Programmazione

Giovanni Da San Martino

Dipartimento of Matematica, Università degli Studi di Padova
giovanni.dasanmartino@unipd.it
A.A. 2021-2022

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Previuosly on Programmazione

- assegnamento: x=x+y
- esecuzione condizionale:
  - if () then {} else {} // se () allora eseguo {}, altrimenti eseguo {}
    - if (x>=0) { printf(“positivo”);} else {printf(“negativo”); }
  - if () then
    - if (x<0) { x=-1*x; }
- iterazione
  - while () {} //finché () è vera eseguo {}
    - while (x>0) { printf(“positivo\n”); x=x-1; }
  - for (;condizione;) {} // finché (condizione) è vera eseguo {}
    - for (i=0;i<10; i=i+1) { printf(“%d\n”,i); }

Esercizi di Laboratorio

- Capire bene la consegna! Confrontatevi con i compagni a fianco per esserne sicuri
- Elencare una serie di azioni necessarie per risolvere il problema in Italiano (senza restrizioni del linguaggio eseguito)
- Ripetere l'operazione cercando di utilizzare solamente il vocabolario dei comandi che conosciamo (se allora altrimenti, finché), eventualmente suddividendo il problema in sottoproblemi.
- Traducete in codice la soluzione precedente (Dovete conoscere a menadito il significato dei comandi)
- Confrontare la vostra soluzione con quelle dei prof e dei vostri compagni, per capire se alcune parti possono essere migliorate

Correttezza

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Programmazione: Fonti di Errore

- Tipologie di Errore
  - sintassi
  - semantica
- Debug: permette di trovare gli errori di implementazione
- Unit testing : può aiutare per individuare errori logici
- Correttezza: aiuta per dimostrare che il nostro programma fa sempre quello che vogliamo

Errori di Sintassi

- Sintassi: insieme di regole che descrive come si possano costruire “frasi” (programmi) validi
- Errore di sintassi: il compilatore riesce a tradurre il nostro codice se e solo se rispetta la sintassi del linguaggio
- Se non ci riesce la compilazione fallisce e il file eseguibile non viene generato
  - Il compilatore prova a darci informazioni sull’errore (dove si è bloccato e che tipo di problema ha riscontrato)
  - Di solito sono molto precise ed utili, ma a volte (per esempio in casi particolari quando ci si dimentica una parentesi, non indicano dove sia esattamente l’errore)

Warning

- In aggiunta il compilatore analizza il codice per trovare sequenze di istruzioni possibilmente problematiche anche se corrette sintatticamente, i cosiddétti warning

- Es. una variabile utilizzata prima di essere assegnata
  - int x, y; y=x+2; // non si sa quanto valga x!
- Es. int x=3; if (x=2) {printf(“x=5”);} // un assegnamento ha valore di verità vero!

Warning

- Es. for(i=0; i<10; i=i+1); { printf("%d\n",i); }
- Il blocco viene eseguito una volta stampando 10

- Per visualizzare tutti i warning usare:

gcc **–Wall** –o file_eseguibile file_sorgente.c

Errori Semantici

- Un programma può quindi avere errori semantici (ossia di logica: il programma non fa quello che si voleva) che si evidenziano a **run-time** (ossia quando si esegue il programma)
- il problema può derivare
  - dalla non correttezza dell'algoritmo
  - dall'errata implementazione dello stesso

Debug

L'analisi di un programma può essere effettuata tramite

1. l'aggiunta di istruzioni printf in opportuni punti del codice che danno informazioni sulle variabili
2. l'uso di un debugger, ossia un programma che
   - esegue il codice in modo controllato e permette di:
   - eseguire le istruzioni fermandosi dopo ciascuna
   - verificare e cambiare il contenuto delle variabili
   - fermarsi automaticamente quando si verificano certe condizioni

Correttezza dei Programmi

- Come possiamo verificare che il nostro programma calcola ciò che ci aspettiamo?
- Unit tests: calcoliamo a mano una serie di input/output e poi verifichiamo che il calcolatore restituisca, per ogni input, lo stesso output.
  - Un test fallito dimostra che un programma non è corretto
  - Una serie di test passati non significa che il programma sia corretto per ogni input, ma soltanto per quelli testati
- In certi casi potremmo dover fornire evidenza che, per un certo insieme di input, l'output sia corretto. In questo caso si procede quasi come se fosse una dimostrazione matematica.

