mlampis.dev

C

Manuale per le ripetizioni di Informatica

Marco Lampis

21 marzo 2023

Indice

0 Informazioni generali

Ciao! Sono **Marco**, sono uno studente magistrale in *Software Engineering* al Politecnico di Torino e mi sono laureato in Ingegneria Informatica all'università di Pisa. Nella vita sono un programmatore, uno smanettone e amante di videogiochi! Amo quello che studio, e per questo motivo fornisco ripetizioni di informatica con particolare attenzione a:

- programmazione (Java, C++, C, Python, C#, Javascript, PHP)
- algoritmi e strutture dati
- basi di Dati
- più o meno tutto quello che riguarda l'informatica!

Sia per studenti delle scuole superiori che per l'università.

0.1 Come svolgo le lezioni

La mia metodologia è **innovativa** e **interattiva**: utilizzo una piattaforma apposita per lavorare contemporaneamente sullo stesso file con gli studenti, come se fossimo accanto. In questo modo, le lezioni sono più coinvolgenti e divertenti, e gli studenti possono imparare in modo facile ed efficiente. Inoltre, le lezioni includono sia esercitazioni che approfondimenti teorici, aiuto compiti e revisione.

Per aiutare gli studenti a prepararsi al meglio e a esercitarsi anche a casa, ho creato un sito web dedicato con risorse utili come teoria, esercizi e slide. Su questo sito, gli studenti possono trovare tutto il materiale di cui hanno bisogno, e il materiale è sempre disponibile gratuitamente per i miei studenti. Inoltre, uso un iPad per prendere appunti durante le lezioni e poi condividerli con gli studenti, in modo che possano rivedere tutto ciò che è stato trattato.

Se volete migliorare le vostre conoscenze in informatica e avere un supporto personalizzato e professionale, non esitate a contattarmi. Sarò felice di fornirvi maggiori informazioni e di fissare una lezione con voi.

1 introduzione

Il *C* è un linguaggio di programmazione procedurale a basso livello e general-purpose sviluppato da *Dennis Ritchie* negli anni '70 per sistemi operativi di tipo UNIX. È stato successivamente adottato da molti altri sistemi operativi e piattaforme hardware e, a partire dagli anni '80, è stato utilizzato come linguaggio di riferimento per lo sviluppo di altri linguaggi, come C++ e Java.

Uno dei principali punti di forza di C è la sua efficienza e la sua capacità di controllo diretto delle risorse hardware del sistema. Queste caratteristiche lo rendono un linguaggio adatto allo sviluppo di sistemi operativi, driver di dispositivi e altri tipi di codice che richiedono prestazioni elevate e un livello di controllo accurato.

1.1 Caratteristiche del linguaggio C

C è un linguaggio di programmazione procedurale, il che significa che i programmi sono suddivisi in funzioni che vengono eseguite in sequenza per completare un compito specifico. C è un linguaggio a basso livello, il che significa che fornisce un livello di controllo diretto sulla memoria e sulle risorse hardware del sistema. C è un linguaggio general-purpose, il che significa che può essere utilizzato per sviluppare una vasta gamma di applicazioni, dai sistemi operativi ai giochi. C è un linguaggio compilato, il che significa che il codice sorgente viene convertito in codice binario eseguibile dal compilatore prima di essere eseguito dal sistema. Esempio di codice in C Ecco un semplice esempio di codice in C che stampa una stringa di testo a schermo:

```c
#include <stdio.h> // Importa la libreria standard di input/output

int main() { // Dichiara la funzione main, che rappresenta il punto di ingresso del programma
    printf("Ciao, mondo!\n"); // Stampa la stringa "Ciao, mondo!" a schermo
    return 0; // Restituisce 0 come codice di uscita del programma
}
```

Questo codice stamperà:

```
Ciao, mondo!
```

Come puoi vedere, il codice include una libreria standard di input/output (stdio.h) per poter utilizzare la funzione printf(), che viene usata per stampare la stringa “Ciao, mondo!” a schermo.

La sintassi della funzione main() è la seguente:

```c
int main() {
    // Codice del programma
    return 0;
}
```

Come puoi vedere, la funzione main() ha il tipo di dati int (che significa che restituisce un valore intero) e non accetta argomenti. Il codice del programma viene inserito all’interno delle parentesi graffe {} e, alla fine del codice, viene restituito il valore 0 per indicare che il programma è terminato correttamente.

