindi non devo usare alcuna direttiva .balign).

Iniziamo a tradurre la funzione main. La traduzione di qualunque cosa non sia dichiarazione di variabili si mette nella sezione .text, dato che il programma dovrà sono leggerci e non anche scriverci (nel file prima non avevamo messo niente, automaticamente verrà messo tutto nella sezione .text). Dichiaro quindi l’etichetta main come globale e poi la definisco facendo “main:”. Si inizia col prologo, creando il record di attivazione della funzione. Salvo quindi in pila il valore di RBP, metto il valore attuale di RSP in RBP e devo vedere quanto spazio allocare per le variabili locali (main non ha parametri d’ingresso). Facciamo allora lo schemino dello stack, decidiamo quale spazio riservare a ris e vediamo che servono 4 byte. rsp però deve essere allineato a 16, e quindi devo fare almeno sub %16, %rsp per rispettare l’allineamento. Fatto questo definiamo l’etichetta ris cui assegniamo il valore dell’offset di ris rispetto a dove punta rbp.

Non devo copiare nulla in memoria per il momento, visto che non ho parametri d’ingresso.

Traduciamo ora il corpo della funzione. Traduciamo la prima riga: chiamiamo leggint. Non prende alcun parametro d’ingresso, quindi basta fare “call leggint”. Questa funzione leggerà un intero e lo restituirà, e come sempre i parametri d’uscita, se stanno entro 8 byte, vengono messi in rax. In particolare, poiché dovrà restituire un intero che sta su 4 byte, questo intero verrà messo solo sui 4 byte meno significativi di rax, e quindi eax.

**Sintassi nuova**. Ora devo mettere il valore di questo intero appena letto in alfa. Non so a priori dove verrà caricata la sezione data, dunque non so se alfa avrà un indirizzo per cui bastano 32 bit oppure ne servono 64, e facendo mov %eax, alfa, troncherei l’indirizzo dove sta alfa ai 32 bit meno significativi (offset sono a 32 bit), rischiando così di mettere eax in un posto a caso.

La soluzione allora è questa: **ogni volta che devo indirizzare una variabile globale “a”**, lo si fa facendo **a(%rip)**. Infatti, quello che succede è che al posto di a verrà messo dal collegatore l’offset tra rip e a, così che offset\_tra\_rip\_e\_a + rip mi dà effettivamente l’indirizzo dove sta a. Il vantaggio sta nel fatto che questo offset\_tra\_rip\_e\_a sta sempre su 32 bit, poiché il compilatore fa in modo che la sezione .data e la sezione .text non siano troppo distanti tra loro. Allora l’indirizzamento offset\_tra\_rip\_e\_a(%rip) funzionerà, poiché l’offset sta su 32 bit, e quindi così posso indirizzare le variabili globali senza rischi. Faccio quindi mov %eax, alfa(%rip).

Stesso ragionamento per la traduzione dell’altra riga.

Traduciamo ora l’altra riga. Dobbiamo chiamare la funzione elab1 passandogli i parametri alfa e beta. Vanno passati in rdi e rsi, e poiché alfa e beta sono su 4 byte, li passo sulla parte bassa dei registri, quindi edi e esi. Chiamo poi la funzione facendo call elab1, che metterà il parametro d’uscita in rax. In particolare, la elab1 lo metteva solo su eax, restituendo un intero. Devo mettere poi il risultato in ris, e faccio quindi mov %eax, ris(%rbp) (ris è nello stack).

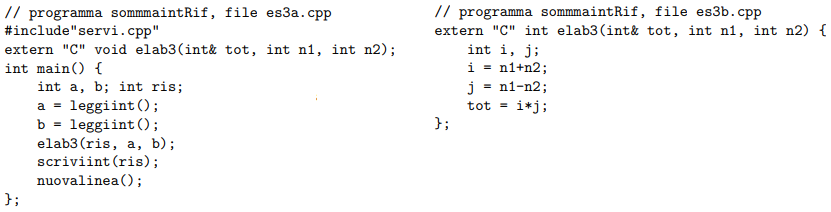
Traduciamo ora la prossima riga. Devo chiamare la scriviint, passandogli come parametro ris. Lo metto allora in %edi (parte bassa di %rdi) e faccio call scriviint.

Chiamo poi la funzione nuovalinea facendo call nuovalinea semplicemente.

La funzione main restituisce 0, e lo restituirà su rax, quindi faccio xor %rax, %rax.

Infine c’è l’epilogo. Non ho usato registri preservati (a parte RBP), quindi metto direttamente leave e ret.

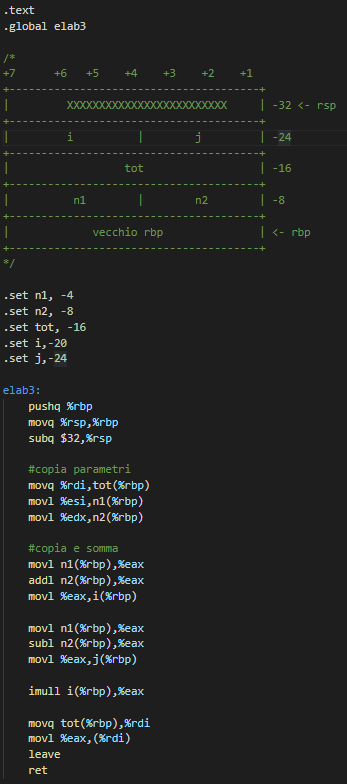
**Programma con parametri passati per riferimento**



Anche qui il file .cpp ha il nome della funzione preceduto da extern “C”, dunque il nome della funzione va tradotto semplicemente con un’etichetta uguale al nome della funzione. La rendo poi globale, così che il collegatore possa vederla.

Si inizia col prologo, creando il record di attivazione della funzione. Salvo quindi in pila il valore di RBP, metto il valore attuale di RSP in RBP e devo vedere quanto spazio allocare per variabili locali e parametri d’ingresso. Prima cosa importante: i riferimenti in Assembly non esistono**,** per cui **i riferimenti si traducono con un puntatore all’oggetto riferito**. Facciamo allora lo schemino dello stack e decidiamo quale spazio riservare ai 4 interi e all’indirizzo dell’oggetto passato per riferimento. Deciso dove metterli, vediamo che servono 3 righe, quindi 24 byte, ma rsp deve essere allineato a 16, quindi devo fare almeno sub $32, %rsp per mantenere l’allineamento.

Traduzione file es3b.cpp:



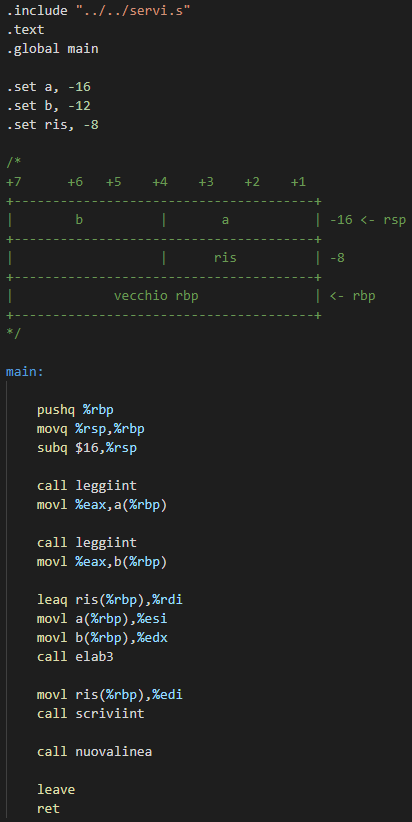
Fatto questo, definiamo le etichette con lo stesso nome delle variabili locali/parametri d’ingresso cui assegniamo il valore del loro offset rispetto a dove punta rbp.

