IDENTIFICATORI E CLASSI

Funzioni globali C++

- **Funzioni:**
  - **secondo le regole del compilatore C++, l'identificatore Assembler di una funzione globale (di livello più esterno) dipende dalla firma (*signature*) della funzione C++ stessa.**
- **Definizione di una funzioni globale C++:**

  type-r fun-id (type1 par1, type2 par2, …, typeh parh) { … }
  - ***type-r*: tipo del risultato;**
  - ***fun-id:* identificatore C++ della funzione;**
  - ***type1*, *type2*, …, *typeh* : tipi dei parametri formali;**
  - ***par1*, *par2*, …, *parh* gli identificatori dei parametri formali.**
- **Dichiarazione di una funzioni globale C++:**
  - **la dichiarazione, oltre all'assenza del corpo, non richiede il nome dei parametri formali.**
- **Firma di una funzione globale:**
  - **costituita dalla dichiarazione senza la specifica del tipo del risultato.**

firma di funzione e tipi dei parametri

- **Tipi di parametri formali C++:**
  - ***tipi semplici predefiniti,* ciascuno costituito da un identificatore predefinito (identificatori dei vari tipi interi e dei vari tipi reali, identificatori *char* e *bool*);**
  - ***tipi semplici utente*, ciascuno costituito da un identificatore dichiarato dall'utente (identificatori di un tipo enumerazione, di un tipo struttura o unione, di un tipo classe);**
  - ***tipi composti*, ciascuno costituito da un tipo semplice, predefinito o utente, e da uno o più operatori scelti fra *, & e [ ], con l'eventuale utilizzo dell'attributo *const.***

Identificatori Assembler di funzioni globali C++

- **Identificatore Assembler di una funzione globale C++:**

  _Z*C*fun-idT1T2…Th

  - **inizia con un segno di sottolineatura (*underscore*) e con la lettera predefinita Z;**
  - **è seguito da *h+1 sottoidentificatori*.**
- **Primo sottoidentificatore Assembler (*Cfun-id*):**
  - **costituito dal numero *C* di caratteri di cui è composto l'identificatore C++ della funzione, seguito dall'identificatore C++ stesso.**
- **Altri sottoidentificatori Assembler (*T1*, *T2*, …, *Th*):**
  - **costituiti dalle notazioni Assembler utilizzate per rappresentare, rispettivamente, il tipo del primo, del secondo, …, dell'ultimo parametro formale della funzione C++;**
    - **nel caso dei tipi predefiniti, si hanno delle regole indipendenti dal fatto che il tipo appartenga alla categoria degli interi o a quella dei reali.**

Tipi semplici predefiniti C++

- **Tipi semplici predefiniti:**
  - **i sottoidentificatori Assembler sono costituiti da singoli caratteri (spesso coincidono con il primo carattere del tipo C++).**

tipo C++	*Sottoidentificatore Assembler*
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**
…	

- **Esempio:**

  funz(char);

  - **identificatore Assembler:**

    _Z4funzc

Esempi con tipi semplici predefiniti

- **Funzione C++** **Identificatore Assembler**

  funz(); **_Z4funzv**
  funz(int); **_Z4funzi**
  funz(long int); **_Z4funzl**
  funz(short int); **_Z4funzs**
  funz(int, int); **_Z4funzii**

- **Attributo *const* per tipi semplici predefiniti:**
  - **non ha rilevanza in Assembler:**
    funz(const int); **_Z4funzi**

Tipi semplici utente (1)

- **Tipo semplice utente (enumerato, struttura o unione, classe):**
  - **identificatore C++:**
    type-id
  - **sottoidentificatore Assembler:**
    Ctype-id
    dove *C* indica il numero di caratteri del tipo semplice utente.
- **Esempi di dichiarazioni di tipo:**

```
enum enumerazione {    };
struct struttura {    };
union unione {    };
class classe {    };
```

- **Esempi di dichiarazioni di funzione (con tipi semplici predefiniti e utente):**

```
funz(int, unione, classe);
funz(enumerazione, int , struttura);
```

- **Corrispondenti identificatori Assembler:**

```
_Z4funzi6unione6classe
_Z4funz12enumerazionei9struttura
```

Tipi semplici utente (2)

- **Indicazione, nel sottoidentificatore Assembler, del numero di caratteri del tipo semplice utente:**
  - **consente di effettuare discriminazioni.**
- **Esempi:**
  - **dichiarazioni di tipo:**
    struct struttura { };
    struct strutturai { };
  - **Firme di funzioni:**
    funz(struttura, int);
    funz(strutturai);
  - **identificatori Assembler delle 2 funzioni precedent:**
    _Z4funz9strutturai
    _Z4funz10strutturai
- **Attributo *const* per tipi semplici utente non ha rilevanza in Assembler:**
  - **firma:** **funz(const struttura);**
  - **identificatore Assembler:** **_Z4funz9struttura**

Tipi composti con tipi semplici predefiniti: i puntatori (1)

- **Tipo composto costituito da un puntatore (a un tipo semplice predefinito *t*):**
  - **sottoidentificatore Assembler:**

    Pt

    P è un carattere predefinito per i puntatori e *t* specifica in Assembler il tipo (predefinito) della variabile puntata.
  - **esempio (puntatore al tipo *int*):**

    in C++ *int** **sottoidentificatore Assembler: *Pi***
- **Entità puntata:**
  - **può possedere l'attributo *const* (l'entità puntata non può essere modificata attraverso il puntatore);**
  - **nel sottoidentificatore Assembler, la notazione relativa al tipo predefinito puntato è preceduta dalla lettera predefinita K:**

    PKt
  - **esempio (puntatore al tipo *const int*):**

    in C++ *const int** **sottoidentificatore Assembler: *PKi***
- **In Assembler, non ha rilevanza che il puntatore sia costante (*int* const*).**

Tipi compositi con tipi semplici predefiniti: i puntatori (2)

- **Esempio di firma di funzione:**

  funz(bool*, const char*, const int* const);

- **Identificatore Assembler (nel terzo parametro, l'attributo *const* dopo *int** non ha rilevanza):**

  _Z4funzPbPKcPKi

- **Tipo dell'entità puntata:**
  - **può anche essere a sua volta un puntatore, eventualmente con l'attributo *const*, come nel seguente esempio:**

    funz(char, const bool**); // funz((char*) *, (const bool*) *);**
  - **identificatore Assembler :**

    _Z4funzPPcPPKb

    tipo del secondo parametro: puntatore a al tipo puntatore a un *bool* con l'attributo *const* (const è il bool).