1.2 Case sensitive

Il linguaggio C è case sensitive, il che significa che le lettere minuscole e maiuscole sono considerate diverse. Per esempio, le variabili `a` e `A` sono considerate due variabili diverse.

ASuggerimento: è possibile utilizzare le lettere maiuscole e minuscole per identificare le variabili, ma è una buona pratica utilizzare solo lettere minuscole per identificare le variabili.

Attenzione: possono esistere variabili con stesso nome ma che differiscono per le lettere maiuscole e minuscole. Per esempio, `ciao` e `CIAO` sono due file diversi.

2 main

In C, la funzione main è la funzione principale di un programma e viene eseguita quando il programma viene avviato.

Esempio:

```c
int main() {
  // codice del programma
  return 0;
}
```

In questo esempio, abbiamo definito la funzione main che non accetta alcun argomento e restituisce un valore di tipo int. All'interno del corpo della funzione, possiamo inserire il codice del nostro programma. La funzione main termina con l'istruzione return 0, che indica che il programma è stato eseguito con successo.

La funzione main può anche accettare argomenti dalla riga di comando:

```c
int main(int argc, char *argv[]) {
  // codice del programma
  return 0;
}
```

In questo esempio, la funzione main accetta due argomenti: argc e argv. Il primo argomento, argc, è un intero che indica il numero di argomenti passati alla riga di comando. Il secondo argomento, argv, è un array di stringhe che contiene i valori degli argomenti passati alla riga di comando.

Possiamo quindi utilizzare gli argomenti argc e argv all'interno del nostro codice per manipolare e utilizzare gli argomenti passati alla riga di comando. Ad esempio, possiamo utilizzare il valore di argc per determinare se sono stati passati argomenti alla riga di comando:

```c
if (argc > 1) {
  // codice da eseguire se sono stati passati argomenti alla riga di
     comando
} else {
  // codice da eseguire se non sono stati passati argomenti alla riga
     di comando
}
```

In questo caso, il codice all'interno del blocco dell'istruzione if verrà eseguito se sono stati passati argomenti alla riga di comando, altrimenti verrà eseguito il codice all'interno del blocco dell'istruzione else.

La funzione main è una parte fondamentale di ogni programma in C poiché viene eseguita quando il programma viene avviato. È importante assicurarsi di definire la funzione main nel nostro codice e di utilizzarla per inserire il codice principale del nostro programma.

3 variabili

In C, una variabile è un contenitore utilizzato per memorizzare un valore. Ogni variabile ha un nome univoco e un tipo di dati che determina il tipo di valori che può contenere.

Esempio:

```
int x = 5;
```

In questo esempio, abbiamo dichiarato una variabile chiamata x di tipo int, che può contenere valori interi. Abbiamo quindi assegnato alla variabile x il valore 5.

È possibile utilizzare le variabili all'interno del nostro codice per manipolare e utilizzare i valori che contengono. Ad esempio, possiamo utilizzare le variabili all'interno di espressioni matematiche:

```
int y = 7;
int z = x + y; // z conterrà 12
```

In questo caso, abbiamo dichiarato una nuova variabile chiamata y e l'abbiamo inizializzata con il valore 7. Abbiamo quindi utilizzato le variabili x e y all'interno di un'espressione matematica per calcolare il valore della variabile z. La variabile z conterrà quindi il valore 12, poiché 5 + 7 = 12.

È anche possibile utilizzare le variabili all'interno delle istruzioni di controllo come if e switch per eseguire codice in base al valore di una variabile:

```
if (x > 3) {
  printf("x è maggiore di 3\n");
} else {
  printf("x è minore o uguale a 3\n");
}
```

In questo caso, stiamo utilizzando il valore della variabile x all'interno dell'istruzione if per determinare se il codice all'interno del blocco dell'istruzione if verrà eseguito o meno. Poiché il valore della variabile x è maggiore di 3, il codice all'interno del blocco dell'istruzione if verrà eseguito e verrà stampato il messaggio "x è maggiore di 3".

