

Bigné alla crema

- Come si preparano i bigné alla crema?
- si prepara la **pasta choux**
- si depositano piccole quantità di impasto sulla teglia
- si fanno cuocere per 7-8 minuti, ottenendo i bigné
- si prepara la **crema**
- si farciscono i bigné con la crema
- Già, ma per preparare **pasta choux** e **crema** ho bisogno delle relative ricette!

Prodotto matriciale

- Data una matrice A ($M \times P$) ed una matrice B ($P \times N$), come si calcola la matrice C prodotto di A e B ?
- ogni elemento C_{ij} si ottiene dal **prodotto scalare** della riga i di A e della colonna j di B : $\mathbf{a}_i^T \times \mathbf{b}_j$
- Adesso ho tuttavia bisogno di un sotto-programma per calcolare il **prodotto scalare**!

Modularizzazione

- Quando abbiamo a che fare con un problema complesso spesso lo suddividiamo in problemi più semplici che risolviamo separatamente, per poi combinare insieme le soluzioni dei sottoproblemi al fine di determinare la soluzione del problema di partenza.
- Questo procedimento è applicabile anche alla programmazione.
 - si suddivide un problema complesso in problemi di volta in volta più semplici
 - una volta individuati (sotto)problemi sufficientemente elementari si risolvono questi ultimi direttamente
 - si combinano le soluzioni dei sottoproblemi per ottenere la soluzione del problema di partenza

- **Approccio top-down**: si parte dall'alto, considerando il problema nella sua interezza e si procede verso il basso per raffinamenti successivi fino a ridurlo ad un insieme di sottoproblemi elementari
- **Approccio bottom-up**: ci si occupa prima di risolvere singole parti del problema, per poi risalire procedendo per aggiustamenti successivi fino ad ottenere la soluzione globale.
- I linguaggi di programmazione mettono a disposizione dei meccanismi di **astrazione** che favoriscono un approccio modulare

Astrazione sui dati - si possono definire nuovi tipi di dato specifici per il particolare problema (**tipi di dato astratti**)

- collezioni di valori + relative operazioni

Astrazione funzionale - si possono definire **sottoprogrammi** per (sotto)problemi specifici.

- i sottoprogrammi sono di solito **parametrici** e in C si realizzano attraverso le **funzioni**
- possono essere (ri)usati alla stessa stregua delle operazioni built-in del linguaggio

Funzioni

- $y = f(x)$

In una funzione matematica, ad ogni valore della **variabile indipendente** o **argomento** x corrisponde uno e un solo valore della **variabile dipendente** y .

- Compito dell'informatica è quello di trovare delle tecniche per calcolare le funzioni alla base dei problemi da risolvere. In C esiste il concetto di **funzione**:
 - le variabili indipendenti sono chiamate **parametri formali**
 - il risultato viene restituito attraverso il comando **return**
- Un programma C è definito come un insieme di funzioni (una obbligatoria, il **main**).

Funzioni

- Una funzione può essere vista come una **scatola nera**:

parametri di ingresso \longrightarrow F \longrightarrow valore calcolato

- risolve un sottoproblema specifico
- attraverso i parametri e il risultato scambia informazioni con il `main` e con altre funzioni

Esempio:

x \longrightarrow abs \longrightarrow $|x|$

x, y \longrightarrow mcd \longrightarrow $mcd(x, y)$

b, e \longrightarrow exp \longrightarrow b^e

x_1, \dots, x_n \longrightarrow sum \longrightarrow $\sum_{i=1}^n x_i$

Esempio: Definizione di `abs` in C

```
int abs(int x)
{
    int ris;
    if (x<0)
        ris = -x;
    else
        ris = x;
    return ris; }
```

● Uso della funzione

```
main()
{
    int x1, x2, z, w;
    ...
    z = abs(x1);
    ...
    printf("%d\n", w + abs(x2));
    ...
}
```

Funzioni: perché?