Tipi composti con tipi semplici predefiniti: i riferimenti

- **Tipo composto costituito da un riferimento (di un tipo semplice predefinito *t*):**

  sottoidentificatore Assembler:

  Rt

  R è un carattere predefinito per i riferimenti e *t* è la notazione Assembler che indica il tipo della variabile riferita;

  - **esempio (riferimento del tipo intero):**

    in C++ *int&* **sottoidentificatore Assembler: *Ri***

- **Entità riferita:**
  - **può possedere l'attributo *const* (per indicare che l'entità stessa non può essere modificata attraverso il riferimento);**
  - **sottoidentificatore Assembler:**

    RKt
  - **esempio:**

    in C++ *const int*& **sottoidentificatore Assembler: *RKi***

- **Esempio di firma di funzione:**

  funz(int&, const char&);

  identificatore Assembler :

  _Z4funzRiRKc

- **In C++:**
  - **i riferimenti per definizione sono costanti (i puntatori non lo sono) e non si possono avere riferimenti di riferimenti.**

Tipi compositi con tipi semplici predefiniti: riferimenti di puntatori

- **Riferimenti di puntatori:**
  - **esempio:**

    int*& **sottoidentificatore Assembler:** ***RPi***
- **Utilizzo dell'attributo *const*:**
  - **per l'entità puntata;**
  - **per il puntatore riferito;**
  - **per entrambi.**
  - **esempi**

    const int*& ***const*** **si riferisce all'intero puntato**

    int* const & ***const*** **si riferisce al puntatore**

    const int*const & ***const*** **(il secondo) per il puntatore e** ***const*** **(il primo) per l'intero puntato**
  - **sottoidentificatori:**

    RPKi

    RKPi

    RKPKi **riferimento di un puntatore** ***const*** **a un intero** ***const***

Tipi composti con tipi semplici predefiniti: gli array

- **Array monodimensionale (elementi di un tipo semplice *t* predefinito):**
  - **il tipo del parametro formale è un puntatore al primo elemento;**
  - **sottoidentificatore Assembler:**

    Pt

    P è il carattere predefinito per i puntatori e *t* è la notazione Assembler che indica il tipo degli elementi dell'array.
  - **esempio:**

    in C++ *int ar[ ]* **sottoidentificatore Assembler: *Pi***
- **Array monodimensionale definito costante:**
  - **la specifica del tipo degli elementi è preceduta dalla lettera K.**
  - **esempio:**

    inC++ *const int ar[ ]* **sottoidentificatore Assembler: *PKi***
- **Nota:**
  - **la definizione di un array costante equivale a definire costanti gli elementi puntati (il primo elemento, e in virtù dell'aritmetica degli indirizzi, tutti gli altri);**
    - **ricordare che *ar[i]* equivale a: **(ar+i)*.**

Tipi composti con tipi semplici predefiniti: gli array multidimensionali (1)

- **Array multidimensionale:**
  - **array monodimensionale (primario) di array monodimensionali (secondari) di array …**
- **Array primario:**
  - **il tipo è un puntatore al primo elemento.**
- **Array secondari, …:**
  - **il tipo deve essere completamente specificato, in tipo e numero dei suoi elementi.**
- **Array multidimensionale a *N* dimensioni:**
  - **sottoidentificatore Assembler:**
    PAnum-2_Anum-3_…_Anum-N_t
  - **P rappresenta un puntatore ad una variabile di tipo array a *N-1* dimensioni, A ed _ sono caratteri predefiniti;**
  - ***num-2, num-3, …, num-N* indicano, rispettivamente, la lunghezza della seconda, della terza, …, della *N*-esima dimensione dell'array;**
  - ***t* è la notazione Assembler che indica il tipo degli elementi dell'array stesso.**
- **Esempio:**
  - **parametro array di interi a 3 dimensioni:**
    in C++ *int a[ ][9][10]*
  - **sottoidentificatore Assembler:**
    PA9_A10_i

Tipi composti con tipi semplici predefiniti: gli array multidimensionali (2)

- **Array multidimensionale definito costante:**
  - **la specifica del tipo degli elementi è preceduta dalla lettera K.**
  - **esempio:**

    const int a[ ][9][10]
  - **sottoidentificatore Assembler:**

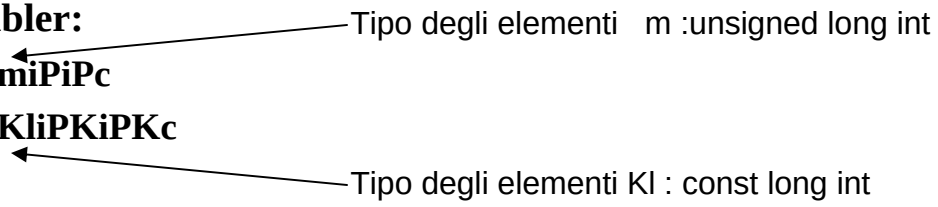
    PA9_A10_Ki
- **Esempi completi:**
  - **firme di funzioni C++:**

    funz(unsigned long int[ ][9][10], int, int[ ], char[ ]);

    funz(const long int[ ][9][10], int, const int[ ],const char[ ]);
  - **identificatori Assembler:**

    _Z4funzPA9_A10_miPiPc

    _Z4funzPA9_A10_KliPKiPKc



Tipi composti con tipi semplici definiti dall'utente

- **Tipo semplice utente:**
  - **preceduto dal numero di caratteri che costituiscono l'identificatore C++ del tipo semplice stesso:**
- **Regole di composizione:**
  - **le stesse viste per i tipi semplici predefiniti.**
- **Esempi:**
  - **dichiarazioni di tipo:**

    enum enumerazione { };
    struct struttura { };
    union unione { };
    class classe { };
  - **firme di funzioni:**

    funz(struttura*, int, char, classe**);**
    funz(struttura&, int&, char*&, classe*&);
    funz(const int*, classe&, const struttura* const &, enumerazione*);
- **Identificatori Assembler delle funzioni:**

