Introduzione alla programmazione in C

Appunti delle lezioni di Tecniche di programmazione

Giorgio Grisetti
Luca Iocchi
Daniele Nardi
Fabio Patrizi
Alberto Pretto

Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Facoltà di Ingegneria dell'Informazione, Informatica, Statistica
Università di Roma "La Sapienza"

Edizione 2020/2021

2021

Indice

4. Funzioni

4.1. Astrazione sulle operazioni: funzioni

L'astrazione sulle operazioni è supportata da tutti i linguaggi di programmazione attuali (Java, C#, C++, C, Pascal, Fortran, Lisp, ecc.). In C l'astrazione sulle operazioni si realizza attraverso la nozione di *funzione*. Una funzione può essere vista come una scatola nera che prende dei parametri in ingresso e restituisce dei risultati o compie delle azioni.

4.2. Definizione di funzioni

Definizione di una funzione

Sintassi:

```
intestazione
{
    istruzioni
}
```

- *intestazione* ha la seguente forma:

  ```
  tipoRisultato nomeFunzione(parametriFormali)
  ```

  dove

  - *tipoRisultato* è il tipo del risultato restituito dalla funzione oppure void se non viene restituito alcun risultato
  - *nomeFunzione* è il nome della funzione
  - *parametriFormali* è una lista di dichiarazioni di parametri (tipo e nome) separate da virgola; ogni parametro è una variabile; la lista di parametri può anche essere vuota.

- *istruzioni* sono le istruzioni che saranno eseguite all'invocazione della funzione stessa.

Semantica:
Definisce una funzione specificandone intestazione e corpo.

- L'intestazione indica il nome della funzione, il numero e il tipo dei parametri formali e il tipo dell'eventuale valore restituito.
- Il corpo della funzione specifica le istruzioni che devono essere eseguite quando essa viene attivata.
- I parametri formali servono a passare informazioni da elaborare nel corpo della funzione. I parametri formali vengono utilizzati nel corpo della funzione esattamente come variabili già inizializzate (l'inizializzazione avviene all'atto dell'invocazione della funzione, assegnando ai parametri formali il valore dei parametri attuali - vedi dopo).
- L'eventuale risultato restituito è il valore dell'invocazione della funzione. Se la funzione non restituisce risultati, allora essa viene utilizzata per effettuare operazioni (ed il tipo del risultato è void).

Esempio:

La funzione main ha la forma

```
int main () {
  ...
}
```

4.2.1. Risultato di una funzione: l'istruzione return

Istruzione return

Sintassi:

```
return espressione;
```

dove

- *espressione* è un'espressione il cui valore è compatibile con il tipo del risultato dichiarato nell'intestazione della funzione.

Semantica:
L'istruzione return eseguita all'interno di una funzione termina l'esecuzione della stessa e restituisce il risultato alla parte di programma dove è stata invocata la funzione.

Esempio:

```
int main() {
    return 0;  // terminazione normale del programma
}
```

Se il tipo del risultato della funzione è void, l'istruzione return si può omettere, oppure può essere usata semplicemente per interrompere l'esecuzione della funzione stessa. Dato che non si deve restituire un risultato, la sintassi dell'istruzione è in questo caso:

```
return;
```

L'esecuzione dell'istruzione return fa sempre terminare la funzione, anche se ci sono altre istruzioni che seguono.

4.2.2. Esempi di definizione di funzioni

Esempio: Funzione di stampa

```
void stampaSaluto() {
  printf("Buon giorno!\n");
}
```

La funzione stampaSaluto non ha parametri formali e non restituisce un risultato. Il suo corpo è costituito da una sola istruzione che stampa sullo schermo la stringa "Buon giorno!"

Esempio: Funzione per calcolo matematico

```
double distanza(double x1, double y1, double x2,
    double y2)
{
    return sqrt(pow(x2-x1,2)+pow(y2-y1,2));
}
```

La funzione distanza ha 4 parametri formali di tipo double e restituisce un valore dello stesso tipo. In particolare, calcola e restituisce la distanza tra i punti (x1,y1) e (x2,y2).

Esempio: Potenza tramite definizione di due funzioni

```
int moltiplicazione(int moltiplicando,
                    int moltiplicatore) {
  int prodotto = 0;
  while (moltiplicatore > 0) {
    moltiplicatore--;
    prodotto = prodotto + moltiplicando;
  }
  return prodotto;
}
```

```
int potenza(int base, int esponente) {
  int risultato = 1;
  while (esponente > 0) {
    esponente--;
    risultato = moltiplicazione(risultato, base);
  }
  return risultato;
}
```

Si noti che in questo caso il ciclo più interno del doppio ciclo è racchiuso nell'invocazione della funzione moltiplicazione.