La stesura di un programma riflette l'analisi funzionale del problema da risolvere. L'uso delle funzioni nasconde al resto del programma i dettagli implementativi, ponendo l'accento su **cosa il programma fa** rispetto a **come lo fa**, consentendo:

- la modularità
- la chiarezza e leggibilità
- la fattorizzazione del codice
- la separazione di ciò che cambia da ciò che resta uguale:
 - posso apportare modifiche alla funzione in un punto solo, senza rischiare modifiche parziali
 - posso anche cambiare l'implementazione della funzione senza cambiare il programma che la usa

Scrivi una volta, usa tutte le volte che vuoi

Ogni funzione rappresenta un'unità indipendente. Di conseguenza:

- chi scrive la funzione può non coincidere con chi la usa.
- la stessa funzione può essere usata in altri programmi (riuso del codice)
- ogni funzione può essere trattata separatamente dal resto

- Il linguaggio deve mettere a disposizione strumenti per
 - **definire** nuove operazioni astratte (funzioni)
 - **usare** le nuove operazioni definite
- Distinguiamo due momenti diversi:
 - la **definizione della funzione**
definisce il codice che realizza l'operazione astratta
 - e la **chiamata della funzione**
corrisponde all'utilizzo della funzione
- Ad una stessa definizione possono corrispondere diverse chiamate (come $z = \text{abs}(x1)$ e $w + \text{abs}(x2)$ nell'esempio precedente).
- Nella definizione della funzione, il codice fa riferimento agli **argomenti** o **parametri formali** della funzione (nell'esempio x)
 \implies un parametro formale non corrisponde ad un valore vero e proprio: è semplicemente un riferimento simbolico (ad un argomento della funzione)

Esempio:

```
int exp(int base, int esponente)
{
    int ris = 1;
    while (esponente > 0)
    {
        ris = ris * base;
        esponente = esponente - 1;
    }
    return ris;
}
```

- La definizione di una funzione consta di due parti fondamentali:
 - Intestazione
 - Blocco (come nel `main()`).
- I **parametri formali** sono `base` ed `esponente`

- Al momento della chiamata, alla funzione vengono forniti i valori degli argomenti, o **parametri attuali**, rispetto ai quali effettuare il calcolo

```
int exp(int base, int esponente)
{...}
```

```
main() {
int b, e, r1, r2;
...
r1 = exp(2,5);
...
scanf("%d %d", &b, &e);
r2 = exp(b, e);
... }
```

- Prima chiamata `exp(2,5)`
 - 2 è il parametro attuale corrispondente a **base**
 - 5 è il parametro attuale corrispondente a **esponente**
- Seconda chiamata `exp(b,e)`
 - **b** è il parametro attuale corrispondente a **base**
 - **e** è il parametro attuale corrispondente a **esponente**

Funzioni: definizione

Come le variabili, anche le funzioni vanno dichiarate.

Sintassi:

```
intestazione blocco
```

dove

- blocco è il **corpo della funzione**
- intestazione è l'**intestazione della funzione** ed ha la seguente forma:
id-tipo identificatore (parametri-formali)
- id-tipo specifica il **tipo del risultato** calcolato dalla funzione
- identificatore specifica il **nome** della funzione ed è un qualsiasi identificatore C valido
- parametri-formali è una sequenza (eventualmente vuota) di dichiarazioni di parametro (tipo e nome) separate da virgola

Esempi: intestazioni di funzione

- **int** abs (int x)
int MassimoComunDivisore(int a, int b)
- **double** Potenza(double x, double y)
float media (int vet[], int lung)

Funzioni: chiamata (invocazione, attivazione)

Sintassi:

`identificatore (parametri-attuali)`

- `identificatore` è il nome della funzione
- `lista-parametri-attuali` è una lista di **espressioni** separate da virgola
- i parametri attuali devono corrispondere in **numero** e **tipo** ai parametri formali, altrimenti sarà rilevato un errore di sintassi

Esempi: chiamate di funzioni

```
int mcd, x, y1, y2;  
double exp, w, v, z;  
...  
mcd = MassimoComunDivisore(x+1, y1+y2);  
exp = Potenza(z, 3.0);  
...  
exp = Potenza(z, Potenza(v,w));
```

Semantica (informale) di una chiamata di funzione

- Dentro il corpo di una funzione **F** compare una chiamata di un'altra funzione **G**
 - **F** viene detta funzione **chiamante**
 - **G** viene detta funzione **chiamata**
- **Esempio:** nel `main` c'è un assegnamento `x = abs(x);`
⇒ `main` è il chiamante, `abs` il chiamato
- Una chiamata di funzione è un'espressione, la cui valutazione avviene come segue:
 - viene sospesa l'esecuzione di **F** e viene “ceduto il controllo” a **G**, dopo aver opportunamente associato i parametri attuali ai parametri formali (**passaggio dei parametri**, fra poco ...)
 - vengono eseguite le istruzioni di **G**, a partire dalla prima
 - l'esecuzione di **G** termina con l'esecuzione di un'istruzione speciale (istruzione `return`) che calcola il risultato della chiamata (è il valore dell'espressione corrispondente alla chiamata)
 - al termine dell'esecuzione di **G** il controllo ritorna a **F**, che prosegue l'esecuzione a partire dal punto in cui **G** era stata attivata