Funzioni

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Funzioni

- Per semplificare la struttura di un programma complesso è possibile suddividerlo in moduli, sottoprogrammi o, nel gergo del C, funzioni
- Una funzione è una serie di istruzioni che assolvono a un compito preciso (ad es. calcolare se un numero è primo) e a cui è stato dato un nome
- Sintassi della definizione di una funzione:

```
tipo_restituito nomeFunzione (parametri) {
        //definizioni variabili locali
        comandi della funzione
        return
}
```

La scelta del nome della funzione segue le regole per le variabili

Funzioni

- Ogni funzione può essere considerata un piccolo programma isolato dalle altre funzioni
- Una funzione viene invocata scrivendo il suo nome seguito dalle parentesi ()
  - nome_funzione()
- Definire una funzione è come definire un nuovo comando: dopo che il corpo della funzione è stato eseguito, si torna ad eseguire il comando successivo a nome_funzione()

Funzioni


```
void stampa_ciao_mondo() {
        int i; // la visibilità di questa funzione è locale al blocco dove è definita
        for(i=0;i<10; i=i+1) {
                printf("ciao mondo!\n");
        }
}

int main () {
        stampa_ciao_mondo();
        printf("finito\n");
}
```

Funzioni

- Una funzione può restituire un valore (per esempio di tipo int), che può essere utilizzato all'interno del codice come se fosse una variabile di quel tipo. Se una funzione non restituisce niente, si usa il tipo void. Es.

```
int numero_gatti() {
        return 44; //comando per restituire un valore alla funzione chiamante
}

int main () {
        int x = numero_gatti();
        printf("Ci sono %d gatti", numero_gatti()+3);
}
```

Funzioni

Vantaggi della programmazione modulare:

- il programma complessivo ha un maggior livello di astrazione perché i moduli “nascondono” al loro interno i dettagli implementatativi delle funzionalità realizzate
- il codice per ottenere una certa funzionalità viene scritto una volta sola e viene richiamato ogni volta che è necessario
- il codice complessivo è più corto
- essendo più piccoli, i moduli sono più semplici da implementare e da verificare
- il codice di un modulo correttamente funzionante può essere riutilizzato in altri programmi

Parametri di Funzioni

- Per rendere le funzioni più flessibili ed interessanti, si ha la possibilità di passare, all'interno delle parentesi tonde, dei parametri sui quali la funzione possa operare.
- Nella definizione della funzione, per ogni parametro bisogna indicare il tipo

```
int successivo(int n) { //parametro formale della funzione
        return n+1;
}

int main () {
        int x=2;
        printf("x+1=%d\n", successivo(x)); //x=parametro attuale della funzione
}
```

Passaggio di Parametri

- Gli argomenti che la funzione riceve dal chiamante devono essere memorizzati in opportune variabili locali alla funzione stessa dette parametri
- I parametri sono automaticamente inizializzati con i valori degli argomenti
- Nell'esempio precedente:

```
int successivo(int n) { return n+1; }
```

quando invoco successivo(x), potete immaginare quello che succede come:

```
int successivo() { int n=x; return n+1; }
```

- in questo caso diciamo che il parametro è passato per valore

Passaggio di Parametri

- se ci sono più parametri, i valori dei parametri attuali vengono assegnati ai parametri formali in ordine
- i parametri della funzione si comportano come variabili locali.

```
int somma(int x, int y) {
        return x+y; //x=4, y=5
}
int main(void) {
        int a=4,b=5, somma;
        somma = somma(a,b);
}
```

Parametri

- Argomenti e parametri devono corrispondere in base alla posizione e al numero (almeno per le funzioni che definiremo noi)
- I nomi dei parametri sono indipendenti dai nomi delle variabili del chiamante
- Se la funzione non richiedere parametri è preferibile indicare void tra le parentesi
- In memoria i parametri sono del tutto distinti e indipendenti dagli argomenti, quindi cambiare il valore di un parametro non modifica l'argomento corrispondente.

Parametri

- La visibilità di una funzione indica dove essa può essere richiamata:
- si estende dal punto in cui viene definita fino a fine file (quindi può essere utilizzata solo dalle funzioni che nello stesso file seguono la sua definizione)
- vedremo come ovviare a questa limitazione:
- Per ovviare a questa limitazione, basta aggiungere il prototipo di una funzione (la prima riga con l'aggiunta del ;)
  - int somma(int b, int e);
- Adesso è possibile invocare la funzione dalla riga successiva del prototipo (quindi conviene aggiungere il prototipo subito dopo gli include
- Notate che il prototipo fornisce tutte le informazioni necessarie a chi voglia utilizzare la funziome

