18 settembre 2019

Storia dell'informatica in breve

Partita dall'elettronica. Inizia dopo la WW2 (Turing - rompere enigma). Prima funzionava a valvole che si rompevano ogni 2 minuti. Il PDP11 ha rivoluzionato la scena: più piccolo, architettura ~ client-server.

A causa delle restrizioni, i programmatori provavano a usare il meno risorse possibili, non come si fa oggi. Ciò ha causato problemi come il Millennium Bug (venivano salvate le ultime due cifre delle date per risparmiare spazio e l'anno 2000 avrebbe causato overflow e errori in applicazioni critiche).

Il secondo salto avviene con la creazione del Personal Computer (PC). Poi si passa alla W.W.W e infine, al giorno d'oggi, ai Social Network. Con il web inizia il problema del Big Data (raccolta di dati e la loro vendita che portano a vere e proprie guerre cibernetiche).

Visto che l'hardware diventa sempre più potente, anche i programmi diventano molto più complessi. Ciò porta l'informatica a sviluppare dei criteri per creare programmi più semplici e mantenibili (Dennis Ritchie - Unix).

Che cos'è l'informatica

E' uno **studio di algoritmi** che descrivono e trasformano **informazioni**. Per **algoritmo si intende un programma che elabora dei dati.**

Lo sviluppo del software non segue quello classico della manifattura: il costo si applica tutto nello sviluppo iniziale (1a copia). Ci sono molti modelli economici (esempio: modello open-source (RedHat))

Programma corso

Reperibile su beep.

- Logica, interi e reali in binario + operazioni binarie
- Sintassi, struttura programma monomodulo, astrazione dati e flusso (C)
- Sottoprogrammi e ricorsione
- Strutture dati dinamiche e persistenti

Diviso in lezione e laboratorio. Il laboratorio sarà diviso in due squadre (sarò squadra 1). Lezione e laboratorio si svolgono in parallelo.

Libri (optional)

- Kim N king: Programmazione in C
- P. Deitel, H. Deitel: Il linguaggio C
- Jeri R. Hanly, Elliot B. Koffmann: Problem solving e programmazione in C
- Ira Pohl, Al Kelley: C: didattica e programmazione

Materiale

Reperibile su beep. Anche le esercitazioni per il laboratorio sono lì.

Esame

Prova scritta con discussione (la discussione non è un esame a parte). Niente prove intermedie. Gli orali solo su richiesta esplicita del docente in casi speciali. Non ti puoi iscrivere oltre la data ufficiale. Il laboratorio non influenza il voto d'esame.

19 settembre 2019

Linguaggi di programmazione

Diversi linguaggi di programmazione esprimono diversi concetti con diversi livelli di efficienza (come le vere lingue). Diversi linguaggi hanno anche diversi scopi.

Piccola cronologia:

- FORTRAN (54/57)
- LISP (59)
- Algol, COBOL (1960)
- BASIC (1962)
- ...
- Java (1995)
- Python (1996)

Un linguaggio di programmazione può essere di **alto e basso** livello. Tanto **più è di basso, più è vicino all'hw. Più è alto più si distacca dall'hw.** Un linguaggio può essere **generale o specifico** (C vs. SQL). Può essere **compilato o interpretato** (C vs. Python). Un linguaggio può essere **imperativo o dichiarativo** (**dichiarativo**: si **esprime l'obbiettivo** e la macchina lo traduce nell'imperativo; **imperativo**: si **esprimono i vari passaggi per ottenere il risultato**).

Il C è di **alto livello, generale, compilato e imperativo.** Esso è inoltre la base di molti altri linguaggi e ha influenzato tantissimi altri.

In un linguaggio di programmazione è importante **lo stile**. *The cost of software maintenance increases with the square of the programmer's creativity*.

Notazione Backus Naur Form (BNF)

Delle regole per definire la sintassi.

- **elementi terminali**: termini che non possono essere cambiati dalle regole (parole chiave del linguaggio ecc.)
- **elementi non terminali**: termini che saranno sostituiti (es. `{}` , gruppo)
  - Metasimboli: es. `{}n+`

Le **regole contengono una o più espressioni**, separate da `|` che indica scelta. Regola: `non terminale ::= terminali / nonterminali / metasimboli`

Es.: `identifier ::= letter {letter | digit}0+`

Primo programma in C

Sintassi e semantica del C

La sintassi deve essere perfetta sennò non compila. La semantica (cosa il programma *significa*) è controllata dal programmatore.

Sintassi in BNF:

```
indentifier ::= letter {letter|digit}0+
expression ::= constant | ... | relational_expr | logical_expr | arithmetic_expr
relational_expr ::= expression relational_op expression
logical_expr ::= expression relational_op expresison
relational_op ::= == | != | > | < | >= | <=
logical_op ::= || | && | !
```

- **Case-sensitive**
- **Non ci possono essere due identificatori uguali**

- **Parole chiave (es. if) e built-in sono riservati (printf)**
- **Non ci possono essere due identificatori per lo stesso elemento**

Priorità operatori (Tabella 2.1 Kernighan-Ritchie)

Schema semplificato di un programma

```
program ::= directive_part {global_declarative_part}_opt
    {function_definition}0+
    int main () { {local_declarative_part}0+ executable_part }
directive_part ::= ... | #include idetifier // es. #include <stdio.h>
global_declarative_part ::= constant_declarations | type_declarations | variable_declarations
variable_declarations ::= {type_specifier variable_identifier}1+
executable_part ::= statement_sequence
statement_sequence ::= single_statement | {{single_statement}1+}
single_statement ::= stuff ;
```

Istruzioni IO base

- `printf(string, expr)`
  - `string` : stringa racchiusa tra `"`
  - `expr` : espressione da stampare coerente con il formato di stampa (es. `%d`)
- `scanf(string, pointer)`
  - `string` : specifica come interpretare il messaggio con un indicatore di formattazione
  - `pointer` : `&variable_identifier`
    - `&` restituisce l'indirizzo di memoria della variabile
- `int getchar(void)` : legge un carattere da stdin e lo restituisce come intero
- `void putchar(int c)` : scrive su stdout il carattere `c`

Non verrà controllata la correttezza dell'input (tipo corretto e rispetto dei limiti dell'intero).

Programma

Vedi `./src/first.c` .

Programmazione strutturata

Nata a metà degli anni 60'. **Costruisce il programma su alcuni elementi prefabbricati:**

- **sequenza**: lista di istruzioni
- **selezione binaria** (if): se la condizione è vera, eseguire una determinata sequenza se no leggi un'altra
- **ciclo** (while): ripeti una sequenza finché una condizione è vera

Il **teorema di Bohm-Jacopini** dimostra che bastano queste 3 strutture per costruire qualsiasi algoritmo.

Istruzione condizionale

`if (logical_expr) statement_sequence {else statement_sequence}_opt` . Esempio:

```
if (a !=b) {
    ...;
} else {
    ...;
}
```

Le varie istruzioni condizionali possono essere combinate (nest, `else if`).

ATTENZIONE: In caso di assenza delle parentesi graffe, `else` può essere ambiguo. Il compilatore risolve l'ambiguità associando `else` a `if` più vicino.

```
/*
 * ambiguo: è l'else del primo o del secondo?
 * Il compilatore decide: del secondo
 */
if (cond) if (cond) ...; else  ...;
/* non ambiguo: l'else è del secondo */
if (cond){ if (cond) ...; else ...; }
```

La `logical_expr` può essere anche in contenuto di una variabile: 0 è False, 1 True.

Osserva:

```
if (x = y) ...;
// a causa dell'ordine degli operatori, l'espressione sopra si traduce in:
x = y; if(x) ...;
```

Istruzione ciclica

`while (logical_expr) statement_sequence` . Esempio:

```
while (a == b) {
    ...;
    ...;
}
```

20 settembre 2019

Asincronismo tra `scanf` , `getchar` , `printf` e `putchar`

`scanf` usa un un buffer per catturare l'input dell'utente. Se il buffer è vuoto, aspetta finché l'utente premerà `\n` . Il buffer può essere popolato anche prima della chiamata `scanf` : dopo un prompt con `printf` l'utente potrà inserire i vari caratteri anche se tra la `scanf` che li parserà ci saranno altre istruzioni.