Valore di ritorno di una funzione: istruzione `return`

- **Esempio:** Funzione che restituisce il massimo tra due interi.

```
int max(int m, int n) {  
    if (m >= n)  
        return m;  
    else  
        return n;    }
```

- Chiamata di `max`, ad esempio da `main`:

```
main() {  
    int i, j, massimo;  
    scanf("%d%d", &i, &j);  
    massimo = max(i,j);  
    printf("massimo = %d\n", massimo);    }
```

- La funzione `main` tramite i parametri attuali **comunica** alla funzione `max` i valori (di `i` e `j`) sui quali calcolare la funzione.
- La funzione `max` tramite il valore di ritorno **comunica** il risultato al `main`.

- Nel corpo **deve** esserci l'istruzione `return espressione;` la cui esecuzione comporta:
 - il calcolo del valore di `espressione`: questo valore viene restituito al chiamante come risultato dell'esecuzione della funzione
 - la cessione del controllo alla funzione chiamante

Osservazioni

- in `return espressione`, il tipo di `espressione` deve essere lo stesso del tipo del risultato della funzione dichiarato nella definizione
- l'esecuzione di `return espressione` comporta la **terminazione** dell'esecuzione della funzione

Esempio:

```
int max(int m, int n) {  
    if (m >= n)  
        return m;  
    else  
        return n;  
    printf("pippo");    /* non viene mai eseguita */ }  
}
```

Dichiarazioni di funzione (o prototipi)

- I parametri attuali nella chiamata di una funzione devono corrispondere in numero e tipo (in ordine) ai parametri formali.
- Dobbiamo permettere al compilatore di fare questo controllo
⇒ prima della chiamata deve essere nota l'intestazione
- Due possibilità:
 - 1 la funzione è stata **definita** prima
 - 2 la funzione è stata **dichiarata** prima

Sintassi della **dichiarazione di funzione** (o **prototipo**)

`intestazione;`

ovvero:

`id-tipo identificatore (parametri-formali);`

- c'è un “;” finale al posto del blocco
- nella lista di parametri formali può anche mancare il nome dei parametri — interessa solo il tipo
- il compilatore usa la dichiarazione per controllare che l'attivazione sia corretta
- dopo **deve** esserci una definizione della funzione coerente con la dichiarazione

Ordine di dichiarazioni e funzioni

- Bisogna dichiarare o definire ogni funzione **prima** di usarla (chiamarla)
- È pratica comune specificare in quest'ordine:
 - 1 dichiarazioni (**prototipi**) di tutte le funzioni (tranne `main`)
 - 2 definizione di `main`
 - 3 definizioni delle funzioni
- In questo modo ogni funzione è stata dichiarata prima di essere usata
- L'ordine in cui mettiamo le definizioni non deve necessariamente corrispondere a quello delle dichiarazioni.

Esempio:

```
int max(int, int);
```

```
int foo(char, int);
```

```
main() ...
```

```
int max(int m, int n) ... /* OK. definizione coerente  
con il prototipo */
```

```
int foo (int z, char c) ... /* NO! definizione non coerente  
con il prototipo */
```

Nella definizione di `foo` i parametri formali non sono nell'ordine specificato dal prototipo.

Passaggio dei parametri

- Abbiamo visto che le funzioni utilizzano **parametri**
 - permettono uno **scambio di dati** tra chiamante e chiamato
 - nell'intestazione/prototipo: lista di **parametri formali** (con tipo associato) – sono delle variabili
 - nell'attivazione: lista di **parametri attuali** — possono essere delle espressioni
- Al momento della chiamata ogni **parametro formale viene inizializzato al valore del corrispondente parametro attuale.**
- Il valore del parametro attuale viene **copiato** nella locazione di memoria del corrispondente parametro formale.
- Questo meccanismo di passaggio dei parametri viene comunemente detto **passaggio per valore.**