Le variabili sono una parte fondamentale della programmazione in C, poiché permettono di memorizzare e manipolare i valori all'interno del nostro codice. È importante assicurarsi di utilizzare nomi di variabili significativi e di assegnare loro valori appropriati per garantire che il nostro codice funzioni correttamente.

4 if

In C, l'istruzione if è utilizzata per eseguire un blocco di codice solo se una determinata condizione è vera.

Esempio:

```
if (condizione) {
  // blocco di codice da eseguire se la condizione è vera
}
```

In questo esempio, il blocco di codice all'interno delle parentesi graffe verrà eseguito solo se la condizione specificata tra parentesi è vera. Ad esempio:

```
int x = 5;
if (x > 3) {
  printf("x è maggiore di 3\n");
}
```

In questo caso, il codice all'interno del blocco if verrà eseguito poiché la variabile x è effettivamente maggiore di 3.

È anche possibile utilizzare l'istruzione else per eseguire un blocco di codice diverso se la condizione non è vera:

```
int x = 5;
if (x > 3) {
  printf("x è maggiore di 3\n");
} else {
  printf("x è minore o uguale a 3\n");
}
```

In questo caso, il codice all'interno del blocco else non verrà eseguito poiché la variabile x è effettivamente maggiore di 3. Tuttavia, se la variabile x fosse stata minore o uguale a 3, il codice all'interno del blocco else sarebbe stato eseguito.

È inoltre possibile utilizzare più istruzioni if e else per creare una catena di condizioni:

```
int x = 5;
if (x > 5) {
  printf("x è maggiore di 5\n");
```

```
4 } else if (x > 3) {
5   printf("x è maggiore di 3 ma minore o uguale a 5\n");
6 } else {
7   printf("x è minore o uguale a 3\n");
8 }
```

In questo caso, il codice all'interno del primo blocco if non verrà eseguito poiché la variabile x non è maggiore di 5. Tuttavia, il codice all'interno del secondo blocco else if verrà eseguito poiché la variabile x è effettivamente maggiore di 3 ma minore o uguale a 5.

5 switch

In C, l'istruzione switch è utilizzata per eseguire un blocco di codice diverso in base al valore di una variabile.

Esempio:

```c
int x = 2;
switch (x) {
  case 1:
    // blocco di codice da eseguire se x è uguale a 1
    break;
  case 2:
    // blocco di codice da eseguire se x è uguale a 2
    break;
  default:
    // blocco di codice da eseguire se x non è uguale a nessuno dei
       valori specificati nei casi precedenti
    break;
}
```

In questo esempio, il blocco di codice all'interno del caso case 2 verrà eseguito poiché la variabile x è effettivamente uguale a 2. È importante notare che ogni caso deve terminare con l'istruzione break per interrompere l'esecuzione del codice e passare al codice successivo dopo l'istruzione switch. Inoltre, il blocco default viene eseguito se nessun altro caso è vero.

È anche possibile utilizzare l'istruzione switch per confrontare stringhe:

```c
char nome[50] = "Mario Rossi";
switch (nome) {
  case "Mario Rossi":
    // blocco di codice da eseguire se nome è uguale a "Mario Rossi"
    break;
  case "Luca Bianchi":
    // blocco di codice da eseguire se nome è uguale a "Luca Bianchi"
    break;
  default:
    // blocco di codice da eseguire se nome non è uguale a nessuno dei
       valori specificati nei casi precedenti
    break;
}
```

In questo caso, il blocco di codice all'interno del caso case "Mario Rossi" verrà eseguito poiché la stringa contenuta nella variabile nome è effettivamente uguale a "Mario Rossi".

L'istruzione switch può essere uno strumento molto utile in C per eseguire codice in base al valore di una variabile. Tuttavia, è importante assicurarsi di utilizzare l'istruzione break in ogni caso per evitare che il codice venga eseguito in modo imprevisto.

6 for

La costruzione for in C è un altro tipo di ciclo che viene utilizzato per eseguire un blocco di codice ripetutamente. A differenza del ciclo while, che verifica una determinata condizione prima di ogni iterazione del ciclo, il ciclo for include una serie di espressioni all'interno della sua sintassi che controllano il flusso del ciclo.