Lo stesso comportamento lo ha `getchar` : legge dal buffer ogni volta un carattere. `getchar` si ricorda la posizione dell'ultimo carattere letto e legge sempre quello successivo.

Analizzerà la serie di caratteri interpretandola in base alla stringa di formato passata. Dopo che ha finito, rende libero il buffer.

`grocery.c`

Chiedi quanto devi comprare di qualcosa, quanto costa e calcola la un eventuale sconto. Scrivi a video il totale da pagare.

Qual è il miglior modo per determinare lo sconto? Vedi `grocery-tests`

Programmazione strutturata

[...]

Istruzione ciclica

Sintassi

```
while (logical_expr) statement_sequence
```

Comportamento

L'istruzione, finché `logical_expr è vera` , esegue la `statement_sequence` . La `logical_expr` deve essere calcolabile prima dell'entrata nel ciclo. La `logical_expr` viene valutata prima dell'entrata nel ciclo: se è falsa alla prima entrata la `statement_sequence` non viene mai eseguita.

Sono scoraggiati l'uso di `while(0), while(1), break` .

Esempio

```
/* Leggi una riga fino a newline e conta i caratteri (escluso new-line) */
int c, n;
int main(void){
    c = getchar()
    while (c != '\n'){
        numero++;
        c = getchar();
    }
    printf("Caratteri: %d", n)
}
```

Istruzione condizionale multipla (non necessaria)

Sintassi:

```
switch (expression) {
    case valore: statement_sequence
    ...
    case valore-n: statement_sequence
    {deafult: statement_sequence}_opt
}
```

Proprietà

- I valori devono essere costanti.
- `expression = valore` : viene eseguito il case e tutti quelli dopo. Interrompere il case con `break`
- `expression != nessun valore` : viene eseguito `default` se c'è
- Non possono esserci valori uguali
- La `statement_sequence` può non avere le graffe

Istruzione ciclica (a condizione finale)

Sintassi

```
do {statement_sequence} while (logical_expr)
```

Comportamento

Uguale a while solo che inverte l'ordine di esecuzione e valutazione: la condizione viene valutata solo dopo le istruzioni. Questo permette di modificare la condizione nel ciclo prima della sua iniziale.

Istruzione (for)

Sintassi

```
for (expression1; expression2; expression3) {statement_sequence}_{opt}
```

Tutte le tre le `expression` posso essere omesse. Avremo un ciclo infinito:

```
for ( ; ; ) {statement_sequence}_opt
```

Un `for` infinito che non fa nulla è: `for( ; ; );`

Semantica

```
expression1;
while (expression2) {
    {statement_sequence}_{opt}
    expression3
}
```

Comportamento

La `expression1` inizializza variabili utili da usare nella `expression2` . La `expression2` è la `logical_expr` già vista negli altri cicli. La `expression3` è eseguita alla fine di ogni ciclo.

Il ciclo `for` viene usato quando so il numero di iterazioni: uso `expression1` per inizializzare il contatore, la controllo con `expression2` e incremento con `expresison3`

Sono scoraggiati l'uso di cicli infiniti e `break` .

Istruzioni di salto

1. `break` : esce da `for, while, do-while, switch` . Da evitare nei cicli, da usare nel `switch`
2. `continue` : salta alla prossima iterazione del ciclo
3. `goto X X: ..` : **MAI**

Astrazione dei dati

Rappresentati in diversi modi:

- In linguaggio macchina: `00001111`
- in Assembly `MEM: RES 1`
- in C: **variabile**

La variabile ha 5 caratteristiche:

- ha un nome
- ha un tipo (dominio e operazioni)
- non ha valore iniziale
- modalità di allocazione e tempo di vita: le variabili dichiarate nella `global_declarative_part` sono allocate insieme al programma Il tempo di vita delle variabili dichiarate nella `global_declarative_part` sono utilizzabili fino alla fine dell'esecuzione del programma
- scope: per le variabili dichiarate nella `global_declarative_part` sono visibili in ogni parte del programma

25 settembre 2019

Astrazione dei dati

Una variabile è tipizzata, quindi viene allocato uno spazio che può contenere solo il tipo di variabile dichiarato.

Il tipo

Dire che una variabile ha un tipo è in verità specificare 2 cose:

1. il dominio dei valori
2. il set di operazioni

Perché si tipizza? Aiuta nella verifica del buon uso delle variabili e nel sapere a priori quanto spazio serve.

Esistono 2 tipi:

- i tipi semplici
- i costruttori di tipo: permettono di creare nuove strutture complesse partendo dai tipi base.

> Esempio di costrutto è l'array (vettore).

Conversione tra tipi

Può avvenire in tre casi:

1. Espressione con elementi eterogenei: le variabili di tipo più piccolo vengono promosse a variabili di tipo più grande (`int + float => float`)
2. Assegnamento eterogeneo da espressione: il risultato dell'espressione viene convertito nel tipo della variabile

```
int i;
i = float + int // conversione con troncamento
```

3. Casting esplicito tramite l'operatore di cast `cast ::= (type) expression`

```
float a;
int b,c;

a = (float) b/c;
```

I tipi semplici (built-in)

- void: nulla
- intero: diverse definizioni:
  - `short` : intero "corto" (vedi `limits.h` per i limiti)
  - normale (vedi `limits.h` per i limiti)
  - `long` : intero "lungo" (vedi `limits.h` per i limiti)
  - `signed` : intero positivo o negativo (vedi `limits.h` per i limiti)
  - `unsigned` : intero solo positivo (vedi `limits.h` per i limiti)

  Operazioni: `+ - * \ = == < > <= >= ++ --`
- numero reale: diviso in `float` e `double` . Può essere scritto con la notazione `7E+20`
  - `float` : usa 32 bit
  - `double` : usa 64 bit

  Operazioni: le stesse dell'intero.
- carattere: codifica un intero che corrisponde a un carattere in ASCII

26 settembre 2019

Tipi semplici

[...]

- carattere: 8 bit di cui vengono usati solo 7. Viene usata la codifica ASCII.

ASCII

Set abbastanza ristretto di caratteri. Ogni carattere è associato a un numero di 1 byte.

Esempio di manipolazione di caratteri:

```
#include <stdio.h>

char c;

int main(void)
```

```
{
    printf("Carattere: ");
    scanf("%c", &c);

    printf("\nMaiuscola di %c = %c (ASCII %d)\n", c, c - ('a' - 'A'), c);

    return 0
}
```

`'a' - 'A'` ottiene la distanza tra maiuscola e minuscola nella codifica usata (di solito ASCII).

Algoritmi

Prima si deve analizzare il problema e capirlo. Questo, oltre che capire cosa devo fare, devo anche cercare di analizzare eventuali dati problematici o significativi. Dopo di ciò si può pensare all'algoritmo risolvente. Non esiste solo un algoritmo risolvente. Bisogna, quindi, valutare le varie soluzioni e scegliere quella più adatta. IMPORTANTE: l'esecutore non può verificare la correttezza dell'implementazione. Per verificare gli algoritmi si usano i `test`. Dopo aver verificato la correttezza dell'algoritmo, si può passare all'implementazione in un dato linguaggio.

L'algoritmo implementato deve avere delle caratteristiche.

- ogni azione non è ambigua
- ogni istruzione non è infinita
- l'esecuzione ordinata delle azioni termina dopo un numero finito di passi.

Diagrammi di flusso

I rettangoli rappresentano un'azione atomica che il mio esecutore capisce. Il fluire è rappresentato da delle frecce che collegano le varie istruzioni. Il rombo rappresenta una istruzione condizionale.

Il calcolatore

DIgressione su come funziona un computer.