Esempio:

```
int succ (int);    /* prototipo di succ */

main()
{  int z, w;
   z = 10;
   w = succ(z);
   printf("%d", w);
}

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

- L'effetto della chiamata `succ(z)` può essere **simulato** dall'esecuzione della seguente porzione di codice:

```
x = 10;          /* il parametro formale si inizializza con
                  il valore del parametro attuale */
x = x + 1;       /* esecuzione del corpo della funzione
return x;        succ */
```

- Chiamate diverse corrispondono ad inizializzazioni diverse delle variabili corrispondenti ai parametri formali

```
w = succ(20);           x = 20;
                        ⇒  x = x + 1;
                        return x;
```

- In questo caso il valore assegnato alla variabile w è 21.

```
z = 10;
w = succ(z+3);         x = 13;
                        ⇒  x = x + 1;
                        return x;
```

- In questo caso il valore assegnato alla variabile w è 14.
 - Se non vi è corrispondenza perfetta tra il tipo del parametro formale e quello del parametro attuale, viene effettuata una **conversione implicita** di tipo secondo le regole già viste.
 - Il passaggio dei parametri di tipo array **non** comporta la copia dei valori dell'array (fra poco ...)

Procedure

- Non sempre le operazioni astratte di cui abbiamo bisogno possono essere descritte in modo naturale come funzioni matematiche.

Esempio: progettare un'interfaccia utente per la stampa di figure geometriche, in cui l'utente può scegliere:

- 1 la forma della figura
 - 2 la dimensione
 - 3 il carattere di riempimento
 - 4 ...
- In questo caso il compito dell'operazione astratta non è (o non è soltanto) produrre un valore, ma è produrre effetti di altro tipo, tipicamente **modifiche di stato**.
 - ⇒ in questi casi possiamo utilizzare **procedure**
 - le **procedure** sono un'astrazione delle **istruzioni**
 - le **funzioni** sono un'astrazione delle **espressioni**

Le procedure in C

- Una procedura è una funzione avente come tipo del risultato il tipo speciale `void`.
- La definizione/dichiarazione di procedure e la loro chiamata è analoga al caso delle funzioni

Esempio:

```
void emoticon (int n)
{ /* stampa n volte la sequenza -:) */
  int i;
  for (i=0; i<n; i++)
    {putchar('-'); putchar(':'); putchar(')'); putchar(' ');}
}
main() {
  ...;
  emoticon(3);
  ... }
```

- Le procedure non contengono di solito un'istruzione `return` (se la contengono è del tipo `return;` che non comporta il calcolo di alcun valore, ma solo la cessione del controllo al chiamante)

- La semantica di una chiamata di procedura **P** da una funzione/procedura **F** è analoga a quella della chiamata di funzione, ma una chiamata di procedura è un'istruzione
- In particolare, il passaggio dei parametri avviene per **valore** come nel caso delle funzioni
- il controllo viene restituito al chiamante al termine dell'esecuzione del blocco che costituisce il corpo della procedura (o in corrispondenza dell'esecuzione di un'istruzione del tipo **return;**)
- Il C non distingue tra funzioni e procedure (queste ultime sono casi particolari di funzioni)
⇒ concettualmente, però, è bene vedere le funzioni come astrazioni di **espressioni** e le procedure come astrazioni di **istruzioni**.

Esempio:

Procedura che stampa una cornice di asterischi di “altezza” parametrica

```
void stampaCornice(int altezza)
{
    int i;
    printf("*****\n");
    for (i=1; i<=altezza; i++)
        printf("*          *\n");
    printf("*****\n");
    return;
}
```

- Come astrazione delle istruzioni, le procedure possono dover **modificare lo stato**.

Esempio: Procedura `abs` che **assegna** ad una variabile intera `x` il suo valore assoluto

- il chiamante deve comunicare alla procedura la variabile `x`
- la procedura deve analizzare il valore della variabile e, se necessario, effettuare il rimpiazzamento
- La seguente realizzazione della procedura non è corretta

```
void abs(int x)
{   if (x < 0)
    x = -x;   }
```

- Simuliamo il comportamento di una chiamata della procedura (come visto in precedenza)

```
int z = -5;
abs(z);            $\implies$    x = -5;
                    if (x < 0) x = -x;
```

- La modifica del parametro formale **non** si ripercuote sul parametro attuale (il passaggio è **per valore**).