Programmazione 1

Lezione 10

Dipartimento di Matematica *Federigo Enriques*
Università degli Studi di Milano

Allocazione statica *vs.* allocazione dinamica

Tutti i tipi di dati che abbiamo visto finora, inclusi gli array e le strutture, sono soggetti ad allocazione statica della memoria, il che significa quanto segue.

Allocazione statica *vs.* allocazione dinamica

Tutti i tipi di dati che abbiamo visto finora, inclusi gli array e le strutture, sono soggetti ad allocazione statica della memoria, il che significa quanto segue.

1. Quando il programma dichiara una variabile di tipo `T`, il sistema assegna (o alloca) al programma memoria centrale della macchina in quantità pari al numero N di byte necessari a rappresentare un dato di tipo `T`.

Allocazione statica *vs.* allocazione dinamica

Tutti i tipi di dati che abbiamo visto finora, inclusi gli array e le strutture, sono soggetti ad allocazione statica della memoria, il che significa quanto segue.

1. Quando il programma dichiara una variabile di tipo `T`, il sistema assegna (o alloca) al programma memoria centrale della macchina in quantità pari al numero N di byte necessari a rappresentare un dato di tipo `T`.
2. Il numero N è *costante*: esso *non cambia* dal momento in cui la variabile è dichiarata, fino al momento in cui essa cessa di esistere.[1]

[1] Il momento in cui ciò accade dipende dal campo di visibilità della variabile.

Allocazione statica *vs.* allocazione dinamica

Tutti i tipi di dati che abbiamo visto finora, inclusi gli array e le strutture, sono soggetti ad allocazione statica della memoria, il che significa quanto segue.

1. Quando il programma dichiara una variabile di tipo `T`, il sistema assegna (o alloca) al programma memoria centrale della macchina in quantità pari al numero N di byte necessari a rappresentare un dato di tipo `T`.
2. Il numero N è *costante*: esso *non cambia* dal momento in cui la variabile è dichiarata, fino al momento in cui essa cessa di esistere.[1]
3. Non appena la variabile cessa di esistere, gli N byte di memoria sono automaticamente liberati, o deallocati, e ritornano nella disponibilità del sistema operativo.

[1] Il momento in cui ciò accade dipende dal campo di visibilità della variabile.

Allocazione statica *vs.* allocazione dinamica

Il C permette anche al programmatore di gestire l'allocazione della memoria dinamicamente, come segue.

Allocazione statica *vs.* allocazione dinamica

Il C permette anche al programmatore di gestire l'allocazione della memoria dinamicamente, come segue.

1. Il programma può richiedere al sistema, in un suo qualunque punto, l'allocazione di N byte di memoria centrale.

Allocazione statica *vs.* allocazione dinamica

Il C permette anche al programmatore di gestire l'allocazione della memoria dinamicamente, come segue.

1. Il programma può richiedere al sistema, in un suo qualunque punto, l'allocazione di N byte di memoria centrale.
2. Se la richiesta può essere soddisfatta, il sistema assegna N byte al programma, che può usarli per memorizzare dati.

Allocazione statica *vs.* allocazione dinamica

Il C permette anche al programmatore di gestire l'allocazione della memoria dinamicamente, come segue.

1. Il programma può richiedere al sistema, in un suo qualunque punto, l'allocazione di N byte di memoria centrale.
2. Se la richiesta può essere soddisfatta, il sistema assegna N byte al programma, che può usarli per memorizzare dati.
3. Durante la sua esecuzione, il programma può anche chiedere che gli N byte ad esso riservati siano *aumentati* o *diminuiti*: si parla di riallocazione (dinamica) della memoria.