Macchina di Von Neumann (1946)

Tutto è collegato a un `bus`. Gli elementi fondamentali sono:

- La `CPU`: il capo di tutto
- La memoria centrale (`RAM`): memoria dove vengono inserite le istruzioni e i dati
  In aggiunta ai due componenti fondamentali, ci sono le periferiche. Le periferiche possono essere dischi, mouse, tastiera, video terminale e altri. Le periferiche sono molto più lenti di `CPU` e `RAM`. Le periferiche, inoltre, hanno una interfaccia per comunicare con il `bus` di sistema, ossia una 'strada' che mette in comunicazione i vari componenti del computer.

Memoria centrale

La memoria è un insieme di 'parole' legate in sequenza una all'altra. Ogni parola ha una dimensione `h` (prenderemo in considerazione una dimensione di 16 bit).

La memoria ha l'abilità di leggere e scrivere una parola di memoria. Per fare ciò si aiuta con i 'registri' della `CPU`, piccole memorie nella CPU.

CPU

La CPU fa tutto. Si aiuta con dei registri dove vengono salvate tutte le informazioni necessarie al suo funzionamento. I registri sono:

- `DR` (Data register): registro dei dati. Grande quando una parola di memoria (16 bit). I dati che servono per l'operazione della CPU vengono spostati dalla memoria centrale nella `DR` per essere elaborati. I dati elaborati vengono, quindi, spostati di nuovo dalla `DR` alla memoria centrale

- `AR` (Address register): capire l'indirizzo della parola su cui operare. Deve bit a sufficienza per rappresentare in modo ogni indirizzo di memoria in modo univoco: se la memoria ha 1024 indirizzi, io avrò bisogno di una `AR` di 10 bit (0-1023) per poterla gestire
- `PC` (Program counter): deve memorizzare l'indirizzo della prossima (o corrente a seconda dell'architettura) istruzione da eseguire. La CPU, infatti, non vede tutto il programma, ma solo quella che sta eseguendo e massimo quella dopo.

Il bus

Nel bus circolano le istruzioni che la CPU manda alla memoria centrale. Ci sono 3 'sotto bus':

- uno per trasferire dati: scambio dati tra CPU, memoria e periferiche
- uno per trasferire indirizzi: scambio di indirizzi di memoria dalla AR per individuare i componenti/sotto-componenti
- uno per trasferire istruzioni: scambio di istruzioni di controllo tra CPU e altri componenti

La larghezza del bus dipende dal tipo di informazioni che ci devono passare.

Assembly basics

Istruzioni base:

```
[W:] ADD X,y     Somma X e Y
[W:] MOV X,Y     Sposta il X in Y
[W:] BR I        Salta a I
[W:] BREQ X,I    Se il contenuto di X = 0, salta a I
[W:] EXIT        Termina l'esecuzione
[W:] READ X      Leggi da tastiera e metti li risultato X
[W:] WRITE X     Scrivi a video X

(Pseudo istruzioni)
[W:] RES N       Alloca n parole di memoria
[W:] END [X]     Posizione della prima istruzione (aggiorna il PC)
```

Le istruzioni possono prendere un valore simbolico `X` che rappresenta il contenuto dell'indirizzo di memoria associato a `X`. Se viene usato `#X`, invece, viene passato l'indirizzo di memoria associato a `X`:

```
|X:  0   10|    MOVE X,Y     |X:  0   10|
|Y:  1   20|                 |Y:  1   10|

|X:  0   10|    MOVE #X,Y    |X:  0   10|
|Y:  1   20|                 |Y:  1   0 |
```

L'assembly deve essere ulteriormente tradotto in un linguaggio che la macchina capisce direttamente. Una istruzione viene tradotta in una parola di memoria. Prendiamo per esempio una parola da 16 bit. La parola viene divisa in:

- Codice operativo dell'operazione: il codice che la macchina assegna all'operazione voluta (4 bit)
- Modificatore del primo operando: Cancelletto o non cancelletto (2 bit)
- Primo operando: Operando dell'operazione. Se il cancelletto non c'è, viene preso il valore all'interno del valore di memoria passato, sennò è letterale. (4 bit)
- Modificatore del secondo operando: Stessa cosa del primo modificatore (2 bit)
- Secondo operando: Stessa cosa del secondo modificatore (4 bit)

Costanti in C

Sono dei valori che non cambiano nel corso del programma. Le costanti però, possono essere sovrascritte:

```
int a = 10;         // la mia costante
[...]
a = 12;             // è permesso, ma non voglio che succeda
```

Le soluzioni al problema sono due:

- usare la parola chiave `const` :

```
const int a = 10;   // la mia costante, read-only
[...]
a = 12;             // ERRORE di compilazione
```

- usare la direttiva `#define` del prepocessore:

```
#define a 10
[...]
a = 12;             // ERRORE
```

Il `const` è scoraggiato perché la variabile viene lo stesso allocata in memoria e, quindi, potrà essere modificata con la forza (`setmem`). La direttiva del prepocessore, invece, non comporta la creazione di una nuova variabile: il prepocessore (il primo step nella compilazione) risolve tutte le `#define` sostituendole con i loro valori.

2 ottobre 2019

Assembly basics

Al livello di linguaggio macchina, il computer non può capire quale sia la prima istruzione. Per questo il programmatore deve specificare a monte con l'istruzione `END` la prima istruzione in un contesto in cui è possibile capire qual è.

CPU nel dettaglio

La parte più importante è la `CU` , l'unità di controllo, che verrà regolata dal `clock` . Un'altra unità è la `ALU` , l'unità aritmetico logica. La CPU contiene alcune memorie con dei dati di utilizzo frequente (la `cache`). Delle memorie più specializzate sono i registri. Esistono vari tipi di registri:

- PC (già visto)
- INTR (Registro interruzioni)
- CIR (Registro istruzione corrente)
- SR (Registro di stato): contiene variabili utili al programma per capire se tutto è andato a buon fine o c'è stato un errore.
- AR (Registro indirizzi)
- DR (Registro dati)

Microprogramma e CU

La `CU` è tipo un piccolo calcolatore: esegue un piccolo programmino da una memoria di sola lettura (ROM). Questa ROM, a differenza della RAM, non perde il proprio stato una volta persa la corrente. Questo piccolo programmino si chiama microprogramma. Il microprogramma è stato ideato nel 1953 da Wikes e fu implementato nel 1964 nel IBM 630.

Il microprogramma, sostanzialmente, è un loop infinito che, finché il computer è acceso, esegue sempre le stesse operazioni. E' diviso in tre fasi che hanno ognuno un obiettivo logico:

1. Fase di fetch: viene caricato in memoria il programma e viene caricato nell'AR l'indirizzo della prossima operazione. Viene allora preso dalla RAM l'istruzione e viene caricata nel DR. L'istruzione viene, allora, caricata nel CIR
2. Fase di decodifica: vengono letti i primi 4 bit dell'istruzione nel CIR. In base all'istruzione vengono gestiti i parametri. Per esempio una `ADD` fa:
    i. Se non c'è il cancelletto, metti l'indirizzo nell'AR, leggi da RAM, carica il ricavato nel DR. Sposta poi il dato dal DR in un altro registro (cache)
    ii. Successivamente, rifetch il secondo dato (1).

    Una `BR` , invece, fetcherà l'indirizzo come la `ADD` nella 1, ma al posto di caricarla nella cache, la carica nel PC
3. Concludi: se tutto è andato a buon fine, incrementa il PC (o salta in caso di istruzione di salto)

Ogni operazione è suddivisa in molte operazioni atomiche (microistruzioni). Ogni microistruzione impiega 1 ciclo del clock.

Correttezza e efficienza

I MHz indicati nelle specifiche tecniche rappresenta la velocità di punta del computer: quante microistruzioni possono essere eseguite nell'unità di tempo. Per misurare la velocità delle varie macchine, sono stati creati dei programma campioni (benchmark) con cui vengono comparate le macchine.