  _Z4funzP9strutturaiPPcPP6classe
  _Z4funzR9strutturaRiRPcRP6classe
  _Z4funzPKiR6classeRKPK9strutturaP12enumerazione

Tipi lunghi simili

- **Tipo lungo:**
  - **tipo composto con un tipo semplice predefinito;**
  - **tipo semplice utente;**
  - **tipo composto con un tipo semplice utente;**
- **Tipo base di un tipo lungo:**
  - **in ogni caso, un tipo lungo contiene un tipo semplice, predefinito o utente (in questo caso si può avere coincidenza), detto *tipo base*.**
- **Tipi lunghi con lo stesso tipo base:**
  - **detti tipi lunghi simili;**
    - **un tipo lungo è sempre simile a se stesso.**
- **Esempi di tipi lunghi simili:**

  int*, int*&, const int*, int*

  struttura, struttura*, struttura&, struttura*&, struttura, struttura&
- **Regole per ricavare i sottoidentificatori dei tipi lunghi fra loro simili:**
  - **in generale, abbastanza complesse;**
  - **per semplicità, esaminiamo solo alcuni casi particolari.**

Tipi lunghi simili con tipo base predefinito: casi particolari (1)

- **Tipo base predefinito, con tipo del primo parametro formale un riferimento, sia *t&*:**
  - **possono esserci altri parametri formali simili con questo, quindi di tipo *t&*:**
    - **indicati col sottoidentificatore Assembler *S_*.**
  - **esempio:**

    firma C++: **funz(int&, int, int&, int&);**

    identificatore Assembler: **_Z4funzRiiS_S_**
  - **non possono esserci altri parametri formali aventi tipi simili.**
- **Tipo base predefinito, con tipo del primo parametro formale un puntatore (o un array), sia *tp**:**
  - **possono esserci altri parametri formali simili con questo, quindi di tipo *t**:**
    - **indicati col sottoidentificatore Assembler *S_*;**
  - **esempio:**

    firma C++: **funz(int*, int, int*, int);**

    identificatore Assembler: **_Z4funzPiiS_i**
  - **non possono esserci altri parametri formali aventi tipi simili.**

Tipi lunghi simili con tipo base definito dall'utente: casi particolari (2)

- **Tipo base definito dall'utente:**
  - **tipo del primo parametro formale costituito da un identificatore utente, sia *type-id*.**
- **Altri parametri formali**
  - **possono esserci altri parametri formali simili con questo, quindi di tipo *type-id*;**
    - **per questi, il sottoidentificatore Assembler è dato da *S_*;**
  - **può esservene al più uno di tip *type-id** oppure *type-id&*;**
  - **per questi, il rispettivo sottoidentificatore Assembler è dato, rispettivamente, da *PS_* oppure *RS_*;**
  - **non possono esservi altri parametri aventi tipi simili.**
- **Esempio:**
  - **dichiarazioniìe di tipo:**

    struct stru { int a; int b; };
  - **firma di funzione:**

    funz(stru, stru*, stru, stru);
  - **identificatore Assembler:**

    _Z4funz4struPS_S_S_

Operatori globali

- **Operatori ridefiniti:**
  - **devono coinvolgere almeno un tipo dichiarato dall'utente;**
  - **operatori globali:**
    - **trattati in modo simile alle funzioni globali.**
- **Regola generale:**
  - **firma di un operatore C++:**

    ***operator op*(type1, type2, …, typeh);**
  - **identificatore Assembler:**

    _Zsymb-opT1T2…Th

    ***dove symb-op* è il simbolo Assembler dell'operatore *op* e le altre notazioni sono quelle viste per le funzioni.;**
  - **tutti i simboli Assembler degli operatori sono di 2 caratteri, ed è assente il numero di caratteri *C*.**
- **Esempio:**
  - **dichiarazione C++:**

    struct ss { };
  - **firma di operatore:**

    ss operator-(ss, int);
  - **identificatore Assembler (*mi* è il simbolo Assembler dell'operatore -):**

    _Zmi2ssi

Simboli per gli alcuni operatori globali C++

Operatori C++	Simbolo Assembler
operator+()	pl
operator-()	mi
operator*()	ml
operator/()	dv
operator++()	pp
operator--()	mm
operator+=()	pL
operator-=()	mI
operator*=()	mL
operator/=()	dV
operator==()	eq
operator!=()	ne

Classi e oggetti classe (2)

- **Funzioni membro (che comprendono eventuali costruttori) e operatori membro di una classe:**
  - **sono dichiarati nella classe;**
  - **possono essere definiti o nella classe o al di fuori della classe.**
  - **nel primo caso questi hanno collegamento interno:**
    - **quando vengono applicati a un oggetto classe, non si è certi che vi sia un meccanismo di chiamata, ma semplicemente l'inclusione del codice della funzione membro nel punto opportuno;**
  - **nel secondo caso questi hanno collegamento esterno:**
    - **i loro identificatori Assembler vengono ricavati in modo similare a quello delle funzioni e operatori globali, con regole che tengono conto anche dell'identificatore C++ della classe a cui appartengono.**
- **Nel seguito:**
  - **considereremo sempre funzioni/operatori membro definiti al di fuori della classe.**

Identificatori delle funzioni membro (1)

- **Classe con funzione membro:**
  - **dichiarazione della classe:**

    class class-id
    { …
    type-r fun-id(type-1, type-2, …, type-h); …
    };
  - **definizione di una funzione membro, fuori della classe:**

    type-r class-id::fun-id(type-1 par-id1, type-2 par-id2, …, type-h par-idh) { }
  - **dove:**
    - ***type-r* indica il tipo dell'oggetto restituito;**
    - ***type-1*, *type-2*, …, *type-h* indicano i tipi dei parametri formali;**
    - ***par-id1*, *par-id2*, …, *par-idh* indicano gli identificatori dei parametri formali;**
    - ***class-id* è l'identificatore della classe e *fun-id* l'identificatore della funzione membro.**
- **Identificatore Assembler della funzione membro:**