Copiamo poi i parametri d’ingresso nella loro zona riservata

Traduciamo ora il corpo della funzione. Iniziamo dalle operazioni di somma e assegnamento, traducendole similmente a come visto nella traduzione di es1b.cpp.

Ora c’è una novità: **dobbiamo fare i \* j e mettere il risultato in tot, cioè proprio dentro l’oggetto che abbiamo passato per riferimento**. Il prodotto lo facciamo con la imul, come visto nella traduzione di es1b.cpp. Per mettere ora il risultato dentro tot, cioè proprio dentro l’oggetto che ho passato per riferimento, l’unico modo è usare **l’indirizzamento indiretto**: metto l’indirizzo dell’oggetto passato per riferimento in un registro a 64 bit, ad esempio rdi (conviene sceglierne uno scratch e che non mi serve più ovviamente), e metto il risultato del prodotto in (%rdi), cioè all’indirizzo puntato da %rdi, e quindi lo vado a mettere proprio dentro l’oggetto che ho passato per riferimento.

Infine c’è l’epilogo. Non ho usato alcun registro preservato (a parte RBP), quindi concludo facendo leave e ret.

Traduzione file es3b.cpp:

Nel file .cpp veniva incluso il file “servi.cpp”. Supponendo che la traduzione di questo file si trovi nella directory “../../servi.s”, la includo attraverso la direttiva Assembly “.include”. La .include va messa fuori da ogni sezione.

Iniziamo a tradurre la funzione main. La traduzione va messa nella sezione .text. Dichiariamo l’etichetta main come globale e, definendola facendo “main:”, iniziamo il prologo, creando il record di attivazione della funzione. Salvo quindi in pila il valore di rbp, metto il valore attuale di rsp in rbp e devo vedere quanto spazio allocare variabili locali e parametri d’ingresso. Facciamo allora lo schemino dello stack e decidiamo quale spazio riservare ai 3 interi. Deciso dove metterli, vediamo che servono 2 righe, quindi 16 byte, ma rsp deve essere allineato a 16, quindi devo fare almeno sub $32, %rsp per mantenere l’allineamento.

Fatto questo definiamo etichette con lo stesso nome delle variabili locali cui assegniamo il valore del loro offset rispetto a dove punta rbp.

Non devo copiare nulla in memoria per il momento, visto che non ho parametri d’ingresso.

Traduciamo ora il corpo della funzione. leggiint non prende alcun parametro, quindi traduco la chiamata facendo call leggiint. La funzione restituisce il valore in rax, e in particolare su eax, visto che restituisce un intero. Questo valore lo metto nella variabile *a* come mi chiede il programma c++.

Stessa cosa per tradurre la riga successiva.

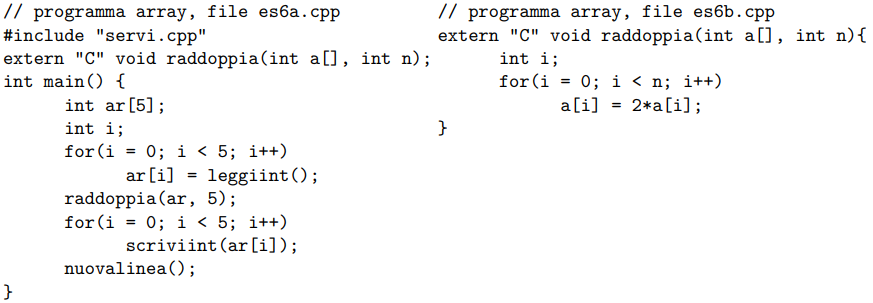
Ora devo chiamare la funzione elab3, passandole ris, a e b. Ricordiamo che ris viene passato per riferimento, dunque devo passare l’indirizzo di dove sta ris. Per fare questo, basta fare lea ris(%rbp), %rdi, e così metto in rdi proprio l’indirizzo di dove sta ris. Gli altri parametri li metto in rsi e rdx, in particolare nella loro parte bassa visto che sono int, quindi occupano solo 4 byte. Faccio poi call elab3.

Ora devo chiamare la funzione scriviint, passandole ris. Questo è un passaggio per valore, quindi metto proprio il valore di ris che sta in pila dentro edi (edi, poiché ris è a 4 byte) e faccio call scriviint.

Dopo devo chiamare la funzione nuovalinea, non passandole alcun parametro. Basta fare call nuovalinea.

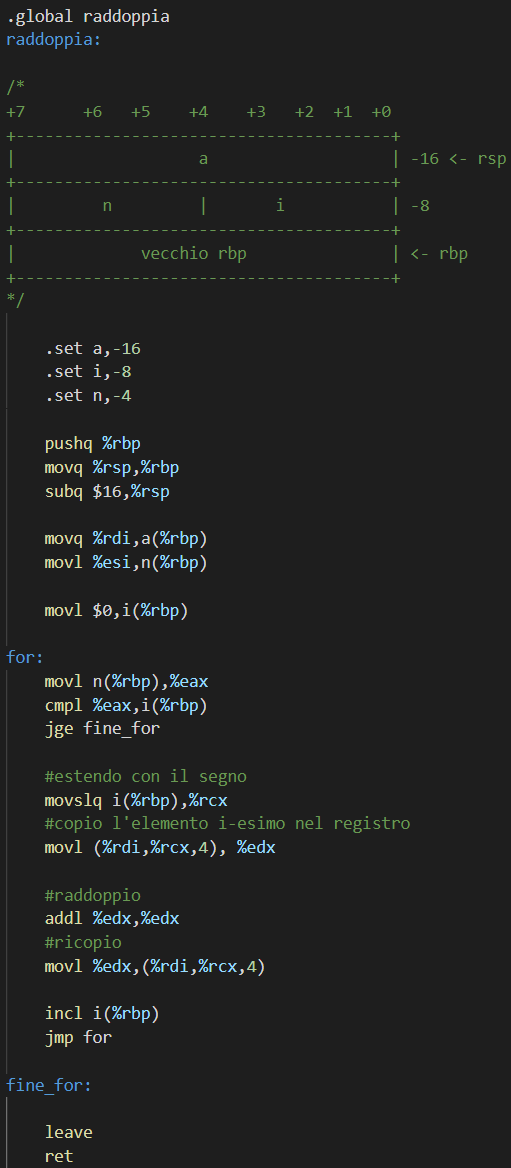
Infine c’è l’epilogo. Non ho toccato alcun registro preservato (a parte RBP), quindi concludo con leave e ret.

**Programma con parametro array**

****

Anche qui il file .cpp ha il nome della funzione preceduto da extern “C”, dunque il nome della funzione va tradotto semplicemente con un’etichetta uguale al nome della funzione. La rendo poi globale, così che il collegatore possa vederla.