La toolchain della compilazione:

1. Scrittura della sorgente con un editor di testo (vim o altri)
2. Prepocessing: le direttive che iniziano con `#` vengono sostituite per creare un nuovo file sorgente
3. Compilazione: l'output del prepocessore viene passato al compilatore che traduce il programma in assembly e poi (assembler) in binario. L'output è un file oggetto. In questa fase vengono segnatali eventuali errori di sintassi
4. Linker: risolve i riferimenti a librerie esterne e rende il file oggetto creato dal compilatore eseguibile. L'output è un file eseguibile (.out/.exe). In questa fase vengono segnalati eventuali riferimenti non risolti
5. Esecuzione: vengono segnalati eventuali errori di esecuzione (Stackoverflow, Segfault)

9 ottobre

Operatore `&` e `*`

L'operatore `&` ridà l'indirizzo in memoria di una variabile.

L'operatore `*` ridà il contenuto di un indirizzo in memoria di una variabile.

Funzione `sizeof`

La funzione `sizeof` ridà la dimensione in byte del proprio argomento.

Enum

I tipo `enum` indica dei valori accettabili da un tipo. Dietro le quinte sono numeri, i nomi sono solo simbolici. Per ciò l'ordine di definizione conta: gli elementi definiti prima sono considerati 'minori' rispetto ai successivi.

I tipi composti

Le strutture dati possono essere costruite attraverso i costruttori di tipo. Per definire un tipo si usa:

```
typedef old new
```

Dove `old` è la descrizione di come è fatta la struttura e `new` è il nome del nuovo tipo che voglio definire. Es:

```
typedef int TipoSalario;            // int adesso può essere sia int che TipoSalario

typedef enum {TRUE, FALSE} Tbool;
```

Costruttori di array

L'array è un aggregato di elementi dello stesso tipo ordinato. E' a dimensione fissa decisa ahead-of-time ed è salvato in memoria centrale. Permette l'accesso posizionale agli elementi. GLi elementi sono indicizzati a partire dallo 0.

VLAs

Il C99 permette l'utilizzo di array a lunghezza variabili. La loro allocazione avviene a runtime. FUORI PROGRAMMA.

Definizione

Sintassi per la definizione:

```
#define DIM 10

[...]

int V[DIM] = {1,2,3,4,5,6,7,8,9,0};
int S[DIM];

[...]
```

La dimensione deve essere una espressione costante (macro o costante). Gli elementi all'interno dell'inizializzazione vengono aggiunti in modo ordinato all'array.

Allocazione in memoria

Gli elementi sono tradotti tutti uno in fila all'altro. Tradotto in Assembly sarebbe una roba del genere:

```
V: RES 10 # (V è l'indirizzo della prima istruzione)
```

Come il tutto è disposto in memoria:

```
    ...
V: |   | V[0]
   |   | V[1]
    ...
   |   | V[8]
   |   | V[10]
    ...
```

Il nome dell'array è un sinonimo dell'indirizzo del primo elemento.

Accesso

Si può accedere agli elementi dell'array tramite l'indice del valore. L'indice è un intero e può essere una costante, una variabile o il risultato (int) di espressione.

L'accesso è sempre possibile: il C non controlla lo sconfinamento.

Assegnamento

Assegnare un vettore a un altro vettore per copiarne il contenuto non è ammesso. Di conseguenza scanf e printf non possono visualizzare l'intero array in una sola istruzione (ad eccezione delle stringhe). Anche fare `V == S` è inutile in quanto sempre falso perchè testerà l'eguaglianza dei due indirizzi, che non avverrà mai.

Costruttore RECORD

Un record è un aggregato di valori eterogenei con nome. E' a dimensione fissa ed è salvato in memoria centrale. Permette l'accesso per nome agli elementi.

Costruttore RICORSIVO

E' un aggregato di elementi dello stesso tipo costruito ammettendo che un tipo possa contenere riferimenti a componenti dello stesso tipo. Non ha dimensione fissa. Permette l'accesso sequenziale agli insiemi.

11 ottobre

Tipi composti

[...]

Array

[...]

Matrici bidimensionali

Le matrici vengono caricate in memoria 'per riga': vengono caricati gli elementi della prima riga, seguiti da quelli della seconda ecc... . Ciò può essere anche sfruttato per inizializzarli per righe.

```
typedef int A[2];   // vettore di due interi
A B[3];             // un vettore B di tre elementi di tipo A

int B[3][2];        // una matrice di 3 righe e 2 colonne
                    // un vettore B di 3 elementi, con in ogni elemento un
                    // array di 2 elementi

int B[3][2] = {{1,2}, {3,4}, {5,6}};
int B[3][2] = {1,2,3,4,5,6};
```

Record

Si può dichiarare in 3 modi:

```
// 1 crea struct 'S' di tipo 'struct S'
struct S {
    int a;
    int b;
};
struct S var1, var2;

// 2 crea un nuovo tipo 'S' definito dallo struct
typedef struct {
    int a;
    int b;
} S;
S var1, var2;

// 3 Crea due struct 'S' contenuti nelle variabili
struct S {
    int a;
    int b;
} var1, var2;
```

Allocazione

Verrano allocate uno in fila all'altro. Pseudo assembly:

```
V.A: RES 1
V.B: RES 1
```

Accesso

Si usa la notazione:

```
{var}.{nome atributo}
```

Si può accedere a un campo alla volta.

Assegnamento

L'espressione `v = z` è permesso se sono dello stesso tipo. Vengono assegnate elemento per elemento.

Testare l'uguaglianza tra due struct, invece, è compile time error.

16 ottobre

Gestione della dinamicità degli array

- Usare un terminatore convenzionale (valore non utile). esempio:
  - le stringhe sono vettori di caratteri delimitate dal carattere nullo `\0`
  - il carattere `EOF` viene usato per terminare la lettura da file
- Usare una variabile che indica l'ultimo elemento (indice o pointer)
- Usare una struttura di supporto che mi indica se l'elemento dell'array sia pieno o vuoto. Permette di gestire i buchi all'interno di un array.

Funzioni di `string.h`

- `int strcmp(char* s1, char* s2)` : confronta due stringhe ritorna:
  - 0 se uguali
  - intero minore di 0 se s1 è precedente alfabeticamente
  - intero maggiore di 0 se s1 è successiva
- `char* strcpy(char* s1, char* s2)` : copia s2 in s1 (`\0` compreso). s1 viene ritornato
- `char* strcat(char* s1, char* s2)` aggiunge s2 a s1 e pone il risultato in s1
- `unsigned int strlen(char* s1)` ritorna la lunghezza di s1 escluso lo `\0`

18 ottobre

Puntatori (pointer)

I puntatori sono variabili contengono un indirizzo di memoria, a differenza delle normali variabili che contengono un valore. Per dichiarare un puntatore si usa questa sintassi:

```
int var = 0;
int* p1;    // crea p1 che punterà all'indirizzo. Ora è INDEFINITO.
int* p2;    // crea p2 che punterà all'indirizzo. Ora è INDEFINITO.
p1 = &var;  // p1 punta al valore contenuto in var.
            // la & indica che si vuole l'indirizzo di una variabile.
p2 = p1;    // p2 e p1 puntano alla stessa variabile var
```

La costante `NULL` significa che un puntatore non punta a nessun valore. Ogni puntatore avrà anch'esso un indirizzo di memoria che potrà essere contenuto da un puntatore a un puntatore e così via: si possono usare pointer-to-pointer o doppia indirezione:

```
int* p1;
int** p2p;
p2p = &p1;
```

I puntatori sono tipizzati, quindi devono puntare a un valore del tipo corrispondente. Un assegnamento tra puntatori di tipo diverso genererà un warning a compile-time perchè il compilatore non sa se l'azione sia voluta o un errore:

```
int* p1;
float* p2;
p2 = p1;    // warning
```

L'eguaglianza tra puntatori verifica che due puntatori puntino alla stessa variabile, ossia che contengano lo stesso indirizzo di memoria:

```
int* p1;
int* p2;
```

```
p1 = p2;
if (p1 == p2)
    printf("ok\n");
------------------------------------------------------------------------
out: ok
```

Per modificare il valore di una variabile a partire da un puntatore si usa l'operatore di dereferenziazione:

```
int var = 0;
int* p1;
p1 = &var;

*p1 = 5;    // adesso var = 5
```

Se si omette la `*` si genera una warning:

```
int* p1;
p1 = 10;        // warning
p1 = (int*) 10; // ok
```

Questo causa il salvataggio di `10` in `p1`. Questo valore verrà interpretato come un indirizzo di memoria. Quindi quando si accederà con la dereferenziazione si accederà al valore contenuto all'indirizzo 10 causando quasi sempre segfault.