  _ZN*L*class-id*M*fun-idET1T2…Th
  - **dove:**
    - **Z, N ed E sono caratteri predefiniti;**
    - ***L* rappresenta il numero di caratteri di *class-id* ed M il numero di caratteri di *fun-id.*:**
    - **T1T2…Th denotano i sottoidentificatori Assembler relativi al tipo del primo, al tipo del secondo, …, al tipo dell'ultimo parametro formale della funzione C++;**
    - **l'identificatore Assembler contiene quindi l'identificatore C++ della classe e la firma C++ della funzione membro**

Identificatori delle funzioni membro (2)

- **Funzione membro costante (non può modificare i campi dato):**

```
class class-id
{   // …
  type-r fun-id(type-1, type-2, …, type-h) const;   …
};
```

  - **identificatore Assembler della funzione:**

    _ZNK*L*class-id*M*fun-idET1T2…Th

- **Regole per determinare i sottoidentificatori Assembler relativi ai tipi dei parametri formali della funzione membro:**
  - **per i tipi, semplici o composti, che non contengono l'identificatore della classe in esame:**
    - **si applicano le regole viste per le funzioni globali.**
  - **per i tipi, semplici o composti, che contengono l'identificatore della classe in esame:**
    - **si applicano regole che si rifanno a quelle dei tipi simili.**

Identificatori delle funzioni membro (3)

- **Funzioni membro:**
  - **come visto, l'identificatore Assembler contiene, come sottoidentificatore, l'identificatore C++ della classe;**
  - **tale identificatore rappresenta, implicitamente, il tipo del primo parametro formale.**
- **Funzioni membro aventi parametri formali espliciti che contengono l'identificatore della classe:**
  - **viene applicata la regola dei tipi simili, analoga a quella delle funzioni globali aventi tipi simili con tipo base definito dall'utente.**
- **Caso particolare:**
  - **funzione membro che ha parametri formali espliciti con le seguenti caratteristiche:**
    - **sono del tipo della classe (anche più di un parametro formale, non necessariamente il primo);**
    - **al più uno è del tipo puntatore alla classe oppure riferimento della classe;**
  - **allora il rispettivo sottoidentificatore Assembler è dato, rispettivamente, da *S_*, *PS_* oppure *RS_*.**

Identificatori delle funzioni membro (3)

- **Esempio:**

```
struct st {  int a; long int b;  };
class punto
{    int ix, iy;
public:
     void fun1(punto);
     void fun2(int, punto, punto);
     void fun3(punto, punto, punto) const;
     void fun4(punto, punto&, int, st, punto);
     void fun5(int, st, punto, punto*);
};
```

In Assembler, gli identificatori delle precedenti funzioni membro sono:

```
_ZN5punto4fun1ES_
_ZN5punto4fun2EiS_S_
_ZNK5punto4fun3ES_S_S_
_ZN5punto4fun4ES_RS_i2stS_
_ZN5punto4fun5Ei2stS_PS_
```

Identificatori dei costruttori e del distruttore (1)

- **Costruttori e distruttore:**

```
class class-id
{  // …
   class-id(type-1, type-2, …, type-h); // costruttore ordinario
   class-id(const class-id&);           // costruttore di copia
   // funzioni membro aventi l'identificatore della classe
   // …
   ~class-id();                         // distruttore

   // funzione membro avente l'identificatore della classe preceduto dal
   // carattere tilde
};
class-id::class-id(type-1 par-id1, type-2 par-id2, …, type-h par-idh) {   }
class-id::class-id(const class-id& class-id) {   }
class-id:: ~class-id() {   }
```

Identificatori dei costruttori e del distruttore (2)

- **Identificatore Assembler di un costruttore (ordinario o di copia):**
  - **in Assembler, il sottoidentificatore di una funzione *costruttore* è dato dai caratteri predefiniti *C1*;**
  - **l'intero identificatore ha pertanto la seguente forma:**

    _ZN*L*class-idC1ET1T2…Th
    - **in particolare, al costruttore di copia corrisponde l'identificatore Assembler:**

    ZN*L*class-idC1ERKS
- **Identificatore Assembler del distruttore:**
  - **in Assembler, il sottoidentificatore della funzione *distruttore* è dato dai caratteri predefiniti *D1*;**
  - **l'intero identificatore ha pertanto la seguente forma:**

    _ZN*L*class-idD1Ev
    - **si ha sempre assenza di parametri formali (simbolo v).**

Identificatori dei costruttori e del distruttore (2)

- **Esempio (classe *punto)*:**

```
class punto
{    int ix, iy;
public:
     punto();
     punto(int, int);
     punto (const punto&);
     ~punto();
     ...
};
```

- **Identificatori Assembler dei costruttori e del distruttore:**

```
_ZN5puntoC1Ev
_ZN5puntoC1Eii
_ZN5puntoC1ERKS_
_ZN5puntoD1Ev
```

Identificatori degli operatori membro (1)

- **Identificatori degli operatori membro:**
  - **trattati in modo simile agli identificatori delle funzioni membro.**
- **Operatore *op* ridefinito nella classe *class-id*:**

  class class-id
  { …
  type-r operator *op*(type-1, type-2, …, type-h); // …
  }
  type-r class-id::operator op(type-1 par-id1, type-2 par-id2, …, type-h par-idh) { }
  - **identificatore Assembler (*symb-op* è il simbolo Assembler che rappresenta *op*):**
    _ZNMclass-id*symb-op*ET1T2…Th
- **Operatori membro costanti:**
  - **viene aggiunto il carattere predefinito *K* dopo il carattere predefinito *N*.**
- **Operatori membro:**
  - **tutti gli operatori globali possono essere ridefiniti come operatori membro (2 carateri);**
  - **alcuni operatori possono essere ridefiniti solo come operatori membro (2 caratteri):**

operatori solo membro	Simboli Assembler
operator=()	aS
operator[]()	ix
operator->()	pt

Identificatori degli operatori membro (2)