Si inizia col prologo, creando il record di attivazione della funzione. Salvo quindi in pila il valore di RBP, metto il valore attuale di RSP in RBP e devo vedere quanto spazio allocare per variabili locali e parametri d’ingresso. **Quando passo un array come parametro, questo decade a puntatore al primo elemento**. Facciamo allora lo schemino dello stack e decidiamo quale spazio riservare ai 2 interi e all’indirizzo del primo elemento dell’array. Deciso dove metterli, vediamo che servono 2 righe, quindi 16 byte. rsp deve essere allineato a 16, e facendo “sub $16, %rsp” lo è.

Traduzione file es6b.cpp:

Fatto questo, definiamo le etichette con lo stesso nome delle variabili locali/parametri d’ingresso cui assegniamo il valore del loro offset rispetto a dove punta rbp.

Copiamo poi i parametri d’ingresso nella loro zona riservata.

Traduciamo ora il corpo della funzione. **Abbiamo una novità: c’è un ciclo for da tradurre.** Innanzitutto, traduco l’inizializzazione, e inizializzo *i* a 0.

A questo punto, traduco ciò che devo fare ciclicamente. Prima c’è da vedere se è soddisfatta la condizione, quindi metto il contenuto di *n* in un registro e confronto *n* con *i:* se *n* è maggiore o uguale a *i*, esco dal for (faccio il confronto al contrario, come visto anche a RL, ma non cambia nulla se l’avessi fatto diverso). Altrimenti, devo eseguire il corpo del for.

Nel corpo del for mi serve indirizzare l’elemento a[i]. Come visto a RL, per muoversi in un vettore conviene usare **l’indirizzamento indiretto** (*base*, *indice*, *scala*):

* in ***base*** metto l’indirizzo di partenza del buffer;
* in ***indice*** metto l’indice dell’elemento in cui mi trovo del vettore;
* in ***scala*** metto la dimensione di un elemento del vettore.

In questo modo, *base* + *indice* \**scala* mi dà proprio l’indirizzo dell’elemento a[indice].

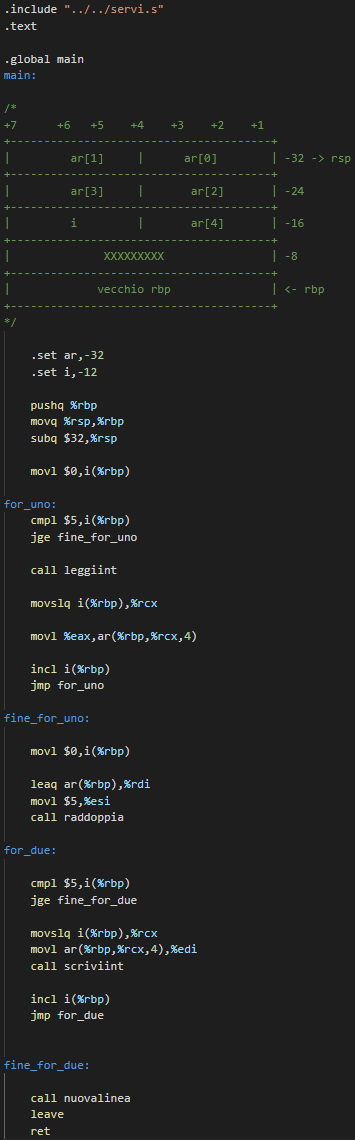
Ricordiamo che base e indice sono dei registri a 64 bit, quindi le prime due informazioni dobbiamo metterle in dei registri a 64 bit. L’indirizzo di partenza del buffer sta già in un registro a 64 bit, che è rdi.Anche *i* va messo un registro a 64 bit, ad esempio rcx, e poiché *i* è un intero (ha segno) ed è a 32 bit, per metterlo in un registro a 64 bit devo estenderlo con segno, e questo si fa attraverso l’istruzione **MOVSLQ.** Scala invece è una costante, e va messo 4 (è un array di int, quindi ogni elemento è 4 byte).

In questo modo, facendo “movl(%rdi, %rcx, 4), %edx”, metto in edx proprio il valore di a[i]. A questo punto, come mi dice la funzione, lo raddoppio e lo rimetto in a[i].

Infine traduco lo step, incrementando *i*, e risalto all’inizio del ciclo.

Usciti dal for, è finito il corpo della funzione, e quindi c’è l’epilogo. Non ho usato alcun registro preservato (a parte RBP), quindi concludo facendo leave e ret.

Infine c’è l’epilogo. Non ho usato alcun registro preservato (a parte RBP), quindi concludo facendo leave e ret.

Traduzione file es6a.cpp:

Nel file .cpp veniva incluso il file “servi.cpp”. Supponendo che la traduzione di questo file si trovi nella directory “../../servi.s”, la includo attraverso la direttiva Assembly “.include”. La .include va messa fuori da ogni sezione.

Iniziamo a tradurre la funzione main. La traduzione va messa nella sezione .text. Dichiariamo l’etichetta main come globale e, definendola facendo “main:”, iniziamo il prologo, creando il record di attivazione della funzione. Salvo quindi in pila il valore di rbp, metto il valore attuale di rsp in rbp e devo vedere quanto spazio allocare variabili locali e parametri d’ingresso. Facciamo allora lo schemino dello stack e decidiamo quale spazio riservare a un intero e ad un array di interi che ha dimensione 5. Deciso dove metterli, vediamo che servono 3 righe, quindi 24 byte, e allora devo fare almeno sub %32, %rsp per rispettare l’allineamento.

Fatto questo, definiamo etichette con lo stesso nome delle variabili locali cui assegniamo il valore del loro offset rispetto a dove punta rbp.

Non devo copiare nulla in memoria per il momento, visto che non ho parametri d’ingresso.

Traduciamo ora il corpo della funzione.

Anche qui c’è un for da tradurre. Innanzitutto, traduco l’inizializzazione, e inizializzo *i* a 0.

Poi traduco ciò che va fatto ciclicamente. Prima c’è da vedere se è soddisfatta la condizione, quindi confronto 5 con *i:* se *i* è maggiore o uguale a *5*, esco dal for (anche qui, faccio il confronto al contrario, come visto anche a RL, ma non cambia nulla se l’avessi fatto diverso). Altrimenti, devo eseguire il corpo del for. Traduciamo ora il corpo del for: devo assegnare ad a[i] il valore restituito da leggiint. Prima allora chiamo la leggint (non richiede parametri, quindi basta fare call leggiint), che mi restituirà un intero in EAX. Ora questo valore lo devo mettere dentro a[i]. Anche qui, per muoversi dentro un vettore conviene usare l’indirizzamento indiretto (base, indice, scala), mettendo i valori che abbiamo specificato prima. In particolare, visto che l’array ce l’abbiamo in pila, per partire dall’indirizzo di partenza del buffer basta fare ar(%rbp, …, …). A questo punto, si mette l’indice dell’elemento in cui mi trovo del vettore in un registro a 64 bit, ad esempio rcx, estendendolo prima con segno. La scala è 4 (la dimensione di un elemento del vettore è 4), e quindi per mettere il risultato di leggint dentro a[i] basta fare   
movl %eax, ar(%rbp, %rcx, 4).