Allocazione statica *vs.* allocazione dinamica

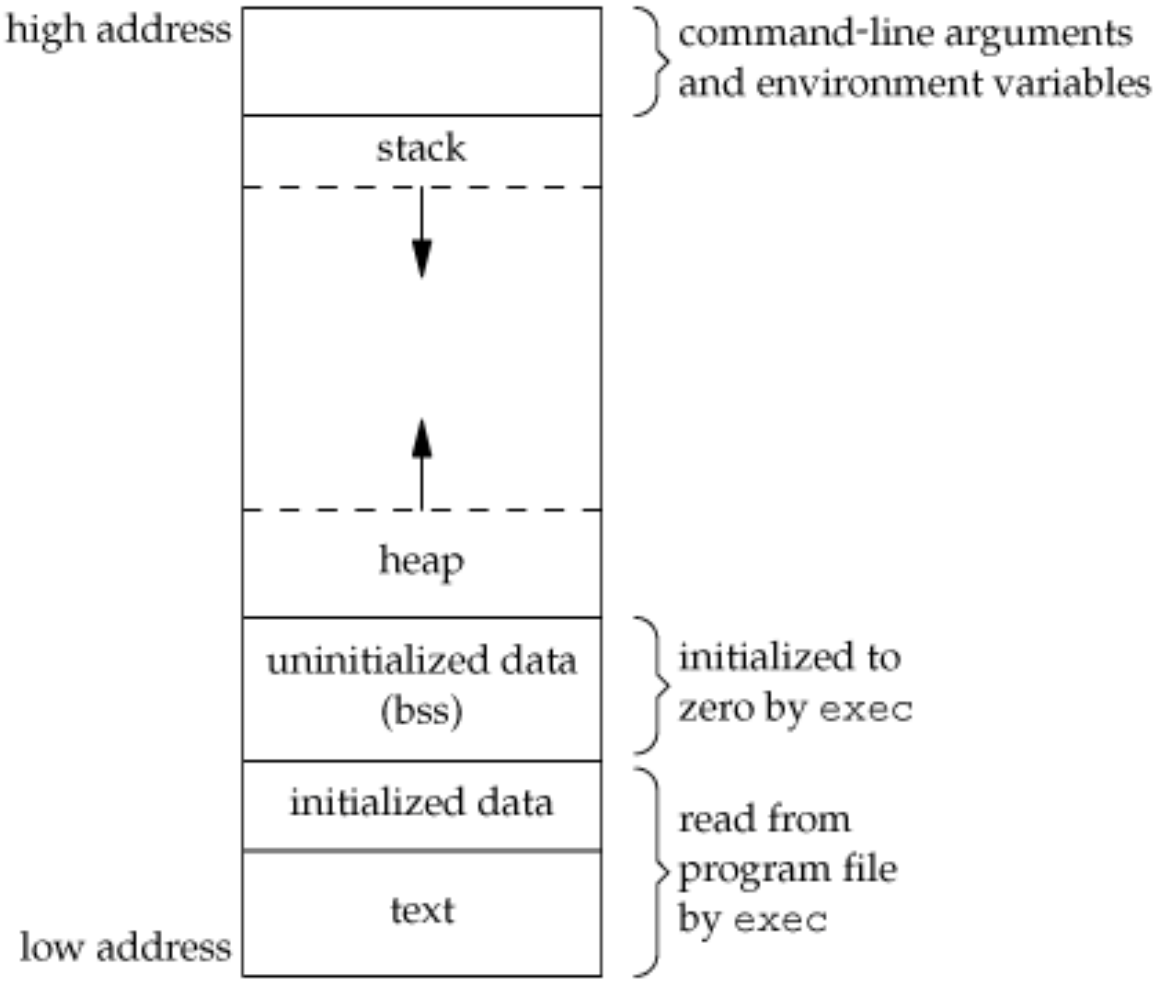
Il C permette anche al programmatore di gestire l'allocazione della memoria dinamicamente, come segue.

1. Il programma può richiedere al sistema, in un suo qualunque punto, l'allocazione di N byte di memoria centrale.
2. Se la richiesta può essere soddisfatta, il sistema assegna N byte al programma, che può usarli per memorizzare dati.
3. Durante la sua esecuzione, il programma può anche chiedere che gli N byte ad esso riservati siano *aumentati* o *diminuiti*: si parla di riallocazione (dinamica) della memoria.
4. Quando il programma non necessità più degli N byte, comunica al sistema che essi possono essere deallocati.

Cenno al *layout* della memoria

Abbiamo già visto che le funzioni di un programma hanno la loro *memoria locale* privata, ove risiedono, ad esempio, le loro variabili locali. D'altro canto, le variabile globali del programma, accessibili da tutte le funzioni, risiedono nella *memoria globale* del programma. È importante tenere presente che lo spazio allocato dinamicamente (da `malloc`, `calloc` ecc.) *risiede nella memoria globale* del programma, ed è quindi accessibile da *tutte* le funzioni che costituiscono il programma.

La memoria allocata dinamicamente non è mai automaticamente liberata, durante l'esecuzione del programma: essa è liberata solo quando il programmatore ne fa richiesta. Quindi è di *fondamentale importanza* evitare che il codice conduca all'allocazione di blocchi di memoria non più utilizzabili perché non più raggiungibili da un puntatore, ma tuttavia non ancora liberati (*memory leakage*).