La `*` può essere anche usata per accedere al valore puntato da un puntatore, ad esempio in una printf Usando l'esempio sopra avremo:

```
printf("%d %d", var, *p1);
------------------------------------------------------------------------
out: 5 5
```

Aritmetica dei puntatori

Per spostare i puntatori manualmente si può usare l'aritmetica dei puntatori. Ci sono 2 operazioni `+` e `-`. Queste operazioni vengono usate soprattutto per muoversi all'interno di un array di elementi.

```
typedef struct {int a; int b;} el;
el vet[10];
el* P = vet;    // punta a vet[0]
*P = 0
*(P + 1) = 1;   // punta a vet[1] e gli assegna 1
```

`P + 1` viene espansa a `P + 1*sizeof(el)`. Stessa cosa avviene con il `-`.

Suando la seguente sotazione:

```
ind dist;
int* p, q;
[...]
dist = (p - q)  // distanza tra l'elemento associato p e quello di q
                // in numero di elementi.
```

`p-q` viene espanso a `(p-q)/sizeof(int)`

I tipi composti

[...]

Costruttore array

[...]

Accesso

[...]

Si può accedere al vettore con un puntatore in modo molto non sicuro:

```
int art[3] = {1,2,3};
int* point = &art;                  // equivalente a &arr[0]
printf("%d %d\n", art[1], point[1]);
-------------------------------------------------------------------------
out: 1 1
```

Si può usare anche l'aritmetica dei puntatori.

Costruttore record

Un record è un aggregato di valori eterogenei con nome. E' a dimensione fissa ed è salvato in memoria centrale. Permette l'accesso per nome agli elementi.

Accesso

SI può accedere ai membri di un record tramite il `.`:

```
struct A {int a; int b;};
struct A var;
struct var.a = 1;
```

Se la variabile dalla quale si accede è un puntatore, si può usare l'operatore `->` invece di combinare `.` e `*`:

```
struct A {int a; int b;};
struct A var;
struct A* point = &var;
struct point->a = 1;        // equivalente a (*point).a
```

23 ottobre

Funzioni

Permette di suddividere un programma in diversi "sotto-programmi" (logica divide and conquer). Sono dei pezzi di codice riutilizzabili in diversi punti del programma. Le funzioni si dichiarano così:

```
function_definition ::= type identifier({formal_parameters}_opt)
    { local_declarative_part executable_part }

formal_parameters      :: = {formal_parameter,}_0+
formal_parameter       :: = type identifier
local_declarative_part :: = type_declarations ??| variable_declarations
```

Si può notare, quindi, che anche lo stesso `main` è una funzione. Le funzioni vanno dichiarate prima del loro utilizzo. Se ciò non avviene, va utilizzato un 'function prototype' (prototipo di funzione):

```
function_prototype ::= type identifier({formal_parameters}_opt);
```

I parametri formali di una funzione hanno visibilità limitata: esse sono raggiungibili solo dalle istruzioni locali alla funzione (vedi regole di scope).

Una funzione deve restituire un valore nel caso il suo tipo non sia `void`. L'istruzione per ritornare uno e un solo valore è `return`;

```
return ::= return expression;
```

Una funzione deve avere un `return` per ogni cammino possibile. Ciò implica che una funzione può avere più di un `return`. Il raggiungimento di una `return` causa il troncamento dell'esecuzione della funzione e il ritorno all'esecuzione della funzione chiamante. Se il `main` chiama `return`, allora l'esecuzione termina.

Non si può definire una funzione annidata in una funzione.

Ogni funzione può contenere dei dati locali: infatti nel programma vengono allocate delle aree dati "locali" strettamente legate alle istruzioni della funzione alla quale le informazioni appartengono. La zona riservata a queste aree è detta 'stack'.

La memoria, quindi, viene allocata dinamicamente nella stack alla chiamata di funzione. Quando la funzione termina l'esecuzione, la memoria relativa alla funzione viene liberata. Ogni elemento della stack è detto 'stack frame'. Il primo stack frame è il main stesso: da esso, infatti, partono tutti gli altri; quando lo stack frame del main viene liberato, l'esecuzione termina.

25 ottobre

Funzioni

[...]

Suddivisione della memoria

I dati possono essere di due tipi: globali e locali. I dati globali sono allocati in un'area di memoria in testa al programma di dimensione fissa. Quest'area può andare in overflow nel caso ci siano troppe dati globali (individuato a compile time).

I dati locali, invece, sono immagazzinati nello stack frame associato alla funzione che li riguarda. Anche lo spazio locale può essere sfondato, ma questo sfondamento può essere individuato solo a runtime in quanto le variabili locali sono allocate solo a runtime. Ogni area di memoria locale associata a una funzione viene detta Record di Attivazione.

"Sottoprogrammi" procedurali

Sono funzioni che non restituiscono nulla. Hanno interfaccia:

```
void indentifier({formal_parameters}_opt)
```

La `return` può ancora esistere:

- se la `return` non ha espressione allora la funzione termina come previsto;
- se la `return` ha espressione il compilatore ci lancia un warning e il valore della return verrà peso;
- la `return` può essere omessa. La funzione termina quando viene raggiunta la `}` che termina il blocco di istruzioni.

Interfaccia di una funzione

L'interfaccia di una funzione sono tutte le informazioni che il chiamante ha: il tipo di ritorno, l'identificatore e i parametri formali.

```
type identifier({formal_parameters}_opt)

formal_parameters      ::= {formal_parameter,}_0+
formal_parameter       ::= type identifier
```

I parametri attuali sono i valori effettivamente passati alla funzione in chiamata:

```
function_call ::= identifier({actual_parameters}_opt)

actual_parameters      ::= {actual_parameter,}_0+
actual_parameter       ::= type identifier
```

I numeri e di parametri attuali e formali deve combaciare. Anche i tipi devono essere compatibili (stare attenti alle regole di conversione implicita). I due identificatori, però, non per forza combaciano.

I parametri vengono passati alle funzioni *per valore*: i valori dei parametri attuali vengono copiati in nuove variabili locali alla funzione.

Parametri formali e attuali

I tipi dei parametri possono essere:

- tipi base;
- tipi definiti da noi;
- struct;
- puntatori;

I vettori non possono essere passati alle funzione: l'array decade a un puntatore al suo primo elemento.

L'ordine di valutazione dei parametri è implemetation defined e non è garantito essere lo stesso ordine di definizione. Espressioni di questo tipo, allora:

```
func(var1++, var1 + 4);
```

Avranno comportamento indefinito.

Restituzione di valori

Una funzione può restituire uno e un solo valore. Il tipo di questo valore può essere:

- tipi base;
- tipi definiti da noi;
- struct;
- puntatori;
- void (la funzione non restituisce)

Per i vettori succede la stessa cosa come con le funzioni: l'array decade a un puntatore al suo primo elemento.

Il valore della return può essere salvata in una variabile. Esempio:

```
a = f(x);
```

Indirizzo di ritorno

Ogni funzione, quando viene chiamata, oltre alle variabili locali, contiene un indirizzo chiamato indirizzo di ritorno (`ret_adr`) che contiene l'istruzione che il program counter eseguirà dopo il termine della funzione, ossia il punto da dove riprendere l'esecuzione del chiamate una volta che la funzione chiamata ritorna.

Passaggio per riferimento

Per passare per riferimento e permettere a una funzione di modificare valori all'esterno del proprio scope bisogna passare un puntatore:

```
int func(int *punt) {...}
```

Per accedere al valore di `punt` nella funzione usiamo:

```
*punt = 10;
```

Per invocarla, invece, dovremo passare l'indirizzo con `&` :

```
var = f(&other_var);
```

Il passaggio per riferimento è l'unica modalità per passare degli array alle funzioni. Di conseguenza, gli array possono essere sempre modificati dalle funzioni.