**NB:** “ar(%rbp, %rcx, 4)” l’ho potuto fare perché ar(%rbp) è proprio l’indirizzo di partenza del buffer, e quindi facendo ar + rbp + indice \* scala mi muovo nell’array. Se invece in ar(%rbp) avessi avuto l’indirizzo del primo elemento, in questo caso ar(%rbp) non è appunto l’indirizzo di partenza, ma contiene l’indirizzo di partenza. Allora mi sarei prima dovuto salvare il contenuto di ar(%rbp) in un registro a 64 bit, es. %rdi e fare (%rdi, %rcx, 4), e **non** ar(%rbp, %rcx, 4), perché così mi muovo partendo da ar(%rbp), e quindi non nell’array!

Traduciamo ora la chiamata alla raddoppia: in rdi va messo l’indirizzo dell’array; in rsi (basta esi, visto che è su 4 byte) si mette 5, e poi si fa call raddoppia.

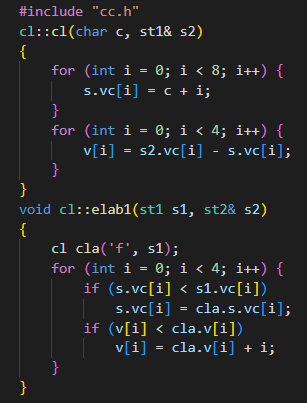
C’è poi un secondo for, roba simile.

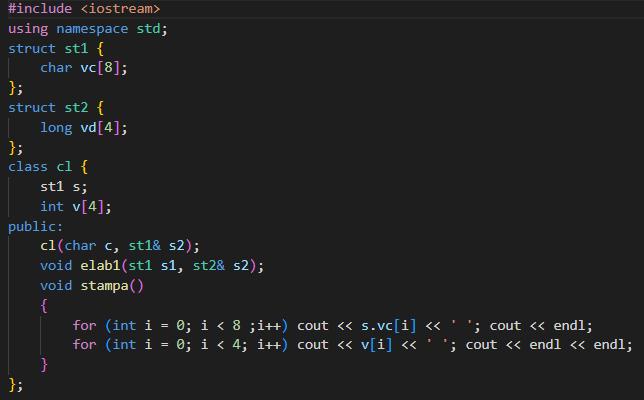
Dopo si chiama nuovalinea, senza doverle passare alcun parametro, quindi basta fare call nuovalinea.

Infine c’è l’epilogo. Non ho toccato alcun registro preservato (a parte RBP), quindi concludo con leave e ret.

**ATTENZIONE ALLA DIMENSIONE**: quando devi spostare robe di 1, 2, 4, 8 byte, usa sempre registri rispettivamente a 1, 2, 4, 8 byte. Questo per due motivi:

* Se devi leggere/scrivere in pila, si fa *etichetta*(%RBP). Il registro che scrivi nell’istruzione vincola anche quanti byte andranno letti/scritti partendo da *etichetta*(%RBP), e quindi mettendo il registro di dimensione corretta leggi/scrivi tutta e sola la variabile locale, senza sforare.
* Quando viene passato ad una funzione un parametro che occupa 1, 2 o 4 byte, viene scritto rispettivamente in AL, AX o EAX, senza toccare la parte alta (noi stessi abbiamo tradotto così). Quindi se invece prendessi il valore di rax, avresti come minimo i 32 bit più alti con valori sporchi, e quindi si avrebbero problemi.
* Quando una funzione deve restituire qualcosa che occupa 1, 2 o 4 byte, scrive solo in AL, AX o EAX, senza toccare la parte alta (noi stessi abbiamo tradotto così). Quindi se invece prendessi il valore di rax, avrei come minimo i 32 bit più alti con valori sporchi, e quindi si avrebbero poi problemi.

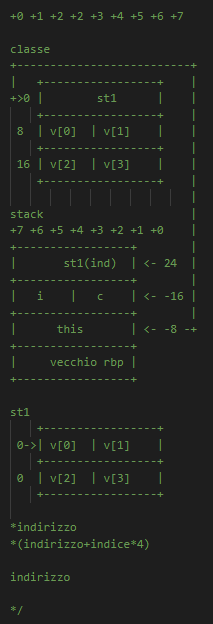
**Prova d’esame 15-06-16**



es1.cpp

cc.h

**Nelle prove d’esame vanno tradotte tutte le funzioni che compaiono nel file es1.cpp**

Traduzione costruttore cl

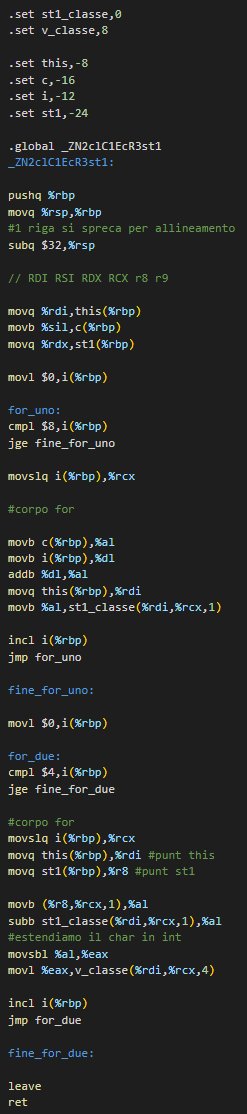
Innanzitutto, abbiamo a che fare con una classe cl. In questi casi, **conviene farsi uno schemino dello spazio occupato in memoria dalla classe, e conviene farlo con gli offset che vanno da sx a dx** (quindi in modo diverso rispetto a come facciamo lo schemino dello stack). Ricordiamo che lo spazio occupato in memoria dalla classe segue delle regole: si allocano i campi uno dopo l’altro, rispettando gli allineamenti.

Iniziamo ora a tradurre la prima funzione. il nome della funzione in es1.cpp non è preceduto da extern “C”, dunque significa che il nome della funzione va tradotto secondo lo standard del C++. E’ una funzione membro, ed è un costruttore, e seguendo le regole di traduzione per questo caso, si ottiene quel nome. La rendo poi globale, così che il collegatore possa vederla, e poi faccio “\_ZN2clC1ECR3st1: ”

Si inizia col prologo, creando il record di attivazione della funzione. Salvo quindi in pila il valore di RBP, metto il valore attuale di RSP in RBP e devo vedere quanto spazio allocare per variabili locali e parametri d’ingresso. **Ricordiamo che tutte le funzioni membro hanno come ulteriore parametro il puntatore this, cioè il puntatore all’oggetto classe su cui si chiama tale funzione**. **Poiché la funzione non deve restituire un oggetto >16 byte, questo sarà l’unico parametro implicito in più, e sarà il primo**. Facciamo allora lo schemino dello stack e decidiamo quale spazio riservare a un char, l’indirizzo dell’oggetto s2 passato per riferimento, un intero e l’indirizzo dell’oggetto classe su cui si chiama la funzione membro. Deciso dove metterli, vediamo che servono 3 righe, quindi 24 byte, e allora devo fare almeno sub %32, %rsp per rispettare l’allineamento. Fatto questo, definiamo etichette con lo stesso nome dei parametri d’ingresso/variabili locali cui assegniamo il valore del loro offset rispetto a dove punta rbp. Inoltre, **conviene definire delle etichette con lo stesso nome dei campi della classe cui assegniamo il valore dell’offset dei campi all’interno di un oggetto classe** (e cioè offset rispetto all’inizio dell’oggetto classe, che è anche dove punta *this*). Servirà per dopo.