Schema di layout della memoria di un programma C

Il file di intestazione `stdlib.h`

Il file d'intestazione `stdlib.h` contiene, *inter alia*, le dichiarazioni delle funzioni dedicate alla gestione dinamica della memoria.

Il file di intestazione `stdlib.h`

Il file d'intestazione `stdlib.h` contiene, *inter alia*, le dichiarazioni delle funzioni dedicate alla gestione dinamica della memoria.

- `void *malloc(size_t N)`
- `void *calloc(size_t n, size_t dim)`
- `void realloc(void *ptr, size_t M)`
- `void free(void *ptr)`

Il file di intestazione `stdlib.h`

Il file d'intestazione `stdlib.h` contiene, *inter alia*, le dichiarazioni delle funzioni dedicate alla gestione dinamica della memoria.

- `void *malloc(size_t N)`
- `void *calloc(size_t n, size_t dim)`
- `void realloc(void *ptr, size_t M)`
- `void free(void *ptr)`

Prima di parlare di queste funzioni discutiamo i due nuovi tipi che in esse figurano:

```
size_t
void *
```

Il tipo `size_t` e l'operatore `sizeof`

Il tipo `size_t` è definito nel file di intestazione `stddef.h`, che è già incluso da `stdlib.h`. I dati di tipo `size_t` rappresentano *la dimensione in byte* di oggetti che risiedono in memoria centrale. Ne segue che i valori rappresentati da questo tipo sono *interi non negativi*, o come anche si dice in programmazione, *interi privi di segno* (*unsigned integers*).

Il tipo `size_t` e l'operatore `sizeof`

Il tipo `size_t` è definito nel file di intestazione `stddef.h`, che è già incluso da `stdlib.h`. I dati di tipo `size_t` rappresentano *la dimensione in byte* di oggetti che risiedono in memoria centrale. Ne segue che i valori rappresentati da questo tipo sono *interi non negativi*, o come anche si dice in programmazione, *interi privi di segno* (*unsigned integers*).

Il linguaggio C comprende l'operatore unario:

`sizeof`

la cui valutazione risulta pari alla dimensione del suo argomento espressa in byte e rappresentata come valore del tipo `size_t`. Se l'argomento è il nome di un tipo racchiuso fra parentesi, il valore in questione è il numero di byte necessario a memorizzare un dato di quel tipo. Si noti che ciò vale sia per i tipi primitivi che per i tipi definiti, incluse le strutture.

Per esempio,

```
size_t N = sizeof (double);
```

dichiara una variabile `N` di tipo `size_t`, e le assegna il numero di byte necessari a memorizzare un `double`. Questo valore *dipende* dalla specifica implementazione del compilatore C.
Si supponga invece di aver dichiarato, per esempio:

```
1  struct punto //Etichetta (opzionale) della struttura
2  {
3      double x; //Primo membro
4      double y; //Secondo membro
5  };
```

Allora la riga:

```
11 size_t N=sizeof (struct punto);
```

assegna ad `N` il numero di byte necessary a memorizzare un valore del tipo di dato strutturato non primitivo: `struct punto`.

sizeof.c

```c
#include <stdio.h>
int main(void)
{
    // typedef rinonima un tipo: typedef Tipo NuovoNome;
    // Cfr. Laboratorio.
    typedef struct
        {
            int a;
            double b;
        } esempio;

    // %lu per stampare unsigned long restituito da sizeof
    printf("Dim. tipo esempio: %lu\n", sizeof (esempio));
    printf("Dim. tipo int: %lu\n", sizeof (int));
    double d;
    printf("Dim. tipo double: %lu\n", sizeof d);
    printf("Dim. tipo float: %lu\n", sizeof (float));

    return 0;
}
```

Il tipo `void *`

Il tipo "puntatore a `void`", ossia:

```
void *
```

è usato per un puntatore che fa riferimento a una zona di memoria contenente dati il cui tipo *non è specificato*. Esso è stato introdotto per risolvere il problema che le funzioni usate per allocare zone di memoria in realtà *non sanno* quali tipi di dati saranno memorizzati in una data zona di memoria. Ne segue che esse *non sanno* quale tipo di puntatore restituire al chiamante.

Per esempio, potremmo voler allocare memoria per un `int`:

```
malloc( sizeof (int) );
```

oppure per un `double`:

```
malloc( sizeof (double) );
```

Cosa deve restituire `malloc`? Il tipo `int *`, oppure il tipo `double *`?