- **Esempio:**

```
class vettore
{       …
public:
        vettore(const vettore&);
        vettore operator+(const vettore&);
        vettore operator+(int);
        vettore operator-(int);
        bool operator==(const vettore&) const;
        bool operator!=(const vettore&) const;
};
```

- **Identificatori Assembler:**

```
_ZN7vettoreC1ERKS_
_ZN7vettoreplERKS_
_ZN7vettoreplEi
_ZN7vettoremiEi
_ZNK7vettoreeqERKS_
_ZNK7vettoreneERKS_
```


id della classe

Classi e oggetti classe (1)

- **Dichiarazione di una classe:**
  - **essendo una dichiarazione di tipo, va ripetuta in tutti i file in cui viene utilizzata;**
  - **tale dichiarazione non produce nessun allocazione di memoria.**
- **Definizione di un oggetto classe:**
  - **determina una allocazione di memoria per contenere i campi dato dichiarati nella classe stessa.**
- **Esempio:**

```
class clai                  // dichiarazione della classe
{   int ix, iy;
public:
    clai();  // …
};
clai::clai() {    };        // definizione del costruttore
// …                        // definizione di altre funzioni/operatori membro
                            // o di funzioni/operatori globali.
```

- **Definizione di un oggetto *c* di tipo *clai*:**

```
int main()
{   clai c;
    // …
}
```

  - **in questo caso, l'oggetto *c* è automatico, per cui i campi dato *ix* e *iy* vengono allocati in pila (nel record di attivazione della funzione *main()*), come le altre variabili interne alle funzioni).**

Lunghezza dei parametri e del risultato senza costruttore di copia (1)

- **Memorizzazione di oggetti classe:**
  - **gli oggetti classe, come le variabili degli altri tipi, possono essere statici, automatici, dinamici;**
  - **le regole per la loro memorizzazione riguardano solo i campi dato.**
- **Funzioni membro:**
  - **sono realizzate in copia unica;**
  - **hanno come parametro aggiuntivo (trasmesso in *rdi*) l'indirizzo dell'oggetto classe a cui si applicano;**
  - **per i costruttori, in *rdi* va trasmesso l'indirizzo dove va costruito l'oggetto.**
- **Tipi classe;**
  - **i tipi classe hanno le stesse limitazioni dei tipi struttura;**
  - **caso in cui il numero di byte dei campi dato non sia superiore a 8 oppure a 16:**
    - **possono essere utilizzati sia singoli registri che coppie di registri, per la trasmissione dei parametri di un tipo valore e per la restituzione del risultato di un tipo valore.**

Lunghezza dei parametri e del risultato senza costruttore di copia (2)

- **Campi dato che occupano un numero di byte superiore a 16:**
  - **i parametri e il risultato possono essere di un tipo riferimento (o di un tipo puntatore), e in questo caso vengono utilizzati singoli registri (questi devono essere sufficienti a contenere tutti i parametri);**
  - **il risultato può essere di un tipo valore, e in questo caso viene richiesto un ulteriore primo parametro aggiuntivo che ne specifica l'indirizzo (trasmesso in *rdi*), oltre a quello relativo all'indirizzo dell'oggetto a cui la funzione si applica (trasmesso in *rsi*).**
- **Singoli parametri di un tipo valore che occupano più di 16 byte, o registri per i parametri non sufficienti:**
  - **devono essere trasmessi in pila (non esamineremo questo caso).**

Esempio: programma *classe1* (1)

```
// programma classe1, file main1.cpp
class clai
{    int ix, iy, iz;
public:
     clai();
     clai(int, int, int);
     // ...
     void stampa();
};

int main()
{    clai c1, c2(1, 2, 3);
     c1.stampa(); c2.stampa();
}
```

Esempio: programma *classe1* (2)

```
// programma classe1, file clai1.cpp
#include "servi.cpp"
class clai
{    int ix, iy, iz;
public:
     clai();
     clai(int, int, int);
     // ...
     void stampa();
};
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();
}
```

Esempio: programma *classe1* (3)

```
# programma classe1, file main1.s
.text
# registro per l'indirizzo dell'oggetto: rdi
# registri per i parametri espliciti del secondo costruttore: rsi, rdx, rcx
.global main
.set    c1, -32    # 12 byte allineati
.set    c2, -16
main:
    pushq       %rbp
    movq        %rsp, %rbp
    subq        $32, %rsp

    leaq        c1(%rbp), %rdi          # indirizzo oggetto c1
    call        _ZN4claiC1Ev            # chiamata primo costruttore
    leaq        c2(%rbp), %rdi          # indirizzo oggetto c2
    movl        $1, %esi
    movl        $2, %edx
    movl        $3, %ecx
    call        _ZN4claiC1Eiii          # chiamata secondo costruttore
    leaq        c1(%rbp), %rdi
    call        _ZN4clai6stampaEv
    leaq        c2(%rbp), %rdi
    call        _ZN4clai6stampaEv

    movl        $0, %eax
    leave
    ret
```

Esempio: programma *classe1* (4)

```
# programma classe1, file clai1.s
.include "servi.s"
.text
# registro per l'indirizzo dell'oggetto da costruire: rdi
.global _ZN4claiC1Ev
_ZN4claiC1Ev:                          # primo costruttore
    movl    $0, (%rdi)                 # 0 in ix dell'oggetto
    movl    $0, 4(%rdi)                # 0 in iy dell'oggetto
    movl    $0, 8(%rdi)                # 0 in iz dell'oggetto
    ret

# registro per l'indirizzo dell'oggetto da costruire: rdi
# registro per i parametri espliciti: rsi, rdx, rcx
.global _ZN4claiC1Eiii
_ZN4claiC1Eiii:                        # secondo costruttore
    movl    %esi, (%rdi)               # I par va in ix dell'oggettto
    movl    %edx, 4(%rdi)              # II par va in iy dell'oggetto
    movl    %ecx, 8(%rdi)              # III par va in iz dell'oggetto
    ret
```