Immagine che contiene testo, monitor, interni, schermo

Descrizione generata automaticamente



Copiamo poi i parametri d’ingresso nella loro zona riservata (attenzione alle dimensioni dei registri!!!”.

Traduciamo ora il corpo della funzione.

C’è un primo for, quindi prima traduciamo l’inizializzazione, azzerando *i.* Poi traduciamo ciò che va fatto ciclicamente. Prima va fatto il controllo tra 8 e *i* e, se *i* è maggiore o uguale a 8, si esce dal for; altrimenti, si entra nel corpo del for.

Nel corpo del for devo indirizzare s.vc[i], cioè l’elemento i-esimo da un array, quindi conviene usare l’indirizzamento indiretto *(base, indice, scala)*. Come prima cosa mettiamo l’indice *i* estendendolo con segnodentro un registro a 64 bit, ad esempio rcx. s.vc[] è un array dell’oggetto classe su cui viene chiamata la funzione membro, per cui l’indirizzo di partenza del buffer lo si ottiene partendo dall’indirizzo contenuto in *this* (che contiene l’indirizzo dell’inizio dell’oggetto classe) sommando *st1\_classe* (offset del campo st1 all’interno dell’oggetto classe). Quindi se metto *this* (e cioè this(%rbp) )dentro un registro a 64 bit, ad esempio rdi, per partire dall’indirizzo di partenza del buffer basta fare st1\_classe(%rdi, …, …). La scala è 1, perché ogni elemento dell’array vc ha dimensione 1. Allora facendo st1\_classe(%rdi, %rcx, 1) riesco ad indirizzare l’elemento i-esimo dell’array s.vc[].

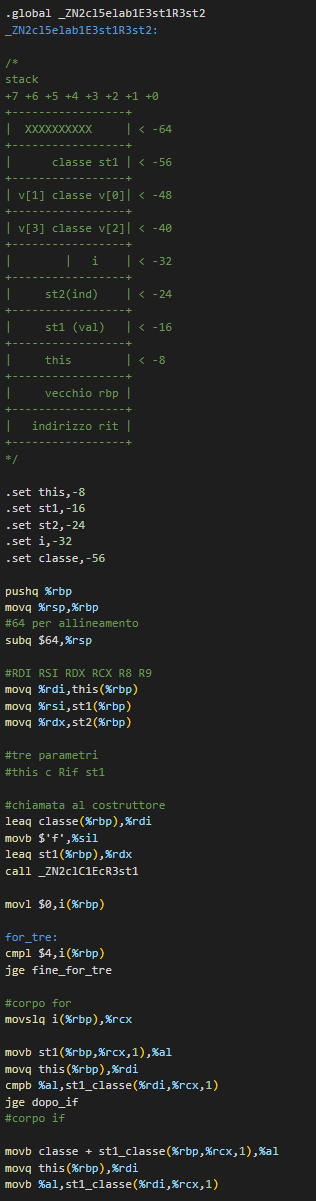
**NB:** in questo caso, prima ho messo *this(%rbp)* dentro un registro, perché l’indirizzo del buffer è *contenuto* in *this*(%rbp), ***non******è*** *this*(%rbp). Quindi non si deve fare this(%rbp, %rcx, 1), perché sennò mi muoverei partendo dall’indirizzo this(%rbp), e quindi non nell’array.

A questo punto sommo *c* e *i* e metto il risultato dentro l’elemento i-esimo dell’array.

Dopo traduco lo step, incrementando *i*, e risalto all’inizio del ciclo.

C’è un secondo for, quindi prima traduco l’inizializzazione, azzerando *i.* Poi traduciamo ciò che va fatto ciclicamente. Si parte dal controllo e poi si inizia a tradurre il corpo del for. Anche qui serve recuperare l’elemento i-esimo degli array v[], s2.vc[] e s.vc[], quindi usiamo l’indirizzamento indiretto *(base, indice, scala).* Per gli array s.vc[] e v[], essendo array dell’oggetto classe su cui viene chiamata la funzione membro, valgono le stesse considerazioni dette prima. Per quanto riguarda invece l’array s2.vc[], in questo caso è un array dell’oggetto s2 passato per riferimento, e quindi nello stack abbiamo il suo indirizzo. Per ottenere allora l’indirizzo di partenza del buffer, si parte da st1(%rbp) (che contiene l’indirizzo dell’inizio dell’oggetto s2) e si somma 0, poiché s2 è di tipo st1, che contiene solo tale array e basta. Quindi se metto st1(%rbp) dentro un registro a 64 bit, ad esempio r8, e per partire dall’indirizzo di partenza del buffer basta fare (%r8, …, …). L’indice *i* come sempre lo estendo con segno e lo metto dentro un registro a 64 bit, ad esempio rcx; la scala è 1 perché ogni elemento dell’array vc ha dimensione 1. Allora facendo (%r8, %rcx, 1), riesco ad indirizzare l’elemento i-esimo dell’array s1.vc[]. Per il resto, le traduzioni sono simili a prima (e ricorda alla fine di tradurre lo step)

Usciti dal for, è finito il corpo della funzione, e quindi c’è l’epilogo. Non ho usato alcun registro preservato (a parte RBP), quindi concludo facendo leave e ret.

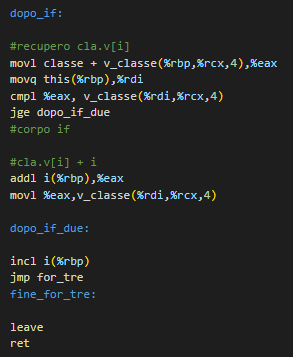
Traduzione elab1

Traduciamo ora elab1. Anche in questo caso, il nome della funzione in es1.cpp non è preceduto da extern “C”, dunque significa che il nome della funzione va tradotto secondo lo standard del C++ (nei temi d’esame sarà sempre così, quindi non lo ripeterò più). E’ una funzione membro, e seguendo le regole di traduzione per questo caso, si ottiene quel nome. La rendo poi globale, così che il collegatore possa vederla, e poi faccio “\_ZN2cl5elab1E3st1R3st2: ”

Si inizia col prologo, creando il record di attivazione della funzione. Salvo quindi in pila il valore di RBP, metto il valore attuale di RSP in RBP e devo vedere quanto spazio allocare per variabili locali e parametri d’ingresso. Ricordiamo di nuovo che tutte le funzioni membro hanno come primo parametro il puntatore this, cioè il puntatore all’oggetto classe su cui si chiama tale funzione. Facciamo allora lo schemino dello stack e decidiamo quale spazio riservare all’indirizzo dell’oggetto classe su cui si chiama la funzione membro, un oggetto st1, l’indirizzo di st2 passato per riferimento, un intero, un oggetto classe *cla* della classe cl. Deciso dove metterli, vediamo che servono 7 righe, quindi 56 byte, e allora devo fare almeno sub %64, %rsp per rispettare l’allineamento. Fatto questo, definiamo etichette con lo stesso nome delle variabili locali/parametri d’ingresso cui assegniamo il valore del loro offset rispetto a dove punta rbp.

**NB:** possiamo usare anche i nomi delle etichette usati prima, poiché con “.set” le etichette si possono ridefinire. In particolare, quando verrà chiamato il costruttore facendo CALL “\_Z…”, attraverso le .set che stanno dentro la traduzione del costruttore verranno usati quei valori delle etichette; quando verrà chiamata la elab1 facendo “CALL “\_Z…” attraverso le .set che stanno dentro la traduzione della elab1 verranno quei valori delle etichette (e quindi le .set vanno scritte dentro il corpo della funzione, così da non avere errori).