Per risolvere questo problema, le versioni moderne del C permettono di usare la sintassi seguente:

```
int *pt1 = malloc( sizeof (int) );
double *pt2 = malloc( sizeof (double) );
```

Queste istruzioni chiedono di allocare memoria sufficiente per un `int` e un `double`, rispettivamente. Se la chiamate non generano errori in esecuzione, `pt1` e `pt2` puntano alle zone di memoria allocate a questi fini, rispettivamente.

Per risolvere questo problema, le versioni moderne del C permettono di usare la sintassi seguente:

```
int *pt1 = malloc( sizeof (int) );
double *pt2 = malloc( sizeof (double) );
```

Queste istruzioni chiedono di allocare memoria sufficiente per un `int` e un `double`, rispettivamente. Se la chiamate non generano errori in esecuzione, `pt1` e `pt2` puntano alle zone di memoria allocate a questi fini, rispettivamente.

La variabile `pt1` è un puntatore a `int`, ma la funzione `malloc` restituisce `void *`. La conversione dall'uno all'altro tipo è eseguita *automaticamente*. Similmente per la variabile `pt2`, che è un puntatore a `double`.

Per risolvere questo problema, le versioni moderne del C permettono di usare la sintassi seguente:

```
int *pt1 = malloc( sizeof (int) );
double *pt2 = malloc( sizeof (double) );
```

Queste istruzioni chiedono di allocare memoria sufficiente per un `int` e un `double`, rispettivamente. Se la chiamate non generano errori in esecuzione, `pt1` e `pt2` puntano alle zone di memoria allocate a questi fini, rispettivamente.

La variabile `pt1` è un puntatore a `int`, ma la funzione `malloc` restituisce `void *`. La conversione dall'uno all'altro tipo è eseguita *automaticamente*. Similmente per la variabile `pt2`, che è un puntatore a `double`.

Il tipo `void *` ha qualche altro uso, sebbene quanto appena visto sia il suo scopo primario. Non ne parleremo oltre.

```
void *malloc(size_t N)
```

```
void *malloc(size_t N)
```

1. Restituisce un puntatore a una zona di memoria di `N` byte (non necessariamente fisicamente contigui) nella memoria centrale, o `NULL` se la richiesta non può essere esaudita.

`void *malloc(size_t N)`

1. Restituisce un puntatore a una zona di memoria di `N` byte (non necessariamente fisicamente contigui) nella memoria centrale, o `NULL` se la richiesta non può essere esaudita.
2. È un *errore grave* non testare il valore restituito da `malloc`. Uso corretto:

```
1  int *pt = malloc(sizeof (int));
2  if (pt != NULL)
3      printf("Allocazione riuscita\n!");
4  else
5      printf("Allocazione fallita\n!");
```

`void *malloc(size_t N)`

1. Restituisce un puntatore a una zona di memoria di `N` byte (non necessariamente fisicamente contigui) nella memoria centrale, o `NULL` se la richiesta non può essere esaudita.
2. È un *errore grave* non testare il valore restituito da `malloc`. Uso corretto:

```
1  int *pt = malloc(sizeof (int));
2  if (pt != NULL)
3      printf("Allocazione riuscita\n!");
4  else
5      printf("Allocazione fallita\n!");
```

3. È pure un errore, in questo esempio, usare una costante nell'argomento di `malloc`, per quanto plausibile essa possa apparire — per esempio, 4: infatti, la dimensione dei dati di tipo `int` è *dipendente dall'implementazione*. Usate *sempre* `sizeof`.

```
void *calloc(size_t n, size_t dim)
```

```
void *calloc(size_t n, size_t dim)
```

1. Come `malloc`, ma alloca un totale di `N=n*dim` byte e li inizializza tutti a zero. Restituisce `NULL` se non è possibile esaudire la richiesta.

`void *calloc(size_t n, size_t dim)`

1. Come `malloc`, ma alloca un totale di `N=n*dim` byte e li inizializza tutti a zero. Restituisce `NULL` se non è possibile esaudire la richiesta.
2. Quindi, per esempio,

```
calloc(10, sizeof (int))
```

e

```
malloc(sizeof (int[10]))
```

sono equivalenti, eccetto che la prima chiamata inizializza la memoria.

```
void *realloc(void *ptr, size_t M)
```

`void *realloc(void *ptr, size_t M)`

- Restituisce un puntatore a una zona di memoria di `M` byte che estende o riduce la zona di memoria di `N` byte puntata da `ptr`. Il puntatore `ptr` *deve* essere stato restituito da una precedente chiamata a `malloc`, `calloc` o `realloc`. I dati presenti nella regione di `N` byte sono preservati, fino alla lunghezza di `M` byte. Se `M>N`, i nuovi `M-N` byte allocati non sono inizializzati. Se `N>M`, i rimanenti `N-M` byte precedentemente allocati vanno persi. Le chiamate con `M` pari a `0` hanno effetto indefinito. Se `ptr` è `NULL`, equivale a `malloc(M)`.