4.2.3. Funzioni: istruzioni o espressioni?

```
double i = distanza(2,3,3,2);
distanza(2,3,3,2);
stampasaluto();
```

In C le espressioni possono essere considerate a tutti gli effetti istruzioni. Ma è buona pratica di programmazione mantenere la distinzione per aumentare la leggibilità del programma.

4.2.4. Segnatura di una funzione

La **segnatura** di una funzione consiste nel nome della funzione e nella descrizione (tipo, numero e posizione) dei suoi parametri.

Esempi:

- sqrt(double x)
- pow(double b, int e)

Nota: il nome del parametro non è significativo nella segnatura.

Il C non consente l'*overloading* (sovraccarico), in quanto non si possono definire funzioni con lo stesso nome e diversa segnatura.

Esempio:

```
int somma(int x, int y) {
    return x+y;
}

int somma(double x, double y) {
    return (int)(x+y);
}

int somma(int x, int y, int z) {
    return x+y+z;
}
```

Le tre definizioni non sono compatibili in C.

Se si usa il compilatore C++ (g++) vengono invece compilate!

4.2.5. Dichiarazione delle funzioni

Abbiamo già visto che in C si applica la regola di *dichiarazione prima dell'uso*. Per le funzioni il compilatore C, tenta di attribuire un tipo agli argomenti di una funzione se questa non è stata precedentemente dichiarata. Tuttavia, è buona prassi far precedere l'intestazione della funzione al suo uso.

promo-arg.c

```
#include <stdio.h>

int somma(int x, int y);

int main(){
    int i = 1;
    double d = 1.0;
    printf("int somma(int x, int y) %d", somma(i,i));
}

int somma(int x, int y) {
    return x+y;
}
```

Se si omette la dichiarazione della intestazione della funzione vengono generati dei messaggi di **warning**, ma il programma viene compilato e può essere eseguito. Tuttavia in alcune situazioni, si possono generare degli errori dovute alle conversioni di tipo che vengono fatte in modo automatico e si raccomanda di inserire sempre la dichiarazione della funzione.

4.2.5.1. Organizzazione del codice

L'applicazione delle regole per la dichiarazione di funzioni suggerisce una strutturazione del codice nelle seguenti sezioni:

- direttive del compilatore (comandi che iniziano con il carattere #, per il momento #include);
- dichiarazioni di tipi;
- dichiarazioni di variabili;
- intestazioni di funzioni;
- programma principale main;
- definizione delle funzioni;

4.3. Passaggio dei parametri

Come abbiamo detto, la definizione di una funzione presenta nell'intestazione una lista di **parametri formali**. Questi parametri sono utilizzati come variabili all'interno del corpo della funzione.

L'invocazione di una funzione contiene i parametri da utilizzare come argomenti della funzione stessa. Questi parametri sono detti **parametri attuali**, per distinguerli dai parametri formali presenti nell'intestazione della definizione della funzione.

Quando attraverso una invocazione **attiviamo** una funzione, dobbiamo **legare** i parametri attuali ai parametri formali. In generale esistono diversi modi possibili di effettuare questo legame.

In C/C++ occorre distinguere tre casi di passaggio dei parametri:

- passaggio per valore
- passaggio mediante puntatori
- passaggio per riferimento (solo in C++)

4.3.1. Passaggio di parametri per valore

Sia pa un parametro attuale presente nell'invocazione della funzione e pf il corrispondente parametro formale nella definizione della funzione: legare pa a pf per valore significa, al momento della attivazione della funzione:

1. valutare il parametro attuale pa (che in generale è un'espressione)
2. associare una locazione di memoria al parametro formale pf
3. inizializzare tale locazione con il valore di pa.

In altre parole, il parametro formale pf si comporta esattamente come una variabile allocata al momento della attivazione della funzione e inizializzata con il valore del parametro attuale pa.

Alla fine dell'esecuzione del corpo della funzione la memoria riservata per il parametro formale viene rilasciata e il suo valore si perde.

Nota: i valori delle variabili che compaiono nell'espressione pa non vengono quindi alterati dall'esecuzione della funzione.