Copiamo poi i parametri d’ingresso nella loro zona riservata.

Immagine che contiene testo

Descrizione generata automaticamente

Traduciamo ora il corpo della funzione.

Viene dichiarato un oggetto classe, dunque c’è la chiamata al costruttore. **Come già detto, ricordiamo che quando viene chiamata una funzione membro, se la funzione membro non deve restituire un oggetto >16 byte, va passato come primo parametro l’indirizzo dell’oggetto classe su cui si chiama tale funzione.** E’ il nostro caso, visto che il costruttore è una funzione membro e non restituisce nulla, quindi in rdi dunque va messo l’indirizzo dell’oggetto classe *cla* su cui si chiama la funzione. *cla* si trova nello stack (è una variabile locale), e quindi per mettere il suo indirizzo in rdi si fa “**lea** classe(%rbp), %rdi” (lea appunto perché va messo l’indirizzo, non il valore del quad). Va poi passato ‘f’ per valore come secondo parametro, e s1 per riferimento come terzo parametro (vedi dichiarazione del costruttore), e quindi va passato il suo indirizzo, usando la lea. A questo punto si fa effettivamente la chiamata al costruttore, facendo “call \_ZN2c1C1EcR3st1”.

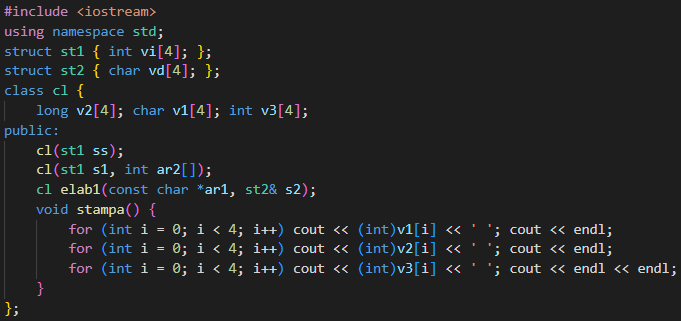
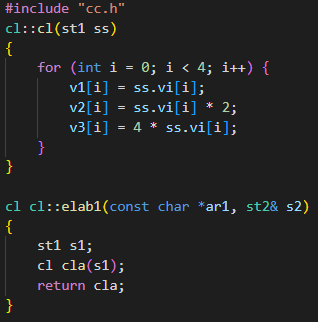
Ora c’è il for. Prima traduco inizializzazione. Nella parte che si fa ciclicamente, dopo il controllo della condizione, c’è il corpo del for. **Abbiamo un if.** Prima traduco la condizione (anche qui, facendo il confronto al contrario). Devo indirizzare elementi di un array, quindi uso l’indirizzamento indiretto *(base, indice, scala).* s.vc[i] abbiamo già visto come indirizzarlo; s1.vc[i] invece è un array dell’oggetto s1 che passiamo per valore, quindi nello stack abbiamo il suo valore. Allora per indirizzarlo, trovandosi nello stack, l’indirizzo di partenza del buffer si ottiene partendo da st1(%rbp, …, …). L’indice *i* lo estendiamo con segno e lo mettiamo dentro un registro a 64 bit, es. rcx, e la scala è 1, poiché l’array è fatto da elementi che hanno dimensione 1. Allora facendo st1(%rbp, %rcx, 1) indirizziamo l’elemento i-esimo dell’array s1.vc[].

Traduciamo allora il corpo dell’if. Anche qui dobbiamo indirizzare elementi di un array, e in particolare dobbiamo indirizzare cla.s.vc[i]. *cla* è contenuto nello stack, e si trova all’indirizzo “classe(%rbp)”. Il campo s si trova ad offset st1 classe all’interno della classe, e quindi si trova all’indirizzo “classe + st1\_classe(%rbp, …, …)”. L’array coincide con tutto s, e quindi l’indirizzo di partenza del buffer è effettivamente “classe + st1\_classe(%rbp, …, …)”. Metto poi indice e scala, e quindi l’elemento i-esimo del vettore cla.s.vc[] si ottiene facendo   
“classe + st1\_classe(%rbp, %rcx, 4)”.

C’è poi un altro if simile.

Alla fine va tradotto lo step.

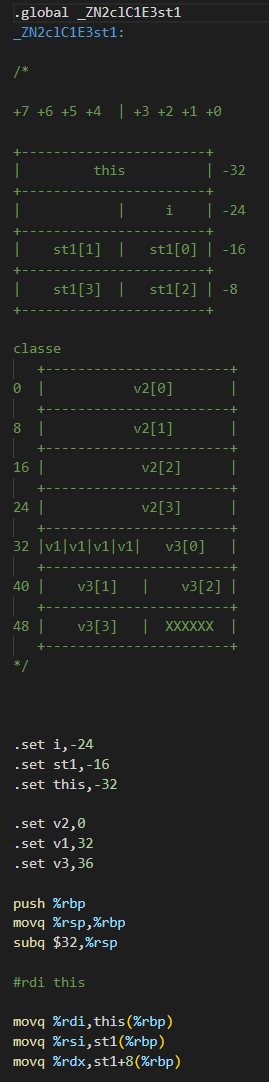
Usciti dal for, è finito il corpo della funzione, e quindi c’è l’epilogo. Non ho usato alcun registro preservato (a parte RBP), quindi concludo facendo leave e ret.

**Prova d’esame 05-02-20**

cc.h

es1.cpp

Traduzione costruttore cl

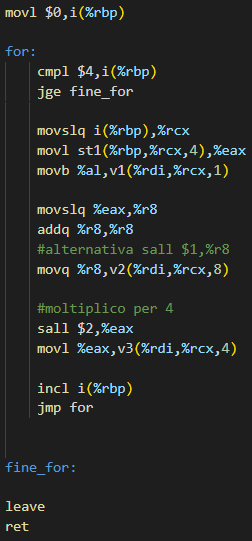


Innanzitutto, abbiamo a che fare con una classe cl, e quindi conviene farsi uno schemino dello spazio occupato in memoria dalla classe, con gli offset che vanno da sx a dx. Ricordiamo inoltre che lo spazio occupato in memoria dalla classe segue delle regole: si allocano i campi uno dopo l’altro, rispettando gli allineamenti.