`void *realloc(void *ptr, size_t M)`

- Restituisce un puntatore a una zona di memoria di `M` byte che estende o riduce la zona di memoria di `N` byte puntata da `ptr`. Il puntatore `ptr` *deve* essere stato restituito da una precedente chiamata a `malloc`, `calloc` o `realloc`. I dati presenti nella regione di `N` byte sono preservati, fino alla lunghezza di `M` byte. Se `M>N`, i nuovi `M-N` byte allocati non sono inizializzati. Se `N>M`, i rimanenti `N-M` byte precedentemente allocati vanno persi. Le chiamate con `M` pari a `0` hanno effetto indefinito. Se `ptr` è `NULL`, equivale a `malloc(M)`.
- Restituisce il puntatore alla zona di memoria opportunamente estesa o ridotta, oppure `NULL` se non può esaudire la richiesta. In questo caso, `ptr` e l'allocazione precedente sono invariate.

`void *realloc(void *ptr, size_t M)`

- Restituisce un puntatore a una zona di memoria di `M` byte che estende o riduce la zona di memoria di `N` byte puntata da `ptr`. Il puntatore `ptr` *deve* essere stato restituito da una precedente chiamata a `malloc`, `calloc` o `realloc`. I dati presenti nella regione di `N` byte sono preservati, fino alla lunghezza di `M` byte. Se `M>N`, i nuovi `M-N` byte allocati non sono inizializzati. Se `N>M`, i rimanenti `N-M` byte precedentemente allocati vanno persi. Le chiamate con `M` pari a `0` hanno effetto indefinito. Se `ptr` è `NULL`, equivale a `malloc(M)`.
- Restituisce il puntatore alla zona di memoria opportunamente estesa o ridotta, oppure `NULL` se non può esaudire la richiesta. In questo caso, `ptr` e l'allocazione precedente sono invariate.
- *Nota.* Il ridimensionamento della zona originariamente puntata da `ptr` può comportare uno *spostamento*: il puntatore restituito può avere valore diverso da `ptr`.

```
void free(void *ptr)
```

```
void free(void *ptr)
```

- Dealloca lo spazio di memoria precedentemente allocato tramite `malloc`, `calloc` o `realloc`, cui punta `ptr`. Se l'argomento è `NULL` non ha alcun effetto.

`void free(void *ptr)`

- Dealloca lo spazio di memoria precedentemente allocato tramite `malloc`, `calloc` o `realloc`, cui punta `ptr`. Se l'argomento è `NULL` non ha alcun effetto.
- L'argomento *deve* essere un puntatore che è stato precedentemente restituito da una chiamata a `malloc`, `calloc` o `realloc`.

`void free(void *ptr)`

- Dealloca lo spazio di memoria precedentemente allocato tramite `malloc`, `calloc` o `realloc`, cui punta `ptr`. Se l'argomento è `NULL` non ha alcun effetto.
- L'argomento *deve* essere un puntatore che è stato precedentemente restituito da una chiamata a `malloc`, `calloc` o `realloc`.
- *Nota.* L'uso scorretto delle funzioni di allocazione e deallocazione della memoria può portare al cosiddetto *memory leakage*, ossia *perdita* o *fuoriuscita di memoria*. Il caso più ovvio si ha quando si alloca memoria, si sovrascrive per errore il puntatore `ptr` restituito dalla funzione di allocazione, e quindi non si ha più la possibilità di fare riferimento alla zona di memoria allocata — né, in particolare, di deallocarla. Si tratta di errori di programmazione *gravi*.