Esempio: programma *classe1* (5)

```
# programma classe1, file clai1.s, continua …
.global _ZN4clai6stampaEv
# registro per l'indirizzo dell'oggetto a cui si applca stampa(): rdi
.set    paro, -8                        # indirizzo dell'oggetto
_ZN4clai6stampaEv:
    pushq       %rbp
    movq        %rsp, %rbp
    subq        $8, %rsp

    movq        %rdi, paro(%rbp)        # parametro aggiuntivo in rdi
    movq        paro(%rbp), %rax        # scriviint  puo' distruggere rax
    movl        (%rax), %edi
    call        scriviint
    movq        paro(%rbp), %rax
    movl        4(%rax), %edi
    call        scriviint
    movq        paro(%rbp), %rax
    movl        8(%rax), %edi
    call        scriviint
    call        nuovalinea

    leave
    ret
```

Esempio: programma *classe2* (1)

```
// programma classe2, file main2.cpp
class clai
{     int ix, iy, iz;
public:
      clai();
      clai(int, int, int);
      clai somma(int, int, int);
      clai operator+(clai);
      void stampa();
};
int main()
{     clai c1, c2(1, 2, 3), c3;
      c1 = c2.somma(5, 10, 15); c1.stampa();
      c3 = c1 +  c2; c3.stampa();   // c3 = c1.+(c2)
}
```

Esempio: programma *classe2* (2)

```
// programma classe2, file clai2.cpp
#include "servi.cpp"
class clai
{     int ix, iy, iz;
public:
      clai();
      clai(int, int, int);
      clai somma(int, int, int);
      clai operator+(clai);
      void stampa();
};
clai::clai()
{   ix = 0; iy = 0; iz = 0;
}
clai::clai(int a, int b, int c)
{   ix = a; iy = b; iz = c;
}
```

```
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;
}
clai clai::operator+(clai oo)
{      clai temp;
       temp.ix = ix+oo.ix;
       temp.iy = iy+oo.iy;
       temp.iz = iz+oo.iz;
       return temp;
}
void clai::stampa()
{   scriviint(ix); scriviint(iy); scriviint(iz);
    nuovalinea();
}
```

Esempio: programma *classe2* (3)

```
# programma classe2, file main2.s
.text
.global main
.set  c1, -80          # 12 byte allineati a 16
.set  c2, -64
.set  c3, -48
.set risuso, -32       # variabile di appoggio
.set risupl, -16       # variabile di appoggio
# registro per l'indirizzo dell'oggetto: rdi
# registri per i parametri espliciti: rsi, rdx, rcx
main:
      pushq %rbp
      movq  %rsp, %rbp
      subq  $80, %rsp

      leaq    c1(%rbp), %rdi   # indirizzo oggetto
      call    _ZN4claiC1Ev     # costruttore I
      leaq    c2(%rbp), %rdi   # indirizzo oggetto
      movl    $1, %esi
      movl    $2, %edx
      movl    $3, %ecx
      call    _ZN4claiC1Eiii   # costruttore II
      leaq    c3(%rbp), %rdi   # indirizzo oggetto
      call    _ZN4claiC1Ev     # costruttore I
```

```
leaq    c2(%rbp), %rdi # indirizzo oggetto
movl    $5, %esi
movl    $10, %edx
movl    $15, %ecx
call    _ZN4clai5sommaEiii   # funz. somma
movq    %rax, risuso(%rbp)  # ris. in risuso
movl    %edx, risuso+8(%rbp)
movq    risuso(%rbp), %rax  # c1 = risuso
movq    %rax, c1(%rbp)
movl    risuso+8(%rbp), %eax
movl    %eax, c1+8(%rbp)

leaq    c1(%rbp), %rdi    # indirizzo oggetto
call    _ZN4clai6stampaEv  # stampa c1

leaq    c1(%rbp), %rdi    # indirizzo oggetto
movq    c2(%rbp), %rsi    # valore c2
movl    c2+8(%rbp), %edx # coppia di registri
call    _ZN4claiplES_    # c1.operator+(c2)
```

Esempio: programma *classe2* (4)

```
movq    %rax, risupl(%rbp)     # ris. in risupl
movl    %edx, risupl+8(%rbp)
movq    risupl(%rbp), %rax     # c3 = risupl
movq    %rax, c3(%rbp)
movl    risupl+8(%rbp), %eax
movl    %eax, c3+8(%rbp)

leaq    c3(%rbp), %rdi    # indirizzo oggetto
call    ZN4clai6stampaEv # stampa c3

movl    $0, %eax
leave
ret
```

- **Ottimizzazioni possibili:**
  - risultato della funzione somma(), lasciato nei registri eax-edx: memorizzato direttamente in c1;
  - risultato dell'operatore +(), lasciato nei registri eax-edx: memorizzato direttamente in c3;

```
# programma classe2, file clai2.s
.include "servi.s"
.text

# funzione membro _ZN4claiC1Ev …
# funzione membro _ZN4claiC1Eiii …
# come nell'esempio precedente

.global _ZN4clai5sommaEiii
.set    temp, -40              # var. interna
.set    paro, -24              # ind oggetto
.set    a, -12
.set    b, -8
.set    c, -4
_ZN4clai5sommaEiii:
  pushq  %rbp
  movq   %rsp, %rbp
  subq   $40, %rsp

  movq    %rdi, paro(%rbp)    # ind. oggetto
  movl    %esi, a(%rbp)       # parametro a
  movl    %edx, b(%rbp)       # parametro b
  movl    %ecx, c(%rbp)       # parametro c
```