Iniziamo ora a tradurre la prima funzione. E’ una funzione membro, ed è un costruttore, e seguendo le regole di traduzione per questo caso, si ottiene quel nome. La rendo poi globale, così che il collegatore possa vederla, e poi faccio “\_ZN2clC1E3st1: ”

Si inizia col prologo, creando il record di attivazione della funzione. Salvo quindi in pila il valore di RBP, metto il valore attuale di RSP in RBP e devo vedere quanto spazio allocare per variabili locali e parametri d’ingresso. Ricordiamo che tutte le funzioni membro hanno come ulteriore parametro il puntatore this, cioè il puntatore all’oggetto classe su cui si chiama tale funzione. Poiché la funzione non deve restituire un oggetto di dimensione >16 byte, questo sarà l’unico parametro implicito, e sarà il primo. Facciamo allora lo schemino dello stack e decidiamo quale spazio riservare ad un oggetto di tipo st1, un intero e l’indirizzo dell’oggetto classe su cui si chiama la funzione membro. Deciso dove metterli, vediamo che servono 4 righe, quindi 32 byte, e facciamo sub %32, %rsp, rispettando anche l’allineamento. Fatto questo, definiamo etichette con lo stesso nome dei parametri d’ingresso/variabili locali cui assegniamo il valore del loro offset rispetto a dove punta rbp. Inoltre, poiché abbiamo una classe, definiamo anche delle etichette con lo stesso nome dei campi della classe cui assegniamo il valore dell’offset dei campi all’interno di un oggetto classe**.**

Copiamo poi i parametri d’ingresso nella loro zona riservata. **Novità: il parametro ss occupa 16 byte, più di 8 byte.** In questo caso allora, il chiamante avrà passato i primi 8 byte in un registro (rsi); i secondi 8 byte nel registro successivo (rdx).



Traduciamo ora il corpo della funzione.

Traduciamo il for. Tradotta l’inizializzazione e il controllo della condizione, c’è il corpo del for.

Vediamo il primo assegnamento. Dobbiamo indirizzare elementi di un array, quindi usiamo l’indirizzamento indiretto *(base, indice, scala).* Innanzitutto, mettiamo l’indice *i* esteso con segno dentro RCX. ss si trova nello stack ed è di tipo st1. Il tipo st1 contiene solo l’array vi[], per cui l’indirizzo iniziale dell’array ss.vi[] si ricava partendo da st1(%rbp, …, …). La scala è 4, perché è un array di int, e quindi st1(%rbp, %rcx, 4) permette di indirizzare l’elemento i-esimo dell’array. Mettiamolo dentro un registro a 4 byte, es. eax.

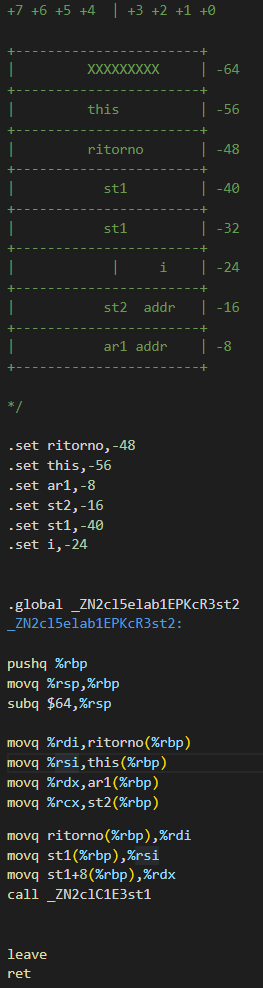
Questo dobbiamo metterlo dentro v1[i], che è un elemento di un array di char, dunque dobbiamo mettere solo il byte meno significativo (quindi %al). Per indirizzare v1[i], questo è un campo dell’oggetto classe su cui viene chiamata la funzione membro, per cui si parte dall’indirizzo dell’oggetto classe (contenuto in *this*) e sommiamo l’offset del campo v1. Dunque mettiamo this(%rbp) dentro un registro a 64 bit, ad esempio %r8, e facendo   
v1(%r8, …, …) partiamo dall’indirizzo dell’inizio dell’array. La scala è 1, poiché è un array di char, e dunque facendo v1(%r8, %rcx, 1) indirizziamo l’elemento i-esimo dell’array. A quel punto possiamo fare il primo assegnamento.

Per il secondo assegnamento, v2[i] lo indirizziamo allo stesso modo, con la differenza che questa volta v2[] è un array di long, quindi ogni elemento è a 8 byte (e quindi si usa   
scala = 8). Dobbiamo assegnargli ss.v[i] \* 2. ss.v[i] sta in %eax, allora quello che facciamo è prima estenderlo con segno, portandolo su un registro a 8 byte, ad esempio rax; fare la moltiplicazione per 2 e poi assegnare il valore a v2[i]. Il motivo dell’estensione è che dobbiamo poi mettere un valore in v2(%r8, %rcx, 8), e ricordiamo che la dimensione dell’operando da mettere è vincolata dal registro sorgente che si usa. Poiché dobbiamo mettere 8 byte dentro v2[i] (è un array di long), necessariamente va usato un registro a 8 byte, e quindi va fatta l’estensione con segno (*con segno* perché ss.v[i] che mettiamo dentro eax può essere anche negativo).

Per il terzo assegnamento, v3[i] lo indirizziamo allo stesso modo, con la differenza che questa volta v3[] è un array di int, quindi ogni elemento è a 4 byte (e quindi si usa scala = 4). Dobbiamo assegnargli ss.v[i] \* 4. In eax abbiamo ss.v[i], quindi quello che facciamo è moltiplicare il contenuto di eax per quattro e assegnare il valore a v3[i].

Dopo va tradotto lo step.

Usciti dal for, il corpo della funzione è finito, e quindi c’è l’epilogo. Non ho usato alcun registro preservato (a parte RBP), quindi concludo facendo leave e ret.

Traduzione elab1.cpp

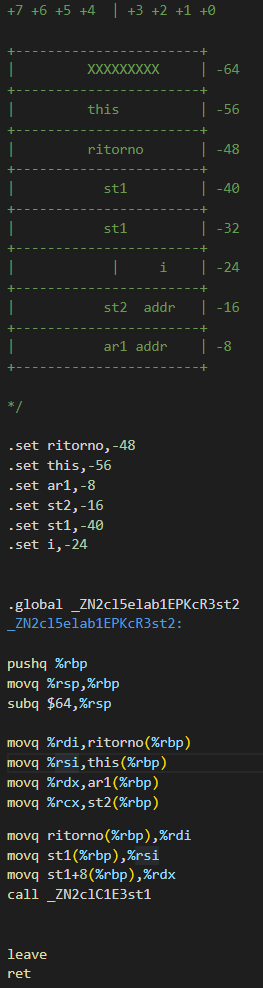
E’ una funzione membro, e seguendo le regole di traduzione, si ottiene quel nome. La rendo poi globale, così che il collegatore possa vederla, e poi faccio “\_ZN2cl5elab1EPKcR3st2: ”

Si inizia col prologo, creando il record di attivazione della funzione. Salvo quindi in pila il valore di RBP, metto il valore attuale di RSP in RBP e devo vedere quanto spazio allocare per variabili locali e parametri d’ingresso. **Novità**: **questa funzione restituisce per valore un oggetto classe che ha dimensione >16 byte, dunque come primo parametro le verrà passato l’indirizzo in cui restituire l’oggetto. Essendo poi una funzione membro, come secondo parametro c’è anche il puntatore *this***, **e cioè il puntatore all’oggetto classe su cui si chiama la funzione membro.**  Facciamo allora lo schemino dello stack e decidiamo quale spazio riservare ad un oggetto di tipo st1, un intero, l’indirizzo di un oggetto di tipo st2 passato per riferimento, l’indirizzo contenuto nel puntatore ar1, l’indirizzo dell’oggetto classe su cui si chiama la funzione membro e l’indirizzo in cui restituire l’oggetto classe. Ci sarebbe anche da riservare spazio per *cla*, ma visto che noi dobbiamo restituire proprio *cla* all’indirizzo in cui restituire l’oggetto classe, un’ottimizzazione che si può fare è non riservare spazio per cla, scrivendo direttamente all’indirizzo in cui restituire l’oggetto classe. Deciso allora dove mettere tutto, vediamo che servono 7 righe, quindi 56 byte, RSP però deve essere allineato a 64, e quindi facciamo almeno sub %64, %rsp. Fatto questo, definiamo etichette con lo stesso nome dei parametri d’ingresso/variabili locali cui assegniamo il valore del loro offset rispetto a dove punta rbp.