alldin.c

```c
#include<stdio.h>
#include<stdlib.h> //Per malloc, realloc e free
int main(int argc, char *argv[])
{
    if (argc==1)
    {
        printf("Inserisci almeno un numero intero come parametro.\n"); return -1;
    }
    int *pt = calloc(argc-1, sizeof (int));
    if (pt == NULL)
    {
        printf("Errore nell'allocazione della memoria.\n");
        return -1;
    }
    for (int i=0; i<argc-1; i++)
        pt[i]=atoi(argv[i+1]);

    printf("Inserisci un altro intero.\n");
    int *tmp = realloc(pt, (sizeof (int))*argc); //ridimensionamento
    if ( tmp != NULL)
    {
        pt = tmp;
        scanf("%d", pt+argc-1); getchar();
    }
    else
    {
        printf("Errore nell'allocazione della memoria.\n"); return -1;
    }
    printf("Ecco:\n");
    for (int i=0; i<argc; i++)
        printf("%d\n",pt[i]);
    free(pt);
    printf("Ho deallocato la memoria. Addio.\n");
    return 0;
}
```

Array multidimensionali come parametri

- Abbiamo già usato gli array multidimensionali in laboratorio. Va ricordato che tali array sono implementati come array monodimensionali i cui elementi sono a loro volta array della dimensione appropriata. Per esempio, `int v[2][3]` dichiara un array di 2 elementi, ciascuno dei quali è un array di 3 `int`.

Array multidimensionali come parametri

- Abbiamo già usato gli array multidimensionali in laboratorio. Va ricordato che tali array sono implementati come array monodimensionali i cui elementi sono a loro volta array della dimensione appropriata. Per esempio, `int v[2][3]` dichiara un array di 2 elementi, ciascuno dei quali è un array di 3 `int`.
- L'identificatore `v`, al solito, è un puntatore costante al primo elemento dell'array, ossia all'array di tre `int` che costituisce la prima riga della matrice `v[2][3]`.

Array multidimensionali come parametri

- Abbiamo già usato gli array multidimensionali in laboratorio. Va ricordato che tali array sono implementati come array monodimensionali i cui elementi sono a loro volta array della dimensione appropriata. Per esempio, `int v[2][3]` dichiara un array di 2 elementi, ciascuno dei quali è un array di 3 `int`.
- L'identificatore `v`, al solito, è un puntatore costante al primo elemento dell'array, ossia all'array di tre `int` che costituisce la prima riga della matrice `v[2][3]`.
- Ne segue che quando si passa `v` a una funzione, è necessario specificare il *numero di colonne* (più in generale, la dimensione dell'ultimo indice dell'array) esplicitamente, come nel prototipo:

```
void f(int v[][3]);
```

Si può rendere il valore 3 di questo esempio parametrico, aggiungendo un parametro intero, come segue.

arraypar.c

```c
#include <stdio.h>
int main(void)
{
        void visualizza(int, int c, int[][c]); /* c serve! */
        int v[2][3];

        int i, j;
        for (i=0;i<2;i++) /* riga */
                for (j=0;j<3;j++) /* colonna */
                        v[i][j]=-1;
        visualizza(2,3,v);
        return 0;
}
void visualizza(int r, int c, int v[][c]) /* int v[][c] +dopo+ c */
{
        int i,j;
        for (i=0;i<r;i++) /* riga */
        {
                for (j=0;j<c;j++) /* colonna */
                        printf("%d\t", v[i][j]);
                putchar('\n');
        }
}
```

Il preprocessore

La compilazione del sorgente C è preceduta da una fase di preparazione detta *preprocessing* (*pre-elaborazione*). In questa fase, il *preprocessore* C esegue una serie di compiti preliminari quali, per esempio, l'inclusione dei file di intestazione.

Il programmatore può richiedere al preprocessore l'esecuzione di specifici compiti tramite istruzioni dedicate note come *direttive per il preprocessore*. Esse *non* fanno parte del linguaggio C in senso proprio.

Il preprocessore

La compilazione del sorgente C è preceduta da una fase di preparazione detta *preprocessing* (*pre-elaborazione*). In questa fase, il *preprocessore* C esegue una serie di compiti preliminari quali, per esempio, l'inclusione dei file di intestazione.

Il programmatore può richiedere al preprocessore l'esecuzione di specifici compiti tramite istruzioni dedicate note come *direttive per il preprocessore*. Esse *non* fanno parte del linguaggio C in senso proprio.

Le direttive per il preprocessore cominciano tutte con il simbolo

#

variamente detto *hash*, *cancelletto*, o *diesis*. Si estendono fino alla fine della riga in cui compare #, ma non comprendono automaticamente le riga successiva.

`#include`

- Come abbiamo visto, include un file di intestazione. Il preprocessore sostituisce alla direttiva di inclusione il file indicato.

`#include`

- Come abbiamo visto, include un file di intestazione. Il preprocessore sostituisce alla direttiva di inclusione il file indicato.
- Nella forma `#include <...>`, il file in argomento è cercato dal sistema in una serie di directory che contiene le libreria standard del C. La semantica precisa dipende dall'implementazione.