Passaggio di Parametri

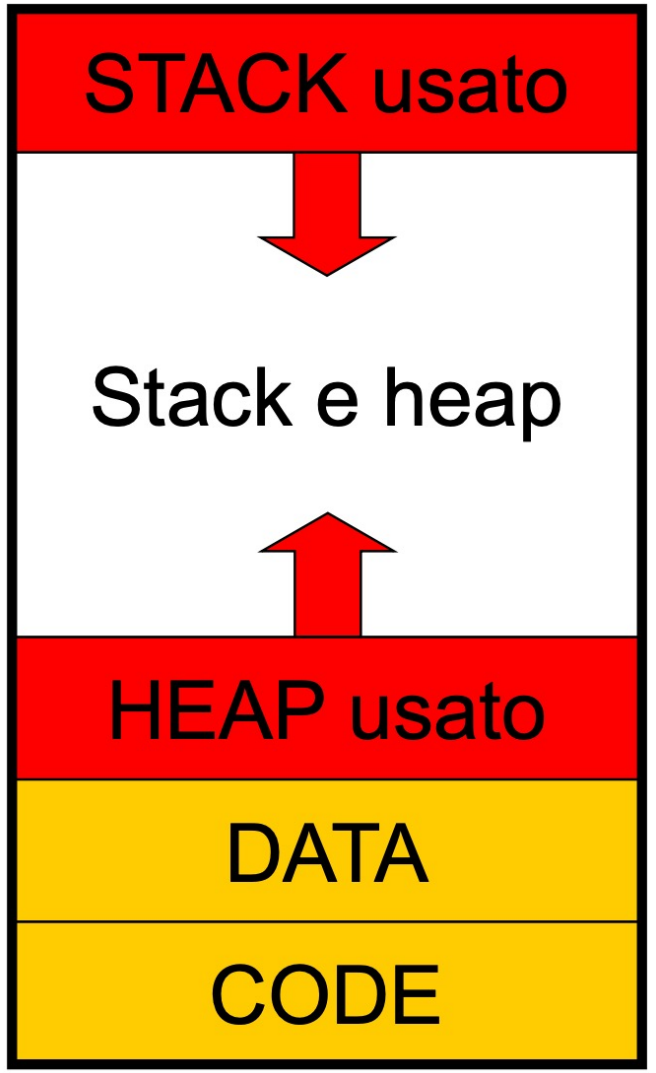

```
int somma(int x, int y);

int main(void) {
        int a=4,b=5, somma;
        somma = somma(a,b);
}

int somma(int x, int y) {
        return x+y; //x=4, y=5
}
```

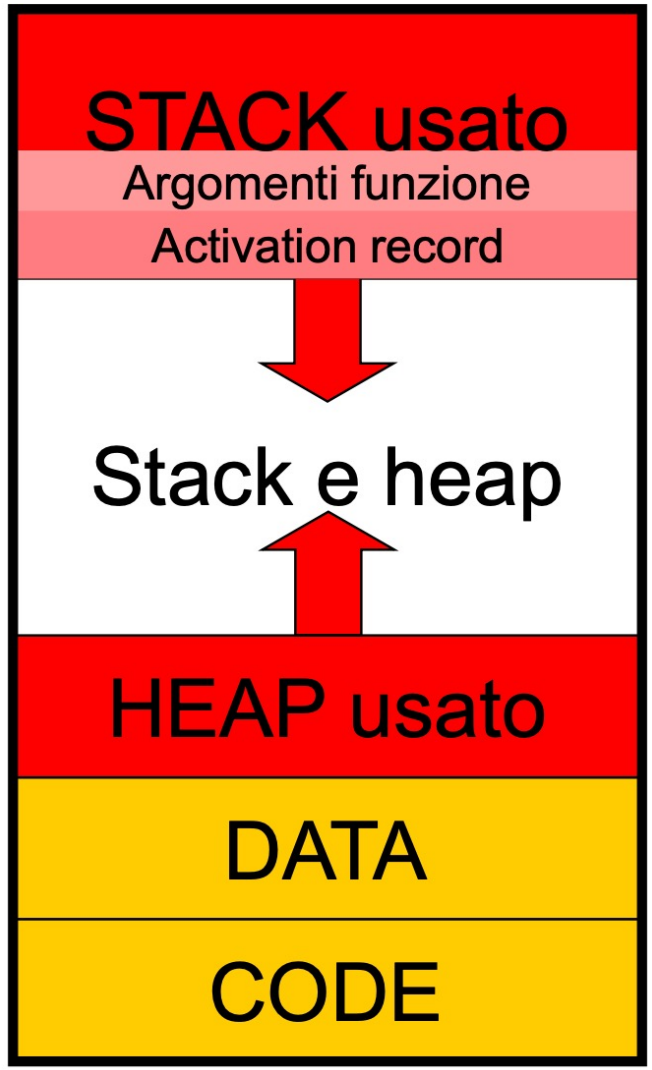
Chiamata di Funzione

- Il programma compilato è costituito da due parti distinte:
- segment del codice: codice eseguibile
- segmento dei dati: costanti e variabili
- Quando il programma viene eseguito, il Sistema Operativo alloca spazio di memoria per:
- il segmento del codice (CS)
- ilsegmento dei dati (DS)
- lo stack e lo heap (condivisi)



Chiamata di Funzione

- Il codice di una funzione è in code
- i parametri e le variabili locali di una funzione vengono allocati nello stack (pila)
- vengono prima copiati i valori dei suoi argomenti e poi vi viene allocato un
- Activation Record (anche detto stack frame) in cui sono allocate le variabili locali della funzione e altro
- Quando la funzione termina, l'AR e gli argomenti vengono rimossi dallo stack che quindi ritorna nello stato



Chiamata di Funzione

- Nell'Activation Record viene anche memorizzato **l'indirizzo di ritorno dalla funzione**: l'indirizzo di memoria che contiene l'istruzione del modulo chiamante da cui continuare l'esecuzione dopo che la funzione è terminata
- Queste operazioni di allocazione e deallocazione di spazio sullo stack e in generale il meccanismo di chiamata e ritorno da una funzione richiedono tempo