23. I tipi derivati
Dai tipi fondamentali, attraverso vari meccanismi, si possono derivare tipi più complessi
I principali costrutti per costruire tipi derivati sono: i riferimenti, i puntatori, gli array, le strutture, le unioni e le classi.

24. La struttura in memoria di un programma
Un programma presenta quattro parti fondamentali.

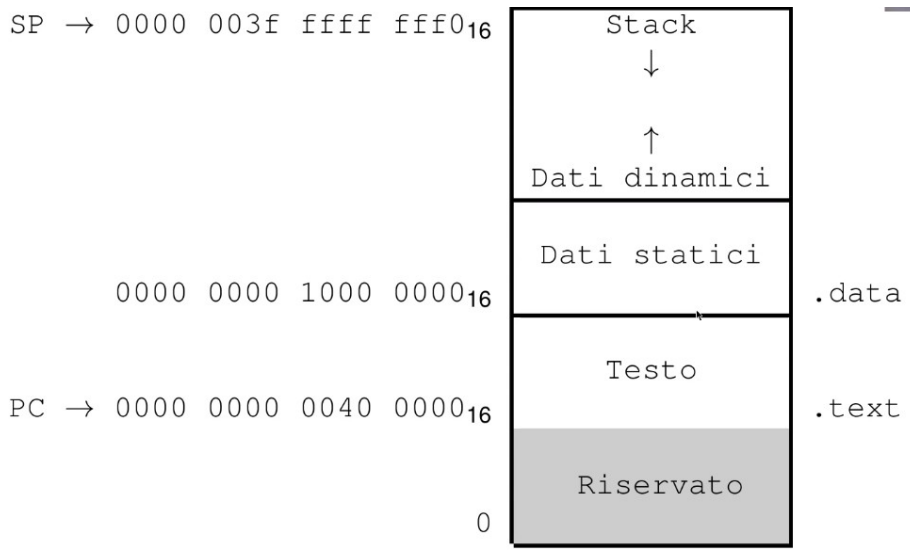
1. Una parte riservata, che contiene istruzioni macchina riservate al sistema operative (istruzioni che permettono di far partire il boot, eseguire il programma, ecc.). Non si ha il diretto controllo di questa parte, a meno che non si programmino parti del kernel o non si scriva un programma in assembler.
2. Area di testo: dove vengono inserite le istruzioni macchina (le istruzioni assembly) del nostro programma. A partire dal *main* e tutto quello che da esso viene chiamato viene inserito qua. Viene creata dal compilatore quando lo chiamiamo per creare l'eseguibile.
3. Componente di dati statici: contiene informazioni relative alle costanti e alle variabili dichiarate nel nostro programma.
4. Area dati dinamica: contiene informazioni dinamiche necessarie all'esecuzione del programma (funzioni, locazioni dinamica di memoria).

Ad esempio se abbiamo un programmino molto semplice con un main che dichiara due variabili x e y, cosa succede? All'interno dell'area dati dinamica verrà prevista dell'area di memoria che conterrà queste variabili.
Tipicamente si usa una rappresentazione virtuale della memoria (indirizzi virtualizzati che partono dall'indirizzo 0 e via via crescono). Processore legge istruzioni nell'area di testo, viene eseguita e viene incrementato o modificato il programm counter (va all'istruzione successiva). Quando sarà necessario ricevere nuova memoria sarà allocata sullo stack.

Come sono rappresentati i programmi sui diversi sistemi operativi?

- Su Windows l'area dati stack ha una dimensione finita ed è hardcoded nel programma. È specificata in compilation time.
- Sotto Unix dipende dal tipo di architettura.
  - Linux: l'area è dinamica (si può modificare a command line senza ricompilare l'eseguibile)
  - MacOS: è hard coded. Può essere modificata con comandi esterni.

Il fatto che questa dimensione è hardocoded è importante nelle funzioni ricorsive: se le chiamate ricorsive sono eccessive si satura la memoria di stack e il programma abortisce.



23.1 Il tipo “Riferimento a”
Il meccanismo dei riferimenti (reference) consente di dare nomi multipli a una variabile (o a un’espressione dotata di indirizzo). In questo modo si definiscono nomi diversi per riferirsi alla stessa area di memoria. Si creano sostanzialmente dei sinonimi per le variabili.

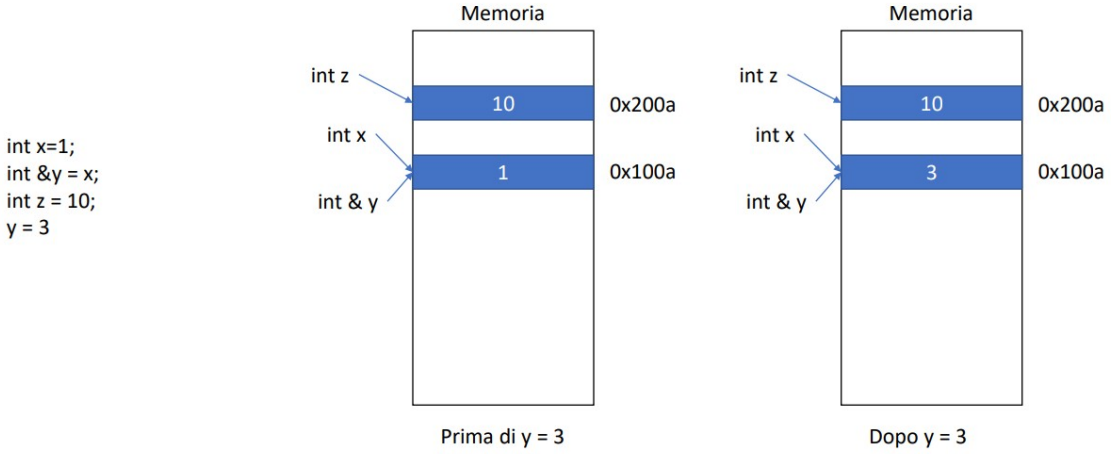
Modificando un riferimento, si modifica anche la variabile a cui si riferisce (“aliasing”). Un riferimento è un’espressione dotata di indirizzo.