`#include`

- Come abbiamo visto, include un file di intestazione. Il preprocessore sostituisce alla direttiva di inclusione il file indicato.
- Nella forma `#include <...>`, il file in argomento è cercato dal sistema in una serie di directory che contiene le libreria standard del C. La semantica precisa dipende dall'implementazione.
- Nella forma `#include "..."`, il file in argomento è cercato dal sistema in una serie di directory dipendente dall'implementazione, che tipicamente include la directory in cui risiede il file sorgente nel quale compare la direttiva. Altrettanto tipicamente, se il file in argomento è un percorso relativo, esso è interpretato dal sistema come relativo al punto in cui risiede il file sorgente.

`#include`

- Come abbiamo visto, include un file di intestazione. Il preprocessore sostituisce alla direttiva di inclusione il file indicato.
- Nella forma `#include <...>`, il file in argomento è cercato dal sistema in una serie di directory che contiene le libreria standard del C. La semantica precisa dipende dall'implementazione.
- Nella forma `#include "..."`, il file in argomento è cercato dal sistema in una serie di directory dipendente dall'implementazione, che tipicamente include la directory in cui risiede il file sorgente nel quale compare la direttiva. Altrettanto tipicamente, se il file in argomento è un percorso relativo, esso è interpretato dal sistema come relativo al punto in cui risiede il file sorgente.
- I file inclusi possono a loro volta contenere direttive `#include`.

`#define`

- La direttiva

  `#define` *identificatore testo*

  chiede al preprocessore di sostituire nel file sorgente, a partire dal punto in cui essa compare, ciascuna occorrenza di *identificatore* con *testo*. Si dice nel gergo del C che `identificatore` è una macro.

`#define`

- La direttiva

  `#define` *identificatore testo*

  chiede al preprocessore di sostituire nel file sorgente, a partire dal punto in cui essa compare, ciascuna occorrenza di *identificatore* con *testo*. Si dice nel gergo del C che `identificatore` è una macro.

- Una successiva direttiva `#define` con il medesimo *identificatore* — cioè, un tentativo di ridefinizione — causa un errore in compilazione, a meno che *testo* non sia identico nei due casi. (Gli spazi prima e dopo *testo* sono ignorati; in *testo*, più spazi equivalgono a uno solo.)

#define

- La direttiva

  `#define` *identificatore testo*

  chiede al preprocessore di sostituire nel file sorgente, a partire dal punto in cui essa compare, ciascuna occorrenza di *identificatore* con *testo*. Si dice nel gergo del C che `identificatore` è una macro.
- Una successiva direttiva `#define` con il medesimo *identificatore* — cioè, un tentativo di ridefinizione — causa un errore in compilazione, a meno che *testo* non sia identico nei due casi. (Gli spazi prima e dopo *testo* sono ignorati; in *testo*, più spazi equivalgono a uno solo.)
- Si usa di frequente la convenzione di definire le macro in maiuscola, per distinguerle da altri identificatori quali i nomi di variabili. Si possono anche definire macro con parametri, ma non approfondiremo l'argomento.

macro.c

```c
#include "definizioni.h"
#include "funzioniaux.h"
int main(void)
{
    int v[MAXL];
    inizializza(v,INT_IN);
    visualizza(v);
    return 0;
}
```

funzioniaux.h

```c
#include "definizioni.h"
#include <stdio.h>

void inizializza(int *, int); /* Prototipi */
void visualizza(int *);

void inizializza(int *v, int d) /* Implementazioni */
{
    int i=0;
    for (;i<MAXL;i++)
        v[i]=d;
}
void visualizza(int *v)
{
    int i=0;
    for (;i<MAXL;i++)
        printf("%d\n",v[i]);
}
```

definizioni.h

```c
#define MAXL 20 /* Lunghezza max array */
#define INT_IN 1 /* Valore di default elementi array int */
```

Conversioni di tipo implicite ed esplicite

- In molte situazioni il compilatore C si trova a dover trasformare dati di un certo tipo in dati di un altro tipo. Si parla di conversione di tipo. La conversione può avvenire *implicitamente*, o può essere *esplicitamente* richiesta dal programmatore. Nel secondo caso si parla di casting, o conversione forzata dei tipi.