Array e funzioni

Ci sono modalità diverse di passare gli array alle funzione:

```
void f(char s[]);
void f(char s[10]);     // Come la sopra: il numero viene ignorato
void f(char *s);
void f(const char *s);  // Rende l'array read-only
void f(array_t a);      // con typedef char array_t[10];
```

La funzione può accedere agli elementi dell'array anche se esso è passato come puntatore. Quindi avremo che:

```
s[i] ::= *(s + i)
```

30 ottobre

Funzioni

[...]

Prototipo di funzione

IL prototipo di una funzione è la sua interfaccia, senza il corpo. Serve solo a definire l'esistenza della funzione e a far capire al compilatore come usarla (numero e tipo di parametri) in codice precedente alla dichiarazione di essa. Il compilatore completerà la definizione con la funzionalità quando incontrerà la definizione completa della funzione.

Il main

Il main è la funzione principale di un programma. Esso può prendere dei parametri:

```
int main(int argc, char* argv[])
```

Dove:

- `argc` è il numero di parametri passati
- `argv` è un array che contiene i parametri più il nome stesso del programma in posizione 0

I parametri sono passati al main direttamente dal loader del sistema operativo. Essi corrispondono ai parametri passati nella linea di comando:

```
/path/to/program/ arg1 arg2 arg3
------------------------------------------------------------------------
argc = 4
argv = 0 /path/to/program
       1 arg1
       2 arg2
       3 arg3
```

Scope

Il C usa regole di scope statico: lo scope non dipende dall'ordine di invocazione delle funzioni. Esse dipendono solo dalla struttura delle definizioni.

I limiti di validità di una variabile sono il blocco in cui è definito:

```
block ::= { statements }
```

Un blocco può contenere più blocchi. Una funzione contiene sempre 1 blocco. E' comodo pensare ai blocchi come una matrioska:

```
int main(int argc, char* argv) {
    {
        int a; /* blocco 1 */
        { int inner /* blocco 3 */ }
    }
    { int b; /* blocco 2 */ }
}
--------------------------------------------------------------------------
global
|
\ main
  |
  \ blocco 1
  | |
  | \ blocco 3
  |
  \ blocco 2
```

Nello stesso blocco non posso usare due volte lo stesso nome, in blocchi diversi sì. Il campo di validità corrisponde, quindi, a tutti i blocchi più interni rispetto a quello di definizione.

Nel risolvere i conflitti di nomi, il compilatore sceglie sempre lo scope più interno in cui è definito il nome.

N.B.: I parametri di una funzione fanno parte del blocco più esterno di una funzione. Considerando l'esempio sopra, i parametri del main appartengono al blocco `main`. Il nome della funzione è, invece, globale (`main` infatti appartiene ai nomi globali).

13 novembre

Funzioni

[...]

Area dati locale di una funzione

Possono essere usati due approcci:

1. Approccio statico: il compilatore alloca a tutti i locali al momento del lancio del programma (+ memoria, - complesso, + veloce, niente ricors.)
2. Approccio dinamico: il compilatore alloca i locali solo al momento dell'invocazione della funzione (- memoria, + complesso, + lento, ricors.)

Il primo approccio è usato dai vecchi linguaggi. Il C usa il secondo.

I locali della funzione vengono salvati nel record di attivazione (RDA).

RDA

Il RDA non contiene solo i locali della funzione, ma anche:

- valore di ritorno della funzione
- parametri della funzione
- registro per referenziare variabili locali (traduce l'indirizzo locale a quello reale)
- indirizzo di ritorno (salto al chiamante)
- variabili locali

- link statico per lo scope (non trattato - permette di trovare i globali)

Il RDA viene allocato nell'area stack della memoria. L'indirizzo del primo elemento libero dello stack viene salvato in un registro del processore: lo stack pointer (SP). Lo SP si muove su è giù lungo la stack puntando il primo elemento libero:

- se viene deallocato un RDA, lo SP si sposta giù di n byte
- se viene allocato un RDA, lo SP si sposta su su di n byte

Assembly delle funzioni

Aggiungiamo al set di pseudo istruzioni le seguenti:

```
JTS I   Eseguire la funzione che inizia dall'etichetta I
RTS     Ritorno da una funzione

(W)     contenuto della parola di memoria il cui indirizzo è nel registro W
```

La direttiva `JTS` può essere tradotta in queste istruzioni:

```
MOV #RET SP     Punto dove ritornare quando si finisce la funzione
ADD #-1  SP     Incremento stack pointer
MOV #FUN PC     Modifica del PC per eseguire la prima instruzione di FUN
```

La direttiva RTS può essere tradotta in queste istruzioni:

```
ADD #1   SP     Fa puntare SP all'indirizzo di ritorno
MOV (SP) PC     Carica l'indirizzo di ritorno nel PC per riprendere il
                chiamante
```

Ogni funzione ha un 'prologo' che inizializza i registri necessari:

```
FUN: MOV R0  (SP)   Salva il vecchi valore di R0 (per non distruggere il chiamante)
     MOV SP  R0     Salva l'indirizzo del primo locale in R0
     ADD #-1 SP     Incrementa lo stack pointer
```

Una funzione usa i registri `Rn` per le sue operazioni e per accedere a parametri/locali. Il primo locale sarà sempre il registro `R0`. `R0` verrà usato per accedere ai vari parametri della funzione, distanti 4 parole di memoria, scendendo nello stack:

```
p1 -> R0 + 4 + 1
p2 -> R0 + 4 + 2
...
pn -> R0 + 4 + n
```

oppure accedere ai locali salendo nello stack:

```
var1 -> R0
var2 -> R0 + 1
...
varn -> R0 + n-1
```

Questo è il motivo per cui ogni funzione chiamata deve salvare il valore di `R0` precedente alla sua invocazione e resettarlo al suo termine in modo da non distruggere l'operazione della funzione chiamante.

Alla fine del prologo, un RDA avrà questa forma:

```
SP ->   ~
        varn
        ...
R0 ->   var1
        vecchio R0
        indirizzo di ritorno
        parametro 1
```

```
        parametro 2
        ...
        parametro n
        valore di ritorno
        --------------------
        [ altro eventuale RDA ]
```

AEsempio di assembly di una chiamata di funzione:

```
     A:  RES 1               # Init globali e stack
     B:  RES 1
     C:  RES 1
 STACK:  RES 1000
    IN:  MOV #2      A       # Allocamento globali
         MOV #3      B
         MOV #STACK  SP      # Init SP
         ADD #1000   SP
         ADD #-1     SP      # Allocamento valore di ritorno nell'RDA
         ADD #-1     SP      # Allocamento primo parametro
         MOV A       (SP)
         ADD #-1     SP      # Allocamento secondo parametro
         MOV B       (SP)
         ADD #-1     SP      # Allocamento spazio per indirizzo di ritorno
         JTS SUM             # Salto alla funzione SUM
   RET:  ADD #3      SP      # Pulizia SP: deallocazione parametri e indirizzo di ritorno
         MOV (SP)    C       # Salva il valore di ritorno in C
         EXIT                # Termina il programma
   SUM:  MOV R0      (SP)    # Salva il precedente R0
         ADD #-1     SP
         MOV SP      R0      # Inizializza R0: alloca il primo locale
         ADD #-1     SP
         MOV R0      R1      # Accedi al primo parametro (A)
         ADD #4      R1
         MOV (R1)    (R0)    # Salva la parola puntata da R1 in quella puntata da R0
         MOV R0      R1      # Accedi al secondo parametro (B)
         ADD #3      R1
         ADD (R1)    (R0)    # Somma il locale (con valore A) con il parametro (B)
         MOV R0      R1      # Copia il locale nel valore di ritorno
         ADD #5      R1
         MOV (R0)    (R1)
         ADD #2      SP      # Dealloca il locale e il backup di R0
         MOV (SP)    R0      # Ripristina la vecchia R0
         RTS                 # Ritorna il controllo al chiamante
         END IN              # Termine programma
```

Funzioni ricorsive

Una funzione ricorsiva è una funzione che chiama se stessa. Per ogni chiamata viene allocato un nuovo RDA, generando una specie di matrioska di funzioni. Con un esempio:

```
int fact(int n)
{
    if (n == 1)
        return 1;
    else
        return n * fact(n-1);
}

fact(4);

RDA di fact(1) -> ritorna 1
|-> RDA di fact(2) -> ritorna 2 * fact(1) = 2 * 1
    |-> RDA di fact(3) -> ritorna 3 * fact(2) = 3 * 2 * 1
        |-> RDA di fact(4) -> ritorna 4 * fact(3) = 4 * 3 * 2 * 1
            |-> main -> fact(4)
```

NB: E' facile causare uno stack overflow con funzioni ricorsive. Ad esempio, modificando `fact()` così:

```
int fact(int n) { return n * fact(n-1); }
```

Chiamando nel `main` `fact(4)` la funzione continuerà a chiamarsi all'infinito.