Esempio: programma *classe2* (5)

```
 # programma classe2, file clai2.s, continua …
  leaq     temp(%rbp), %rdi # costruisce temp
  call      _ZN4claiC1Ev   # costruttore I
  movq     paro(%rbp), %rax # ind. oggetto
  movl     (%rax), %ecx
  addl     a(%rbp), %ecx
  movl     %ecx, temp(%rbp)
  movl     4(%rax), %ecx
  addl     b(%rbp), %ecx
  movl     %ecx, temp+4(%rbp)
  movl     8(%rax), %ecx
  addl     c(%rbp), %ecx
  movl     %ecx, temp+8(%rbp)

  movq     temp(%rbp), %rax   #return temp
  movl     temp+8(%rbp), %edx
  leave
  ret

.global _ZN4claiplES_
.set  temp, -40          # var interna
.set  paro, -24          # ind. oggetto
.set  oo, -16
_ZN4claiplES_:
      pushq %rbp         # prologo
      movq  %rsp, %rbp
      subq   $40, %rsp
```

```
      movq   %rdi, paro(%rbp)  # ind. oggetto
      movq   %rsi, oo(%rbp)     # param. in oo
      movl   %edx, oo+8(%rbp) # param. in oo
      leaq    temp(%rbp), %rdi # costruisce temp
      call     _ZN4claiC1Ev        # costruttore I
      movq    paro(%rbp), %rax # ind. oggetto
      movl    oo(%rbp), %ecx   # valore di oo
      addl    (%rax), %ecx
      movl    %ecx, temp(%rbp)
      movl    oo+4(%rbp), %ecx
      addl    4(%rax), %ecx
      movl    %ecx, temp+4(%rbp)
      movl    oo+8(%rbp), %ecx
      addl    8(%rax), %ecx
      movl    %ecx, temp+8(%rbp)

      movq   temp(%rbp), %rax   # return temp
      movl   temp+8(%rbp), %edx

      leave                              # epilogo
      ret

     # funzione membro _ZN4clai6stampaEv …
     # come nell'esempio precedente
```

Lunghezza dei parametri e del risultato con costruttore di copia (1)

- **Costruttore di copia:**
  - **particolare costruttore avente la forma: *class-id*(*const class-id*&);**
  - **identificatore Assembler: _ZN*Mclass-id*C1ERKS_**
- **Esso viene coinvolto quando:**
  - **un oggetto classe viene inizializzato con un altro oggetto classe:**
    - **notazione C++: *class-id o2(o1*) oppure *class-id o2 = o1*;**
  - **nella chiamata di una funzione, un parametro formale valore del tipo della classe viene inizializzato con un parametro attuale costituito da un oggetto classe;**
    - **il chiamante costruisce una copia del parametro attuale nel suo record di attivazione e trasmette il suo indirizzo al chiamato (tramite un registro);**
  - **il risultato restituito da una funzione è un oggetto classe (istruzione C++ *return*);**
    - **il chiamato costruisce una copia dell'oggetto risultato all'indirizzo che deve essergli stato trasmesso come primo parametro (in RDI) dal chiamante.**

Lunghezza dei parametri e del risultato con costruttore di copia (2)

- **Classe con costruttore di copia:**
  - **ogni parametro valore del tipo della classe e il risultato valore del tipo della classe possono avere lunghezza qualsivoglia (ossia la classe può avere lunghezza qualsivoglia), in quanto:**
    - **nella chiamata viene fatta una copia di ogni parametro attuale e trasmesso il suo indirizzo;**
    - **nella restituzione del risultato viene fatta una copia del risultato stesso all'indirizzo trasmesso dal chiamante;**
  - **notare che quando e definito il costruttore di copia, nella chiamata di una funzione membro che restituisce un risultato valore del tipo della classe, occorre che:**
    - **il chiamante trasmetta al chiamato, in RDI, l'indirizzo dove vuole che sia costruito il risultato, e in RSI l'indirizzo dell'oggetto a cui la funzione membro si applica;**
    - **gli eventuali parametri attuali vengano memorizzati a partire da RDX.**

Lunghezza dei parametri e del risultato con costruttore di copia (3)

- **Copia di un parametro attuale:**
  - **il chiamante costruisce una copia di un parametro attuale di un tipo classe, in quanto il chiamato può alterarne il valore.**
- **Nota:**
  - **il costruttore di copia non viene coinvolto nel caso in cui una funzione abbia un parametro o produca un risultato di un tipo *riferimento di classe*.**
- **Compilatore C++:**
  - **spesso effettua ottimizzazioni, non richiamando sempre il costruttore di copia ridefinito, ma quello predefinito.**
- **Costruttore di copia predefinito:**
  - **corrisponde a un ricopiamento dei campi dato, ogni volta che è richiesta una inizializzazione di un oggetto classe con un altro oggetto classe;**
  - **non viene realizzato come un vero e proprio costruttore.**
- **Completa compatibilità tra file scritti in C++ e corrispondenti file scritti in Assembler:**
  - **occorre inserire nel comando *g*++ l'opzione:**

    ***–fno-elide-constructors*.**

Esempio: programma *classe3* (1)

```
// programma class3, file main3.cpp
class clai
{     int ix, iy, iz;
public:
      clai(int=0, int=0, int=0);
      clai(const clai&);
      clai add(clai);
      void stampa();
};
int main()
{     clai c1(1, 2, 3), c2, c3(c1);        # costr. copia:  si applica a c3 con parametro c1
      c3.stampa();
      c2 = c1.add(c3); c2.stampa();    // totale: utilizzo costr. copia 3 volte
                                       // 1 per costruire c3 da c1
                                       // 1 per costruire una copia del parametro attuale
                                       //       di add() C3, sia oo,  e trasmetterne l'indirizzo
                                       // 1 per costruire il risultato di add() all'indirizzo oo
                                       //       trasmesso dal chiamante main()
}
```

Esempio: programma *classe3* (2)

```
// programma classe3,  file clai3.cpp
#include "servi.cpp"
class clai
{     int ix, iy, iz;
public:
      clai(int=0, int=0, int=0);
      clai(const clai&);
      clai add(clai);
      void stampa();
};
clai::clai(int a, int b, int c)
{     ix = a; iy = b; iz = c;
}
clai::clai(const clai& cc)
{     ix = cc.ix; iy = cc.iy; iz = cc.iz;
}
```

```
clai clai::add(clai oo)
{    oo.ix = ix + oo.ix ;
     oo.iy = iy + oo.iy;
     oo.iz = iz + oo.iz;
     return oo;
}
void clai::clai::stampa()
{   scriviint(ix); scriviint(iy);
    scriviint(iz);
    nuovalinea();
}
```