Esempio: Consideriamo la definizione della seguente funzione

```
int raddoppia(int x) {
    x = x * 2;
    return x;
}

int main() {
    int a,b;
    a=5;
    b=raddoppia(a+1);
    printf("a = %d\n",a);
    printf("b = %d\n",b);
}
```

Analizziamo cosa succede quando viene eseguito il programma:

1. *vengono definite* due variabili intere a,b e a viene inizializzata al valore 5;

2. *vengono valutati i parametri attuali*: nel nostro caso il parametro attuale è l'espressione a+1 che ha come valore 6;

3. *viene individuata la funzione da eseguire* in base al numero e tipo dei parametri attuali, cercando la definizione di una funzione la cui segnatura sia conforme alla invocazione: il nome della funzione deve essere lo stesso e i parametri attuali devono corrispondere in numero e tipo ai parametri formali: nel nostro caso la funzione che cerchiamo deve avere la segnatura raddoppia(int);

4. *viene sospesa l'esecuzione dell'unità di programma chiamante*: nel nostro caso la funzione main;

5. *viene allocata la memoria* per i parametri formali (considerati come variabili) e variabili definite nella funzione (vedi dopo): nel nostro caso viene allocata la memoria per il parametro formale x;

6. *viene assegnato il valore dei parametri attuali ai parametri formali*: nel nostro caso il parametro formale x viene inizializzato con il valore 6;

7. *vengono eseguite le istruzioni del corpo della funzione invocata* (a partire dalla prima): nel nostro caso il valore di x viene moltiplicato per 2 e diventa 12;

8. *l'esecuzione della funzione invocata termina* (o per l'esecuzione dell'istruzione return o perché non ci sono altre istruzioni da eseguire): nel nostro caso incontriamo l'istruzione return x;

9. *viene deallocata la memoria* utilizzata per i parametri formali e le variabili della funzione, perdendo qualsiasi informazione ivi contenuta: nel nostro caso viene rilasciata la locazione di memoria corrispondente al parametro formale x;

10. *se la funzione restituisce un risultato*, tale risultato diviene il valore dell'espressione costituita dall'invocazione nell'unità di programma chiamante: nel nostro caso il risultato è 12;

11. *si riprende l'esecuzione dell'unità di programma chiamante* dal punto in cui era stata interrotta dall'invocazione: il valore 12 viene assegnato alla variabile b.

Nel passaggio di parametri per valore il contenuto delle variabili usate come parametri attuali non viene mai modificato. Il programma illustrato stampa quindi `5 12`.

4.3.2. Passaggio di parametri tramite puntatori

L'uso di variabili di tipo puntatore consente diverse modalità di passaggio dei parametri.

Abbiamo inizialmente considerato il passaggio di parametri per *valore*, in genere utilizzato per i dati di tipo primitivo, in cui il parametro formale può essere considerato come una variabile locale che viene inizializzata al momento della chiamata della funzione con il valore corrispondente al parametro attuale.

Vediamo innanzitutto che il passaggio di parametri per valore può essere effettuato anche con il tipo puntatore, anche se viene a cadere una proprietà cruciale del passaggio di parametri per valore, cioè la garanzia che la funzione non abbia effetti sul contenuto delle variabili del programma chiamante.

Con il passaggio di puntatori diventa possibile utilizzare il puntatore per restituire al programma chiamante il risultato di una funzione.

Il passaggio di parametri di tipo puntatore risulta necessario anche quando i dati che devono essere scambiati tra funzione chiamata e programma chiamante sono voluminosi ed il passaggio del puntatore

risulta molto più efficiente sia in termini di occupazione di memoria che di tempo di calcolo (non occorre la copia del parametro).

Esempio: La seguente funzione ha lo scopo di scambiare il valore di due variabili.

```
void swap (int *a, int *b) {
    int temp = *b;
    *b = *a;
    *a = temp;
}
```

```
int main (){
    int j,i;
    i = 1;  j = 2;
    printf("i = %d\n",i);
    printf("j = %d\n",j);
    swap(&i, &j);
    printf("dopo swap\n");
    printf("i = %d\n",i);
    printf("j = %d\n",j);
}
```

Si noti che in questo caso, nonostante il passaggio di parametri sia per valore, il risultato dell'esecuzione della funzione swap consiste proprio nel modificare il valore di due variabili del programma principale.

In pratica, vengono passati alla funzione i puntatori a due variabili e pertanto è possibile modificare il valore delle variabili puntate, anche se il valore delle variabili puntatore rimane inalterato.

Il meccanismo di passaggio tramite puntatore può essere sfruttato in una funzione per restituire più valori.