15 novembre

Strutture dati dinamiche

A differenza delle variabili (globali e locali) l'allocazione e deallocazione è gestita *interamente* dal programmatore.

Le variabili sono allocate a run-time nella heap, fornendoci più flessibilità ma anche comportando dei problemi.

La heap

La heap è l'area di memoria nel quale vengono allocati i dati a run-time. La heap cresce in direzione opposta alla stack e si trova esattamente sopra l'area dati dei globali.

La heap, a causa delle continue allocazioni e deallocazioni di oggetti di dimensioni differenti, tenderà a frammentarsi. Il C non ha nessun metodo di deframmentazione dello heap.

La frammentazione della heap può essere un problema da tenere a mente. Ci sono alcune strategie per prevenirla. Una può essere:

- allocatore custom
- usare piccoli oggetti con breve durata
- usare la heap solo se necessario

Per più info cerca su StackOverflow.

`malloc` e `free`

L'allocazione e la deallocazione di strutture dinamiche avviene tramite le rispettive funzioni:

```
void *malloc(size_t num);
void free(void* to_free);
```

Entrambe le funzoini sono definite nell'header `stdlib.h` .

`malloc`

La `malloc` può fallire (per eventuali motivi). In quel caso verrà restituito `NULL` . Il tipo di ritorno, `void*` , è un puntatore che punta a qualsiasi oggetto. Il puntatore restituito punta al primo byte della struttura allocata.

Il valore in ingresso è il numero di byte da allocare. `size_t` è un tipo usato per rappresentare una dimensione in byte. `size_t` viene ritornata, ad esempio, dall'operatore `sizeof` . Perciò è buona norma mai usare numeri fissi nella `malloc` . Sempre usare una espressione del tipo:

```
n * sizeof(tipo)

    n = numero di elementi che si vuole allocare
```

Uso

```
// Il casting di void* può essere lasciato implicito. Dipende da compilatore
// e standard utilizzato. Con gcc con C99 può essere lasciato implicito.
tipo *p = (p*) malloc(sizeof(tipo));
```

Attenzione a come vengono gestiti i puntatori e l'accesso alle strutture in quanto si può facilmente corrompere strutture dati adiacenti (heap overflow)

`free`

La `free` dealloca l'area di memoria *allocata da malloc* puntata dal puntatore passato.

L'area di memoria puntata dal parametro, dopo `free` , sarà deallocata, ma non viene sovrascritto il contenuto e il puntatore viene lasciato intatto. Ciò significa che l'area può essere ancora raggiunta dal puntatore, che punterà a un'area a contenuto indefinito. E', quindi, buona norma far puntare il puntatore usato nella `free` a `NULL` per evitare errori.

Problemi

Visto che la gestione della memoria è lasciata al programmatore, possono sorgere dei problemi a causa di errori di programmazione:

- Produzione di garbage: perdita di riferimento a una struttura allocata a causa di sovrascrittura di variabile o di uso in una funzione senza free:

```
t *p = malloc(sizeof(t));
t *q = malloc(sizeof(t));

q = p; // il primo oggetto è irraggiungibile
-----------------------------------------------------------------------
void F() { t *p = malloc(sizeof(t)); }
int main(void) { F(); }
// oggetto allocato in F() non è raggiungibile: p viene deallocato in
// quanto nello stack.  Nella heap, però, è ancora presente l'oggetto
// generato da malloc.
```

- Dangling pointers: utilizzo di un puntatore che punta a memoria deallocata:

```
t *p = malloc(sizeof(t));
...
free(p);
...
p->a = ...; // Errore: possibile corruzione di altre strutture
func(p->a); // Errore: lettura di carbage
-----------------------------------------------------------------------
int *p;
void F() { int n = 10; p = &n; }
int main(void) { F(); }
// p punta a un'indirizzo nel RDA di F, deallocato alla fine
// dell'esecuzione di F.
```

Strutture dati concatenate (linked lists)

Le strutture dati concatenate hanno tutte le stesse caratteristiche:

- insieme di tipo omogeneo
- collegamento tra i vari elementi tramite puntatori
- accesso tramite un punto d'accesso (handle) ben definito

Ogni elemento viene detto nodo e, generalmente, contiene i seguenti campi:

1. contenuto del nodo
2. puntatore all'elemento successivo
3. (optional) puntatore all'elemento precedente

L'ultimo punterà a `NULL` per indicare la fine della lista.

A causa della dinamicità della struttura, i nodi vengono tutti a runtime, quindi saranno nella heap.

Le strutture dinamiche sono molto flessibili: tutte hanno costituzione simile, ma generano strutture ben diverse. Basti pensare a come lo stesso nodo può generare un albero binario o una lista bidirezionale in base a cosa puntano i due puntatori interni

Quale usare?

Non c'è una risposta definitiva. Bisogna usare quella che ci semplifica di più la vita:

- gestione di coda? Queue
- gestione di stack-like? Stack
- gestione di sequenze? Considero i vettori. Mi serve la dinamicità? Lista

In ogni caso, non saltare subito e utilizzare le strutture dinamiche in quanto esse richiedono molto più codice.

Esempi

Lista monodirezionale (singly linked list)

```
struct Node {
    int val;
    struct Node *next;
};
struct Lista {
    struct Node *head;
    struct Node *tail;
};
```

Rappresentazione della struttura:

```
head = &A    | A          | B            | tail = &B
             | next = &A | next = NULL |
```

Lista bidirezionale (doubly linked list)

Simile alla lista monodirezionale, ma ogni nodo contiene anche un riferimento all'elemento precedente.

```
struct Node {
    int val;
    struct Node *next;
    struct Node *prev;
};
struct Lista {
    struct Node *head;
    struct Node *tail;
};
```

Rappresentazione della struttura:

```
head = &A    | A            | B            | tail = &B
             | next = &A    | next = NULL |
             | prev = NULL | prev = &A    |
```

4 dicembre

Strutture dai persistenti

Una richiesta di accesso a dati su disco percorre una catena:

```
IO request
File System (OS)
Unità RW
Driver Disco (Scheduler - OS)
BUS
Controller disco
Disco
```

Ogni linguaggio di programmazione crea una `IO request`. Il sistema operativo gestirà tutto e restituirà un'interfaccia per l'accesso ai dati. Un programma in C non conosce (e non deve) i dettagli di implementazione.

Il file

Esistono diversi tipi di file:

- file speciali (dispositivi)
- file directory
- file normali

Un file può essere di diverso formato:

- testo: tutto è un carattere (ASCII o UTF-8)
- binario: un blocco di bit non codificati

Il file, inoltre, ha diversi tipi di accesso:

- sequenziale: accesso dal primo carattere all'ultimo
- diretto: accesso direttamente alla posizione richiesta (solo binario)

Ogni byte del file ha un'indirizzo: il primo è `0`, il secondo è `1` e così andare in maniera sequeziale. L'ultimo carattere darà sempre `EOF`, un carattere speciale che indica la fine di un file.