Sintassi:
tipo & id = exp; //dove *exp* è un’espressione dotata di indirizzo e *tipo* è il tipo della variabile *exp*
Esempio:
int x=1;
int &y=x; // y e‘ di tipo reference, e‘ sinonimo di x
y = 6; // viene modificato anche x! Infatti y=6 e x=6.

***!* y e *x* hanno lo stesso r-value e lo stesso l-value**

Dopo l’operazione di definizione *y* può venire utilizzata come una semplice variabile.

La memoria si può rappresentare come un rettangolo diviso in rettangoli trasversali che sarebbero le celle di memoria della variabili. Le celle hanno un dato indirizzo (l’ l-value della variabile) e ciò che è memorizzato dentro è l’ r-value della variabile. Come si può vedere *y* e *x* si riferiscono alla stessa cella di memoria, di l-value 0x200a.



23.1.2 Vincoli sull’uso dei Riferimenti
- Nelle dichiarazioni di reference, l’inizializzazione è obbligatoria!
  int &y; // errore!
- Non è possibile ridefinire una variabile di tipo riferimento precedentemente definita*:*
  double x1, x2;
  double &y=x1; // ok
  double &y=x2; // errore! gia’ definita!
- Non è (più) possibile definire un riferimento a un’espressione non dotata di indirizzo, o a un espressione dotata di indirizzo ma di tipo diverso.
  float &y=10.2; // Errore. Tentativo di reference su una espressione senza l-value
  double d=3.1; // Ok
  int &z=d; // Errore. Tentativo di reference su una espressione di tipo diverso.

23.1.3 L’Operatore address-of “&”

L’operatore & (“address-of”) ritorna l’indirizzo (l-value) dell’espressione a cui è applicato.
Può essere applicato solo a espressioni dotate di indirizzo!
È differente dall’uso di “&” nella definizione di riferimenti o dall’”&” usato nell’AND bit-a-bit o come congiunzione logica.

```
int l = 10;
cout << &l << endl; // stampa l’indirizzo di l
cout << &(l*5) << endl; // errore!
int n = 10; int& r = n; // r e’ alias di n, ‘‘punta’’ stessa area di memoria
cout << "&n = " << &n << ", &r = " << &r << endl; //&n=&r e anche r=n;
```

23.2 Il tipo “Puntatore a”

Un puntatore contiene l’indirizzo di un altro oggetto: **l’r-value di un puntatore è un indirizzo** (è l’l-value dell’espressione a cui si riferisce).

Definizione di un puntatore:
Sintassi: *tipo *id_or_init;*//tipo deve essere il tipo dell’espressione a cui il puntatore si riferisce.
Esempio: *int *px;* //px puntatore a un intero.
È sempre necessario indicare il tipo di oggetto a cui punta.
! Un puntatore a tipo T può contenere solo indirizzi di oggetti di tipo T.

Allocazione di memoria al puntatore
Ad una variabile puntatore viene associata una spazio di memoria atto a contenere un indirizzo di memoria, ma non viene riservato spazio di memoria per l’oggetto puntato (il quale dovrebbe avere già uno spazio in memoria in quanto deve essere definito prima).

Lo spazio allocato a una variabile di tipo puntatore è sempre uguale, indipendentemente dal tipo dell’oggetto puntato.
*sizeof(int *) == sizeof(long double *) == sizeof(char *) == sizeof(T *)* per ogni tipo (base o derivato) T .

23.2.1 L’Operatore di Dereference “*”

Per accedere all’oggetto puntato da una variabile puntatore occorre applicare l’operatore di dereference *****.
Se *px* punta a *x*, **px* è un sinonimo temporaneo di *x.* Modificando **px* modifico *x*, e viceversa.

Grazie all’operatore dereference ***** preceduto dall’identificatore del mio puntatore riesco ad accedere all’area di memoria della variabile, cioè all’r-value della variabile, e a modificarlo a mio piacimento.

! anche **px* è un’espressione dotata di indirizzo **!**

Esempio
int x=1; // *x* variabile tipo int
*int *px;* // *px* variabile puntatore a tipo int.
px=&x; // accede alla variabile puntatore. r-value di *px* è l-value di *x.*
**px=x+1;* // accede alla cella di memoria puntata dalla variabile puntatore. **px=2* e anche *x=2*