Esempio: programma *classe3* (3)

```
# programma classe3, file main3.s
.text
.global main
.set  c1, -80              # 12 byte allineati a 16
.set  c2,-64
.set  c3, -48
.set  risu, -32            # risultato di add
.set  oo, -16              # oggetto copia per add
main:
      pushq %rbp
      movq  %rsp, %rbp
      subq   $80, %rsp
      leaq    c1(%rbp), %rdi  # costruisce c1
      movl   $1, %esi
      movl   $2, %edx
      movl   $3, %ecx
      call     _ZN4claiC1Eiii
      leaq    c2(%rbp), %rdi # costruisce c2
      movl   $0, %esi
      movl   $0, %edx
      movl   $0, %ecx
      call     _ZN4claiC1Eiii
      leaq    c3(%rbp), %rdi  # costruisce c3
      leaq    c1(%rbp), %rsi  # parametro &c1
      call     _ZN4claiC1ERKS_ # costr. copia

      leaq   c3(%rbp), %rdi
      call   _ZN4clai6stampaEv

      leaq   oo(%rbp), %rdi       # copia in (oo) (c3)
      leaq   c3(%rbp), %rsi
      call    _ZN4claiC1ERKS_

      leaq    risu(%rbp), %rdi   # ind. risultato add
      leaq    c1(%rbp), %rsi     # ind. oggetto
      leaq    oo(%rbp), %rdx    # ind. par oo copiato

      call    _ZN4clai3addES_  # chiamata di add
                               # lascia il risultato in risu
      movq  risu(%rbp), %rdi   # c2 = risultato
      movq  %rdi, c2(%rbp)
      movl   risu+8(%rbp), %edi
      movl  %edi, c2+8(%rbp)

      leaq    c2(%rbp), %rdi
      call     _ZN4clai6stampaEv

      movl         $0, %eax

      leave
      ret
```

Esempio: programma *classe3* (4)

```
# programma classi3, file clai3.s
.include "servi.s"
.text
# registro per l'indirizzo dell'oggetto: rdi
# registri per i parametri espliciti: rsi, rdx, rcx
.global _ZN4claiC1Eiii
_ZN4claiC1Eiii:                 # costruttore
      movl   %esi, (%rdi)     # ix dell'oggettto
      movl   %edx, 4(%rdi)  # iy dell'oggetto
      movl   %ecx, 8(%rdi)  # iz dell'oggetto
      ret

# registro per l'indirizzo del risultato: rdi
# registro per il riferimento del parametro: rsi
.global _ZN4claiC1ERKS_
_ZN4claiC1ERKS_:          # costruttore di copia
      movl   (%rsi), %eax   # ind parametro in rsi
      movl   %eax, (%rdi)   # ind oggetto in rdi
      movl   4(%rsi), %eax
      movl   %eax, 4(%rdi)
      movl   8(%rsi), %eax
      movl   %eax, 8(%rdi)
      ret
```

```
.global _ZN4clai3addES_    # add(oo copiato)
.set  indris, -24
.set  indo, -16                 # ind oggetto
.set  i_oo, -8                  # ind. par
_ZN4clai3addES_:                # somma
    pushq %rbp
    movq  %rsp, %rbp
    subq   $24, %rsp

    movq %rdi, indris(%rbp) # ind. risultato
    movq %rsi, indo(%rbp)   # ind. oggetto
    movq %rdx, i_oo(%rbp)  # ind. par. oo

    movq indo(%rbp), %rax  # ind. oggetto
    movq i_oo(%rbp), %rdx  # ind par.  oo
    movl  (%rax), %ecx          # ix dell'oggetto
    addl   (%rdx), %ecx         # (i_oo).ix
    movl  %ecx, (%rdx)  # somma in (i_oo).ix
```

Esempio: programma *classe3* (5)

```
# programma classi3, file clai3.s segue …
      movl   4(%rax), %ecx          # iy dell'oggetto
      addl   4(%rdx), %ecx          # (i_oo).iy
      movl   %ecx, 4(%rdx)          # somma in (ioo).iy
      movl   8(%rax), %ecx          # iz dell'oggetto
      addl   8(%rdx), %ecx          # (oo).iz
      movl   %ecx, 8(%rdx)          # somma in (oo).iz
      # oo di main e' stato modificato,
      # c3 di main non lo e' stato

      movq  indris(%rbp), %rdi
      movq  i_oo(%rbp), %rsi
      call     _ZN4claiC1ERKS_     # copia (i_oo) in (indris)
                                   # col costr. di copia

      movq  indris(%rbp), %rax
      leave
      ret

      # funzione membro _ZN4clai6stampaEv …
     # come negli esempi precedenti
```

Ridefinizione dell'operatore di assegnamento (1)

- **Versione predefinita dell'operatore di assegnamento:**
  - **azione collaterale: ricopiamento membro a membro dell'operando a destra (*r-value*) nell'operando a sinistra (*l-value*);**
  - **risultato: l'operando di sinistra (*l-value*);**
  - **tale operatore si applica anche a oggetti classe.**
- **In una classe, può essere ridefinito solo come operatore membro, secondo la seguente dichiarazione:**

  class-id& operator=(const class-id&);
  - **può comportare azioni aggiuntive o azioni differenti rispetto alla versione predefinita;**
  - **identificatore Assembler (il simbolo dell'operatore è *aS*):**

    ZN*Mclass-id*aSERKS
- **In un programma, in caso di assegnamento fra oggetti classe:**
  - **se tale operatore non è stato ridefinito, viene utilizzato l'operatore predefinito;**
  - **se tale operatore è stato ridefinito, viene richiamato quello ridefinito.**

Ridefinizione dell'operatore di assegnamento (1)

- **Esempio di ridefinizione dell'operatore nella classe *clai*:**

```
clai& clai::operator =(const clai& cc)
{  if(this !=&cc)
   {  # azioni aggiuntive o sostitutive;
      # azioni predefinite:
      ix = cc.ix;
      iy = cc.iy;
      iz = cc.iz;
   }
   return *this;
}
```

- **Attenzione:**
  - **nella ridefinizione, non riutilizzare l'operatore che si sta ridefinendo, come nel seguente caso:**

```
clai& clai::operator =(const clai& cc)
{  if(this !=&cc) *this = cc;
   return *this;
}
```