Esempio: La funzione puntoMedio calcola le coordinate del punto medio dei punti (x1,y1) e (x2,y2) e ne memorizza le coordinate nelle variabili a cui fanno riferimento i puntatori passati come ultimi due parametri.

punto-medio.c

```
#include <math.h>
#include <stdio.h>

void puntoMedio(double x1, double y1,
     double x2, double y2, double* xm, double* ym){
    *xm = (x1 + x2)/2;
```

```
    *ym = (y1 + y2)/2;
}

int main(){
    double x1 = 1, y1 = 1, x2 = 3, y2 = 5, xm, ym;
    puntoMedio(x1,y1,x2,y2,&xm,&ym);
    printf("punto medio = (%lf,%lf)\n", xm, ym);
}
```

Si noti come al termine dell'esecuzione della funzione puntoMedio, il risultato sia visibile nelle variabili xm e ym, i cui riferimenti sono stati forniti in input tramite puntatori (ultimi due parametri) alla funzione.

4.3.2.1. Parametri puntatori a costanti

Quando il parametro passato per riferimento non deve essere modificato dalla funzione, si può utilizzare la specifica const

```
void f (const double *x) {
    *x=2.0; // ERRORE di compilazione
            // *x non puo' essere modificato
    ...
}
```

In questo caso il compilatore segnala che c'è un'assegnazione non permessa. In generale con i puntatori a costanti si possono impedire effetti collaterali indesiderati.

4.3.3. Passaggio di parametri per riferimento

Il passaggio dei parametri per riferimento viene illustrato nel seguito per completare l'analisi delle modalità di passaggio dei parametri. Occorre tuttavia ricordare che il passaggio di parametri per riferimento non è consentito in C (ma solo in C++).

Nel passaggio dei parametri per riferimento, il parametro attuale fornito alla funzione è il riferimento (cioè l'indirizzo di memoria) di una variabile. Tuttavia, nella chiamata e nel corpo della funzione si usa la notazione delle variabili normale, cioè senza puntatore.

La dichiarazione di un parametro passato per riferimento avviene usando il carattere & davanti al nome del parametro formale.

Esempio:

```
void swapRef (int &a, int &b) {
    int temp;
    temp = b;
    b = a;
    a = temp;
    return;
}
```

```
int main (){
    int i = 1;
    int j = 2;
    printf("i = %d\n",i);
    printf("j = %d\n",j);
    swapRef(i, j);
    printf("dopo swapRef\n");
    printf("i = %d\n",i);
    printf("j = %d\n",j);
    return 0;
}
```

In questo caso, la funzione C++ che scambia il valore delle variabili, si comporta esattamente come la precedente, ma l'uso del passaggio di parametro per riferimento ne rende più chiara la scrittura ed esplicita la possibilità di modificare i contenuti delle variabili del modulo di programma chiamante, senza usare variabili puntatore esplicite.

4.3.4. Valori restituiti di tipo puntatore

Le funzioni C possono restituire valori di tipo puntatore o di tipo riferimento. Cioè, il risultato di una funzione può essere di tipo puntatore.

```
double *puntatore (double a) {
    double *r = (double *) malloc(sizeof(double));
    *r = a;
    return r;
}

int main () {
    double *pd = puntatore(5.4);
    printf("pd = %p\n",pd);
    printf("*pd = %f\n",*pd);
    return 0;
}
```

In questo esempio la funzione puntatore crea un puntatore ad una variabile di tipo double e la inizializza con il valore passato come argomento.

Si noti che la memoria allocata dinamicamente con la funzione malloc non viene rilasciata al termine dell'esecuzione della funzione. Nell'esempio precedente la variabile puntatore r (variabile locale allocata staticamente) viene deallocata al termine della funzione, mentre la locazione di memoria puntata da r (allocata dinamicamente) rimane allocata.

Attenzione: la variabile a cui punta il risultato della funzione deve essere allocata dinamicamente.

```
double *puntatore (double a) {
    double d = a;
    double *r = &d;  // ERRORE: d allocata
    staticamente
    return r;
}

int main () {
    double *pd = puntatore(5.4);
    printf("pd = %p\n",pd);
    printf("*pd = %f\n",*pd);
    return 0;
}
```

In questo esempio la funzione puntatore restituisce il puntatore ad una variabile che viene rilasciata al termine dell'esecuzione della funzione! Il suo indirizzo di memoria potrebbe essere usato successivamente (in quanto è ritornato dalla funzione), ma la zona di memoria corrispondente non è più allocata. Si potrebbero verificare quindi diversi tipi di errori di esecuzione.