Conversioni di tipo implicite ed esplicite

- In molte situazioni il compilatore C si trova a dover trasformare dati di un certo tipo in dati di un altro tipo. Si parla di conversione di tipo. La conversione può avvenire *implicitamente*, o può essere *esplicitamente* richiesta dal programmatore. Nel secondo caso si parla di casting, o conversione forzata dei tipi.
- Esiste un complesso sistema di regole, formalmente definite, che normano la conversione dei tipi in C. Qui daremo solo qualche cenno. Per un approfondimento, si veda K&R, Sezione 6 dell'Appendice A.

Conversioni di tipo implicite ed esplicite

- In molte situazioni il compilatore C si trova a dover trasformare dati di un certo tipo in dati di un altro tipo. Si parla di conversione di tipo. La conversione può avvenire *implicitamente*, o può essere *esplicitamente* richiesta dal programmatore. Nel secondo caso si parla di casting, o conversione forzata dei tipi.
- Esiste un complesso sistema di regole, formalmente definite, che normano la conversione dei tipi in C. Qui daremo solo qualche cenno. Per un approfondimento, si veda K&R, Sezione 6 dell'Appendice A.
- Un caso semplice si ha quando si combinano tipi diversi in una singola espressione. Per esempio, se `i` è `int` e `d` è `double`, l'espressione

  `d/i`

  ha tipo `double`, e la divisione è eseguita in virgola mobile.

- Questo è un esempio di conversione implicita. Si noti che questa conversione implicita di `d/i` *non comporta perdita di informazione*: il tipo `int` è sottotipo del tipo `double`.

- Questo è un esempio di conversione implicita. Si noti che questa conversione implicita di `d/i` *non comporta perdita di informazione*: il tipo `int` è sottotipo del tipo `double`.
- D'altro canto, sappiamo già che se `j` è anch'esso `int`, l'espressione

  `j/i`

  ha ancora tipo `int`, e la divisione eseguita è quella intera: il risultato è il quoziente; il resto è scartato. Non si ha conversione.

- Questo è un esempio di conversione implicita. Si noti che questa conversione implicita di `d/i` *non comporta perdita di informazione*: il tipo `int` è sottotipo del tipo `double`.
- D'altro canto, sappiamo già che se `j` è anch'esso `int`, l'espressione

  ```
  j/i
  ```

  ha ancora tipo `int`, e la divisione eseguita è quella intera: il risultato è il quoziente; il resto è scartato. Non si ha conversione.
- Analogamente, `char` è sottotipo di `int`. Così, se `c` e `e` sono `char` e `i` è `int`, l'assegnazione

  ```
  i=c-e;
  ```

  è legittima, e comporta una conversione automatica del tipo dell'espressione a membro destro nel tipo dell'espressione a membro sinistro. Tuttavia, il valore del risultato è qui *dipendente dall'implementazione*, perché lo standard lascia libertà alle implementazioni di rappresentare `char` in aritmetica con segno o senza segno.

- Per richiedere al compilatore una conversione forzata, o cast, di una data espressione *espr* al tipo `T`, si usa la sintassi:

`(T)` *espr*

- Per richiedere al compilatore una conversione forzata, o cast, di una data espressione *espr* al tipo `T`, si usa la sintassi:

  `(T)` *espr*

- Ovviamente in alcuni casi tale conversione può portare a *perdita di informazione*: per esempio, quando si chiede di convertire da un tipo in virgola mobile a un tipo intero.

- Per richiedere al compilatore una conversione forzata, o cast, di una data espressione *espr* al tipo `T`, si usa la sintassi:

  `(T)` *espr*

- Ovviamente in alcuni casi tale conversione può portare a *perdita di informazione*: per esempio, quando si chiede di convertire da un tipo in virgola mobile a un tipo intero.
- Più in generale, si ha *sempre* potenziale perdita di informazione quando il tipo di *espr* non è un sottotipo di `T`.

- Per richiedere al compilatore una conversione forzata, o cast, di una data espressione *espr* al tipo `T`, si usa la sintassi:

  `(T)` *espr*

- Ovviamente in alcuni casi tale conversione può portare a *perdita di informazione*: per esempio, quando si chiede di convertire da un tipo in virgola mobile a un tipo intero.
- Più in generale, si ha *sempre* potenziale perdita di informazione quando il tipo di *espr* non è un sottotipo di `T`.
- Vi sono casi in cui il casting è utile. *Tuttavia, esso va usato con cautela.* Se ti trovi a scrivere casting che possono comportare perdita d'informazione, chiediti innanzitutto se la struttura del tuo programma e la scelta dei tipi di dati siano corrette.

Buona prosecuzione degli studi!