Uso concorrente dei file

Nel momento in cui due programmi distinti accedono allo stesso file si ha un problema di concorrenza. Il OS consentirà l'accesso sono nei seguenti casi:

	R	**W**
R	OK	NO
R	NO	NO

Quindi accedere in scrittura a un file è possibile solo se si è gli unici a scriverci, gli altri possono solo leggere. (Coi database il blocco si sposta dai file ai record)

Scrittura di un file

Le scritture su un file non modificano direttamente il file aperto: il sistema operativo salva le operazioni di scrittura in un suo buffer interno. Le scritture vengono effettuate quando:

- il buffer è pieno
- viene chiuso il file
- altro

I File in C

Per aprire un file si usano le funzioni della `stdio.h`:

```
FILE *fopen(char *path, char *mode) - Apre un file con percorso `path` in
                                      modalità `mode`
```

Modalità di apertura di un file:

- `r`: read - lettura (fallisce se il file non esiste)
- `w`: write - scrivi troncando il file (fallisce se il file non esiste)
- `a`: append - scrivi aggiungendo al file (fallisce se il file non esiste)
- `*b`: read/write/append di un file binario (fallisce se il file non esiste)
- `**+`: aggiungere `+` alle mode precedenti crea il file se non esiste

Per chiudere un file si userà:

```
int fclose(FILE *descriptor) - Chiude il file relativo a `descriptor`
```

NB: la `fclose` finalizza eventuali operazioni di scrittura compiute su file aperti. La funzione

```
int fflush(FILE *fd)
```

Scrive tutte le informazioni su disco senza chiudere il file.

FILE e descrittori di file

Il puntatore `FILE` è un puntatore a un'area di memoria al descrittore del file aperto. Un descrittore di file è una struttura gestita dal sistema operativo ed è mantenuta in una tabella contenente tutti i file aperti dal programma. Se la `fopen` fallisce, allora non potrà essere creato il descrittore necessario e quindi `FILE` sarà un puntatore a `NULL`.

Manipolazione di dati dei file:

```
File di testo:
    int fprintf(FILE*, char*, ...) - scrive alla posizione attuale una stringa
                                     formattata su file. RItorna il numero di
                                     caratteri scritti o un numero negativo in
                                     caso di errore
    int fscanf(FILE*, char*, ...)  - legge dalla posizione corrente in base a
                                     una stringa di formato. Ritorna il numero
                                     di caratteri letti o EOF
    char fgetc(FILE*)              - legge un singolo carattere da file
    int fputc(char, FILE*)         - scrive un singolo carattere su file
    char *fgets(char*, int, FILE*) - legge una stringa di lunghezza n da file.
                                     NB: la lettura termina all'incontro di
                                         \n o EOF
    int fputs(char*, FILE*)        - scrive una stringa su file

File binari:
    fread(void*, size_t, size_t, FILE*)   - legge n blocchi di m byte da file
    fwrite(void*, size_t, size_t, FILE*)  - scrive n blocchi di m byte da file
```

6 dicembre

Strutture dati persistenti

...

File e interoperabilità tra programmi

Un programma può scrivere le strutture che usa in binario. Ciò, però, impedisce ad altri programmi di aprirli: l'altro programma dovrebbe definire le stesse strutture dati che vengono usate nel tuo programma. E' un colpo grosso alla portabilità:

```
#define N 10
...
obj V[N] = ...
...
fd = fopen(file_path);
if (fd)
    fwrite(V, sizeof(obj), N, fd);
...
```

Apparentemente non causerebbe problemi se l'utente del file usa la stessa definizione del tipo `obj`. Il problema è che piattaforme diverse potrebbero usare diverse dimensioni per gli stessi oggetti (ad esempio `int` da 4 o 8 byte). La `sizeof` quindi è ha comportamento implementation dependent. Se lo stesso file scritto da da codice sopra viene letto su una piattaforma che utilizza una implementazione discordante da quella del sistema su cui è stato scritto il file verrà letto in modo incorretto.

La soluzione al problema sopra è usare file di testo codificati con una codifica ben nota (ASCII o UTF-8 di solito) e formato ben noto (ad esempio XML, JSON o altri). Allora tra 'scrittore' e 'lettore' rimangono da concordare solamente il senso dei dati

contenuti.

Utilizzo dei file da parte di un programma

Un file può interagire col programma in due modi:

- passivo: tutti i dati vengono scritti/letti in una sola passata. Esempio di uso passivo: salvataggio dello stato
- attivo: i dati vengono scritti/letti quando necessario. Esempio di uso attivo: gestione di un database

11 dicembre

Rappresentazione dell'informazione

L'unità fondamentale di rappresentazione in memoria è il 'binary digit' o bit. Esso è la cifra binaria (0 o 1).

Numeri naturali

Un numero naturale è memorizzato in una parola di memoria di `N` bit. Ciò rende impossibile rappresentare tutti i numeri naturali in quanto sarebbero richieste infinite numeri decimali. Questo modo definisce degli intervalli di rappresentabilità. L'intervallo di rappresentabilità varia in base al numero `N` di bit disponibili ed è pari a `[0; 2^N - 1]`.

Numeri con segno

I numeri con segno vengono salvati seguendo la regola del complemento di 2. Questo comporta che l'intervallo di rappresentabilità è diviso in metà. Per una spiegazione nel dettaglio su come funziona il complemento di 2 vedi CODE di Petzold.

Nel caso che un'operazione vada in overflow, il sommatore setta un bit nel registro di stato per indicare l'overflow. Per controllare che sia avvenuto l'overflow, viene controllato il segno dei due operandi:

1. se esso è concorde ed è uguale a quello del risultato o è allora va tutto bene
2. se esso è discord allora qualsiasi sia il risultato, per come abbiamo definito l'addizione, andrà bene
3. se invece i primi due sono concordi e il risultato è discorde allora si è andati in overflow

Caratteri

I caratteri sono codificati da numeri secondo lo standard ASCII (7-bit).

13 dicembre

Rappresentazione dell'informazione

[...]

Numeri reali

Ci sono diversi modi di rappresentare i reali.

Virgola fissa

I numeri reali hanno due parti diverse da memorizzare: la parte intera e la mantissa. Vengono usati 16 bit per rappresentare la parte intera e 16 per la parte decimale.

La parte intera viene codificata come già visto precedentemente. La parte decimale viene rappresentata usando un algoritmo del "prodotto per 2":

```
Dec   | x2   | Riporto (x2 supera l'unità?)
0.375 | 0.75 | 0
0.75  | 1.5  | 1
0.5   | 1.0  | 1
```

```
0     | 0

0.375 = 0.0110 00...
```

Problema della precisione

L'algoritmo per la rappresentazione in binario del decimale può non convergere, determinando una perdita di precisione. E' necessaria quindi un'approssimazione.

Gli 'step' discreti sono determinati da quanti bit vengono usati per codificare: lo step di una parte decimale codificata con `n` bit sarà pari a `2^-n` . L'errore assoluto sarà, quindi, costante. L'errore relativo, invece sarà minore per numeri molto grandi e maggiore per numeri molto piccoli

Virgola mobile

Lo standard a virgola mobile è lo standard (IEEE 754-1985) in utilizzo oggi. Tutte le operazioni su floating point, allora, saranno 'sbagliate allo stesso modo'. Garantisce la costanza dell'errore relativo, ma non lo elimina.

Lo standard si divide in due: `FP32` (virgola mobile a 32 bit, il `float`) e `FP64` (virgola mobile a 64 bit, il `double`)

Il numero viene prima normalizzato: `+- x * 2^exp` dove:

- `exp` è l'esponente della potenza di due più vicina al nostro numero
- `x` è il numero diviso a `2^exp`

In memoria viene salvato:

- il bit di segno
- la mantissa di `x`
- l'esponente della potenza di 2

```
FP32:
      1  bit | segno
    + 23 bit | mantissa di x
    + 7  bit | max exp (127) + esponente di 2
    = 32 bit
FP64:
      1  bit | segno
    + 52 bit | mantissa di x
    + 10 bit | max exp (1023) + esponente di 2
    = 54 bit
```

Anche se a prima vista sembra che con lo stesso numero di bit si rappresentano molti più numeri rispetto ad un intero, è un'illusione.