4.3.5. Parametri di tipo puntatore a funzione

Un caso particolare di passaggio di parametri di tipo puntatore, è quello in cui il puntatore passato alla funzione punta ad una funzione.

Consideriamo ad esempio una funzione che calcola il valore di una funzione nell'intervallo [a,b] della funzione in ingresso. La sua intestazione sarà:

```
void calcola(double (*f)(double), double primo,
             double ultimo, double inc)
```

oppure semplicemente

```
double calcola(double f(double), double primo,
               double ultimo, double inc)
```

Programma chiamante:

```
risultato = calcola(sin, 0.0, PI / 2, 0.1);
```

All'interno del corpo della funzione calcola possiamo chiamare la funzione alla quale punta f:

```
y = (*f)(x);
```

Il programma completo è il seguente:

pfun.c

```
#include <math.h>
#include <stdio.h>

void calcola(double (*f)(double), double primo,
             double ultimo, double inc);

int main(void) {
    double ini, fin, inc;
    printf("immetti valori (iniziale, incremento,
    finale): ");
    scanf("%lf %lf %lf", &ini, &fin, &inc);
    printf("\n      x        cos(x)"
           "\n   -------    -------\n");
    calcola(cos, ini, fin, inc);
    printf("\n      x        sin(x)"
           "\n   -------    -------\n");
    calcola(sin, ini, fin, inc);
  return 0;
}


void calcola(double (*f)(double), double primo,
             double ultimo, double inc)
{
    double x;
    int i, num_intervalli;
    num_intervalli = ceil((ultimo - primo) / inc);
    for (i = 0; i <= num_intervalli; i++) {
```

```
        x = primo + i * inc;
        printf("%10.5f %10.5f\n", x, (*f)(x));
    }
}
```

4.4. Variabili locali di una funzione

Il corpo di una funzione può contenere dichiarazioni di variabili. Tali variabili vengono dette **variabili locali**. Abbiamo già visto la distinzione tra variabili locali e variabili globali. Dato che le funzioni consentono di definire variabili locali, riconsideriamo due aspetti fondamentali ad esse relativi:

- **campo d'azione** (è una nozione statica, che dipende dal testo del programma)
- **tempo di vita** (è una nozione dinamica, che dipende dall'esecuzione del programma)

4.4.1. Campo d'azione delle variabili locali

Come abbiamo visto, il **campo d'azione** (o **scope**) di una variabile è *l'insieme delle unità di programma in cui la variabile è visibile* (cioè accessibile ed utilizzabile).

Nel caso delle definizioni di funzione, il campo di azione di una variabile locale è il corpo della funzione in cui essa è dichiarata. Cioè la variabile è visibile nel corpo della funzione in cui compare la sua dichiarazione, mentre non è visibile all'esterno.

Questa considerazione deriva dalla regola generale sul campo d'azione delle variabili: una variabile dichiarata in un qualsiasi blocco (istruzione {...}) è visibile in quel blocco (inclusi eventuali blocchi interni), ma non è visibile all'esterno del blocco stesso (vedi Unità 1). Naturalmente, come abbiamo specificato nell'Unità 1, una variabile non può essere utilizzata nel corpo della funzione prima di essere dichiarata.

Nota: il campo d'azione di una variabile è una nozione completamente statica. Infatti esso può essere stabilito analizzando la struttura del programma, senza considerare come il programma si comporta in esecuzione. La maggior parte dei linguaggi di programmazione attualmente in uso adotta questa nozione detta *campo di azione (o scope)*

statico. Pertanto il campo d'azione è un concetto rilevante a tempo di compilazione.

4.4.1.1. Esempio: campo di azione di variabili locali

Consideriamo il seguente programma.

```
int raddoppia (int x) {
    return x*2;
}
void stampa() {
    printf("a = %d\n",a);   //ERRORE a non e' definito
}
int main() {
    int a = 5;
    a = raddoppia(a);
    stampa();
    printf("a = %d\n",a);
}
```

Durante la compilazione del programma sarà evidenziato un errore: nella funzione stampa la variabile a non è visibile (perché definita nella funzione main).

4.4.2. Variabili locali definite static

In C esiste un meccanismo per consentire di definire variabili locali ad un blocco che mantengono il valore tra due esecuzioni successive del blocco stesso. Per ottenere questo effetto occorre usare la parola riservata static nella definizione della variabile.