Copiamo poi i parametri d’ingresso nella loro zona riservata

Traduciamo ora il corpo della funzione. Dichiariamo un oggetto classe *cla* facendo cla(s1), dunque va chiamato il costruttore. Come primo parametro, e cioè come *this,* andrebbe passato l’indirizzo di *cla*, ma visto che per ottimizzazione abbiamo detto che scriviamo direttamente all’indirizzo in cui restituire l’oggetto classe, passiamo proprio tale indirizzo. Dobbiamo poi passare s1, e poiché occupa più di 8 byte, i primi 8 byte li passiamo in rsi, e i successivi 8 byte in rdi. Fatto questo, chiamiamo effettivamente il costruttore.

Finito il corpo della funzione, c’è l’epilogo. Non ho usato alcun registro preservato (a parte RBP), quindi concludo facendo leave e ret.



Traduciamo ora il corpo della funzione.

Traduciamo il for. Tradotta l’inizializzazione e il controllo della condizione, c’è il corpo del for.

Vediamo il primo assegnamento. Dobbiamo indirizzare elementi di un array, quindi usiamo l’indirizzamento indiretto *(base, indice, scala).* Innanzitutto, mettiamo l’indice *i* esteso con segno dentro RCX. ss si trova nello stack ed è di tipo st1. Il tipo st1 contiene solo l’array vi[], per cui l’indirizzo iniziale dell’array ss.vi[] si ricava partendo da st1(%rbp, …, …). La scala è 4, perché è un array di int, e quindi st1(%rbp, %rcx, 4) permette di indirizzare l’elemento i-esimo dell’array. Mettiamolo dentro un registro a 4 byte, es. eax.

Questo dobbiamo metterlo dentro v1[i], che è un elemento di un array di char, dunque dobbiamo mettere solo il byte meno significativo (quindi %al). Per indirizzare v1[i], questo è un campo dell’oggetto classe su cui viene chiamata la funzione membro, per cui si parte dall’indirizzo dell’oggetto classe (contenuto in *this*) e sommiamo l’offset del campo v1. Dunque mettiamo this(%rbp) dentro un registro a 64 bit, ad esempio %r8, e facendo   
v1(%r8, …, …) partiamo dall’indirizzo dell’inizio dell’array. La scala è 1, poiché è un array di char, e dunque facendo v1(%r8, %rcx, 1) indirizziamo l’elemento i-esimo dell’array. A quel punto possiamo fare il primo assegnamento.

Per il secondo assegnamento, v2[i] lo indirizziamo allo stesso modo, con la differenza che questa volta v2[] è un array di long, quindi ogni elemento è a 8 byte (e quindi si usa   
scala = 8). Dobbiamo assegnargli ss.v[i] \* 2. ss.v[i] sta in %eax, allora quello che facciamo è prima estenderlo con segno, portandolo su un registro a 8 byte, ad esempio rax; fare la moltiplicazione per 2 e poi assegnare il valore a v2[i]. Il motivo dell’estensione è che dobbiamo poi mettere un valore in v2(%r8, %rcx, 8), e ricordiamo che la dimensione dell’operando da mettere è vincolata dal registro sorgente che si usa. Poiché dobbiamo mettere 8 byte dentro v2[i] (è un array di long), necessariamente va usato un registro a 8 byte, e quindi va fatta l’estensione con segno (*con segno* perché ss.v[i] che mettiamo dentro eax può essere anche negativo).

Per il terzo assegnamento, v3[i] lo indirizziamo allo stesso modo, con la differenza che questa volta v3[] è un array di int, quindi ogni elemento è a 4 byte (e quindi si usa scala = 4). Dobbiamo assegnargli ss.v[i] \* 4. In eax abbiamo ss.v[i], quindi quello che facciamo è moltiplicare il contenuto di eax per quattro e assegnare il valore a v3[i].

Dopo va tradotto lo step.

Usciti dal for, la funzione finisce, e quindi c’è l’epilogo. Non ho usato alcun registro preservato (a parte RBP), quindi concludo facendo leave e ret.

**BRAINSTORM**

* **IMPORTANTE: ragiona su quando mettere la lea o la mov:** 
  + La lea la usi quando in %rdi devi mettere proprio l’indirizzo di una parte dello stack;
  + La mov la usi quando devi mettere in %rdi l’indirizzo contenuto in una riga dello stack.
* MOV**S**BW, MOVSBL… fanno estensione di segno; (b, w, l, q)
* MOV**Z**BW, MOV**Z**BL… fanno estensione senza considerare il segno (mettono sequenze di zeri).

Per i char va usato MOVSBW, MOVSBL… in quanto sono considerati come “signed char”, cioè come numeri con segno. Dunque è corretto (e si deve fare) estenderli usando MOV**S**… e cioè estendendoli con segno. Penso che magari non devi fare l’estensione con segno se devi estendere un unsigned, perché altrimenti uscirebbe un casino, ma finora non l’ho mai trovato agli esami. Ed estendili, non fare tipo la xor per azzerare il registro, pensando che così su tutto il registro hai il numero, perché tu lavori con numeri interi, e quindi per estenderlo su tutto il registro devi fare l’estensione.

* Da RL, che sia un numero naturale o intero, la riduzione di campo si fa prendendo semplicemente gli n bit meno significativi, a patto che sia rappresentabile su n bit. Quindi effettivamente se hai un numero intero su 32 bit e devi metterlo da una parte a 8, semplicemente prendi gli 8 bit meno significativi, perché tanto se è errato era perché a prescindere non si poteva mettere su 8 bit, non è un problema tuo.
* Attento anche a quando fai i controlli per i salti: falli al contrario, che è più semplice, ma ragiona, che spesso ti confondi e inverti le robe da confrontare.
* Metti l’indice dentro RCX dentro il corpo del for.
* Ricorda che:
  + per indirizzare cose nello stack si fa etichetta(%rbp).
  + per indirizzare robe di array, usi (base, indice, scala), con base =indirizzo iniziale array, indice l’indice e scala dimensione di un elemento dell’array.
* Il costruttore di copia ha un solo parametro. Se c’è più di un parametro, sicuro non è un costruttore di copia.
* Le “.set” vanno scritte dentro il corpo della funzione, così da non avere errori.
* Se hai nello stack un oggetto classe che es, occupa 2 righe e metà della terza, comunque su quella metà non puoi metterci altro, perché comunque è considerata parte dell’oggetto classe (la dimensione di un oggetto classe è il più piccolo multiplo dell’allineamento dell’oggetto tale da contenere tutto l’oggetto stesso, e quindi anche la terza riga è considerata parte dell’oggetto).
* Quando devi sommare/sottrarre due cose che hanno dimensione diversa, fai bene ad estendere quello che dimensione minore a dimensione maggiore e poi fare l’operazione.
* Stai attento al tipo di ritorno, nel caso devi ritornare qualcosa di maggiore di 16 byte.