Esempio: Il programma seguente

static.c

```
#include <stdio.h>

int ricorda() {
    int static c;
    c++;
    return c;
}

int main() {
    printf("ricorda() = %d\n",ricorda());
    printf("ricorda() = %d\n",ricorda());
    printf("ricorda() = %d\n",ricorda());
}
```

stampa

```
ricorda() = 1
ricorda() = 2
ricorda() = 3
```

Si osservi che la variabile statica è differente da una variabile globale (vedi dopo) in quanto essa non è comunque visibile all'esterno del blocco. Inoltre le variabili statiche (a differenza delle variabili locali o globali) vengono sempre inizializzate a zero.

La parola static si può trovare associata anche alle definizioni di funzione. In questo caso il suo significato è di specificare che la definizione della funzione è valida soltanto all'interno del file in cui si trova (vedi successivamente l'alternativa definizione extern).

4.5. Variabili globali

In C le variabili *globali*, cioè definite al di fuori di una definizione di funzione sono visibili in tutte le funzioni. Quindi in ogni funzione sono visibili le variabili dichiarate localmente e le variabili globali.

Esempio:

```
int a = 1; // dichiarazione globale di a
void f1 () {
   printf("In f1\n");
   printf("a = %d\n",a);
// printf("b = %d\n",b); errore in compilazione
   return;
}
int main () {
   int b = 1;
   f1();
   printf("In main\n");
   printf("a = %d\n",a);
   printf("b = %d\n",b);
   return 0;
}
```

4.6. Tempo di vita delle variabili

Il **tempo di vita** di una variabile è il *tempo in cui la variabile rimane effettivamente accessibile in memoria durante l'esecuzione*.

Si distinguono tre casi: variabili locali, variabili globali, variabili statiche.

Una variabile definita all'interno di un blocco di istruzioni ha un tempo di vita pari al tempo di esecuzione del blocco stesso. In particolare, le variabili locali ad una funzione vengono allocate al momento dell'attivazione della funzione (come i parametri formali) e vengono deallocate al momento dell'uscita dall'attivazione. Quindi il tempo di vita di queste variabili corrisponde al tempo in cui viene eseguita la funzione.

Le variabili globali e le variabili statiche invece hanno un tempo di vita pari a tutta la durata dell'esecuzione del programma.

Esempio: tempo di vita delle variabili locali

Consideriamo il seguente programma.

```
int raddoppia (int x) {
    int temp = x*2;
    return temp;
}
void stampa(int b) {
    printf("b = %d\n",b);
}

int main() {
    int a = 5;
    a = raddoppia(a);
    a = raddoppia(a);
    stampa(a);
    printf("a = %d\n",a);
}
```

Durante l'esecuzione del programma la variabile a ha un tempo di vita pari al tempo di esecuzione della funzione main.

Il parametro x della funzione raddoppia (che corrisponde ad una variabile locale) e la variabile locale temp hanno un tempo di vita che corrisponde al tempo di esecuzione della funzione raddoppia. Si noti che ad ogni esecuzione di raddoppia corrispondono in esecuzione diverse istanze di variabili.

Analogamente il parametro x della funzione stampa ha un tempo di vita pari all'esecuzione di stampa.

Si noti, infine, che durante le esecuzioni di raddoppia e stampa la variabile a, rimane in vita anche se non è visibile.

4.7. Modello run-time

Con il termine *modello run-time* si indicano i meccanismi che consentono l'esecuzione dei programmi. In particolare, di seguito viene esaminata la parte del modello run-time che riguarda la gestione delle chiamate di funzione.

A tempo di esecuzione, il sistema operativo deve gestire diverse zone di memoria per l'esecuzione di un programma:

- zona che contiene il codice eseguibile del programma
  - determinata a tempo di esecuzione al momento del caricamento del programma
  - dimensione fissata per ogni funzione a tempo di compilazione
- **heap**: zona di memoria che contiene la memoria allocata dinamicamente
  - cresce e decresce dinamicamente durante l'esecuzione
  - ogni area di memoria viene allocata e deallocata indipendentemente dalle altre
- **pila dei record di attivazione (o stack)**: zona di memoria per i dati locali alle funzioni (variabili e parametri)
  - cresce e decresce dinamicamente durante l'esecuzione
  - viene gestita con un meccanismo a *pila*

4.7.1. Record di attivazione

Quando viene attivata una funzione, viene allocato uno spazio in memoria, detto *record di attivazione*.

Il record di attivazione contiene le seguenti informazioni:

- locazioni di memoria per i parametri formali;
- locazioni di memoria per le variabili locali (se presenti);
- locazione di memoria per memorizzare il valore di ritorno dell'invocazione della funzione (se la funzione ha tipo di ritorno diverso da void);
- locazione di memoria per l'indirizzo di ritorno, ovvero l'indirizzo della prossima istruzione da eseguire nella funzione chiamante.

Tale record viene utilizzato durante l'esecuzione della funzione e viene poi deallocato alla fine dell'esecuzione stessa. Quando il record di attivazione viene deallocato, le locazioni di memoria per le variabili locali e i parametri formali vengono quindi deallocate e il loro contenuto non è più accessibile.

Ad una nuova attivazione della funzione corrisponde una nuova allocazione delle variabili (che non ha nulla in comune con quella delle attivazioni precedenti). Ne segue che ad ogni attivazione della funzione vengono allocate locazioni di memoria diverse per le variabili locali e i parametri formali.

4.7.2. Pila dei record di attivazione

Una *pila* (o *stack*) è una struttura dati con accesso LIFO: Last In First Out = ultimo entrato è il primo a uscire (Es.: pila di piatti).

A run-time il sistema operativo gestisce la **pila dei record di attivazione** (RDA):

- per *ogni attivazione di funzione* viene creato un nuovo RDA in cima alla pila;
- al termine dell'attivazione della funzione il RDA viene rimosso dalla pila.

4.7.3. Esempio di evoluzione della pila dei record di attivazione

Consideriamo le seguenti tre funzioni main, A e B e vediamo cosa avviene durante l'esecuzione della funzione main.

```
int B(int pb) {
/* b0 */ printf("In B. Parametro pb = %d\n", pb);
/* b1 */ return pb+1;
}

int A(int pa) {
/* a0 */ printf("In A. Parametro pa = %d\n", pa);
/* a1 */ printf("Chiamata di B(%d).\n", pa * 2);
/* a2 */ int va = B(pa * 2);
/* a3 */ printf("Di nuovo in A. va = %d\n", va);
/* a4 */ return va + pa;
}
```

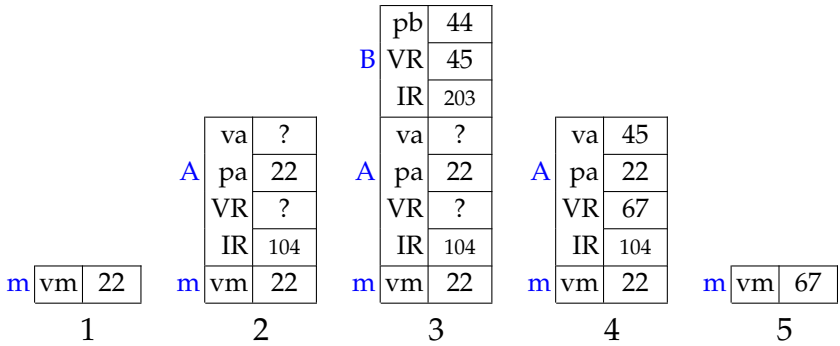
```c
int main() {
/* m0 */ printf("In main.\n");
/* m1 */ int vm = 22;
/* m2 */ printf("Chiamata di A(%d)\n",vm);
/* m3 */ vm = A(vm);
/* m4 */ printf("Di nuovo in main. vm = %d\n", vm);
/* m5 */ return 0;
}
```

Per semplicità, assumiamo che ad ogni istruzione del codice sorgente C corrisponda una singola locazione di memoria.

	main			A			B	
	...			...			...	
100	m0		200	a0		300	b0	
101	m1		201	a1		301	return	
102	m2		202	a2	⇒ B(va*2)	302	...	
103	m3	⇒ A(vm)	203	a3				
104	m4		204	return				
105	return		205	...				
106	...							

```
In main.
Chiamata di A(22).
In A. Parametro pa = 22
Chiamata di B(44).
In B. Parametro pb = 44
Di nuovo in A. va = 45
Di nuovo in main. vm = 67
```

Evoluzione della pila dei RDA (m usato per indicare la funzione main):



Per comprendere cosa avviene durante l'esecuzione del codice, è necessario fare riferimento, oltre che alla pila dei RDA, al **program counter** (PC), il cui valore è l'indirizzo della prossima istruzione da eseguire.

Analizziamo in dettaglio cosa avviene al momento dell'**attivazione** di A(vm) dalla funzione main. Prima dell'attivazione, la pila dei RDA è come mostrato in 1 nella figura di sopra:

1. *vengono valutati i parametri attuali*: nel nostro caso il parametro attuale è l'espressione vm che ha come valore l'intero 22;

2. *viene individuata la funzione da eseguire* in base al numero e tipo dei parametri attuali, cercando la definizione di una funzione la cui segnatura sia conforme alla invocazione (il nome della funzione deve essere lo stesso e i parametri attuali devono corrispondere in numero e tipo ai parametri formali): nel nostro caso la funzione da eseguire deve avere la segnatura A(int);

3. *viene sospesa l'esecuzione della funzione chiamante*: nel nostro caso la funzione main;

4. *viene creato il RDA* relativo all'attivazione corrente della funzione chiamata: nel nostro caso viene creato il RDA relativo all'attivazione corrente di A; il RDA contiene:

   - le locazioni di memoria per i parametri formali: nel nostro caso, il parametro pa di tipo int;
   - le locazioni di memoria per le variabili locali: nel nostro caso, la variabile va di tipo int;
   - una locazione di memoria per il valore di ritorno: nel nostro caso indicata con VR;

- una locazione di memoria per l'indirizzo di ritorno: nel nostro caso indicata con IR;

5. *viene assegnato il valore dei parametri attuali ai parametri formali*: nel nostro caso, il parametro formale pa viene inizializzato con il valore 22;

6. *l'indirizzo di ritorno nel RDA viene impostato all'indirizzo della prossima istruzione che deve essere eseguita nella funzione chiamante al termine dell'invocazione*: nel nostro caso, l'indirizzo di ritorno nel RDA relativo all'attivazione di A viene impostato al valore 104, che è l'indirizzo dell'istruzione di main corrispondente all'istruzione m4, da eseguire quando l'attivazione di A sarà terminata; a questo punto, la pila dei RDA è come mostrato in 2 nella figura di sopra;

7. *al program counter viene assegnato l'indirizzo della prima istruzione della funzione invocata*: nel nostro caso, al program counter viene assegnato l'indirizzo 200, che è l'indirizzo della prima istruzione di A;

8. *si passa ad eseguire la prossima istruzione indicata dal program counter, che sarà la prima istruzione della funzione invocata*: nel nostro caso l'istruzione di indirizzo 200, ovvero la prima istruzione di A.

Dopo questi passi, le istruzioni della funzione chiamata, nel nostro caso di A, vengono eseguite in sequenza. In particolare, avverrà l'attivazione, l'esecuzione e la terminazione di eventuali funzioni a loro volta invocate nella funzione chiamata. Nel nostro caso, avverrà l'attivazione, l'esecuzione e la terminazione della funzione B, con un meccanismo analogo a quello adottato per A; la pila dei RDA passerà attraverso gli stati 3 e 4.

Analizziamo ora in dettaglio cosa avviene al momento della **terminazione dell'attivazione** di A, ovvero quando viene eseguita l'istruzione return va+pa;. Prima dell'esecuzione, la pila dei RDA è come mostrato in 4 nella figura di sopra, (in realtà, la zona di memoria predisposta a contenere il valore di ritorno, indicata con VR nella figura, viene inizializzata contestualmente all'esecuzione dell'istruzione return, e non prima):

1. *al program counter viene assegnato il valore memorizzato nella locazione di memoria riservata all'indirizzo di ritorno nel RDA corrente*: nel nostro

caso, tale valore è pari a 104, che è proprio l'indirizzo, memorizzato in IR, della prossima istruzione di main che dovrà essere eseguita;

2. *nel caso la funzione invocata preveda la restituzione di un valore di ritorno, tale valore viene memorizzato in un'apposita locazione di memoria del RDA corrente*: nel nostro caso, il valore 67, risultato della valutazione dell'espressione va+pa viene assegnato alla locazione di memoria indicata con VR, predisposta per contenere il valore di ritorno;

3. *viene eliminato dalla pila dei RDA il RDA relativo all'attivazione corrente, e il RDA corrente diviene quello immediatamente precedente nella pila; contestualmente all'eliminazione del RDA dalla pila dei RDA, un eventuale valore di ritorno viene copiato in una locazione di memoria del RDA del chiamante*: nel nostro caso, viene eliminato il RDA relativo all'attivazione di A e il RDA corrente diviene quello relativo all'attivazione di main; inoltre, il valore 67, memorizzato nella locazione di memoria VR viene assegnato alla variabile vm nel RDA di main; la pila dei RDA è come mostrato in 5 nella figura di sopra;

4. *si passa ad eseguire la prossima istruzione indicata dal program counter*, ovvero quella appena impostata al passo 1: nel nostro caso, si passa ad eseguire l'istruzione di indirizzo 104, che fa riprendere l'esecuzione di main.