La sintassi di un ciclo for in C è la seguente:

```
for (espressione_iniziale; espressione_condizione;
   espressione_iterazione) {
   // Codice da eseguire ripetutamente
}
```

Le espressioni all'interno delle parentesi del ciclo for controllano il flusso del ciclo come segue:

- L'espressione iniziale viene eseguita una volta all'inizio del ciclo, prima della prima iterazione.
- L'espressione condizione viene verificata prima di ogni iterazione del ciclo. Se la condizione è verificata, il ciclo viene eseguito; altrimenti, il ciclo viene interrotto e il programma prosegue con l'esecuzione del codice successivo.
- L'espressione iterazione viene eseguita alla fine di ogni iterazione del ciclo.

Ecco un esempio di un ciclo for in C:

```
for (int i = 0; i < 5; i++) {
    // Codice da eseguire ripetutamente
    printf("%d\n", i);
}
```

Questo codice stamperà:

```
0
1
2
3
4
```

Come puoi vedere, il ciclo for include tre espressioni all'interno delle sue parentesi:

1. L'espressione iniziale (int i = 0) viene eseguita una volta all'inizio del ciclo, prima della prima iterazione. In questo caso, viene dichiarata una variabile intera i e viene assegnato il valore 0.

2. L'espressione condizione (i < 5) viene verificata prima di ogni iterazione del ciclo. Se la condizione è verificata, il ciclo viene eseguito; altrimenti, il ciclo viene interrotto e il programma prosegue con l'esecuzione del codice successivo. In questo caso, la condizione è verificata finché i è minore di 5.
3. L'espressione iterazione (i++) viene eseguita alla fine di ogni iterazione del ciclo. In questo caso, viene incrementato il valore di i di 1 ad ogni iterazione del ciclo.

7 while

Il costrutto **`while`** è una struttura di controllo che viene utilizzata nei linguaggi di programmazione per eseguire un blocco di codice ripetutamente fino a quando una condizione specificata è vera.

La sintassi del costrutto **`while`** in C è la seguente:

```
while (condizione) {
  codice da eseguire
}
```

La sezione di condizione viene controllata ad ogni iterazione del ciclo. Se la condizione è vera, il codice nel blocco viene eseguito. Se la condizione è falsa, il ciclo viene interrotto e il controllo passa alla linea di codice successiva.

Ecco un esempio di utilizzo del costrutto while in C++:

```
int i = 0;
while (i < 10) {
  cout << i << endl;
  i++;
}
```

In questo caso, il ciclo viene eseguito 10 volte, stampando i numeri da 0 a 9.

8 do-while

La costruzione do-while in C è un tipo di ciclo che viene utilizzato per eseguire un blocco di codice ripetutamente finché una determinata condizione è verificata. A differenza della costruzione while, che esegue il blocco di codice solo se la condizione è verificata, il ciclo do-while esegue sempre almeno una volta il blocco di codice prima di verificare la condizione.

La sintassi di un ciclo do-while in C è la seguente:

```
do {
    // Codice da eseguire ripetutamente
} while (condizione);
```

Ecco un esempio di codice do while:

```
int i = 0;
do {
    // Codice da eseguire ripetutamente
    printf("%d\n", i);
    i++;
} while (i < 5);
```

Questo codice stamperà:

```
0
1
2
3
4
```

Come puoi vedere, il ciclo do-while esegue il blocco di codice all'interno delle parentesi graffe {} almeno una volta, quindi verifica la condizione (i < 5) alla fine di ogni iterazione del ciclo. Se la condizione è verificata, il ciclo viene eseguito di nuovo; altrimenti, il ciclo viene interrotto e il programma prosegue con l'esecuzione del codice successivo.

9 array

In C, un array è una struttura di dati che ti permette di memorizzare una serie di elementi del medesimo tipo in un unico oggetto. Ad esempio, potresti creare un array di interi per memorizzare una serie di numeri interi, oppure un array di caratteri per memorizzare una stringa di testo.

Per dichiarare un array in C, devi specificare il tipo di dati degli elementi che desideri memorizzare, seguito dal nome dell'array e dalle dimensioni dell'array tra parentesi quadre:

```
int interi[10]; // Dichiara un array di interi di dimensione 10
char stringa[20]; // Dichiara un array di caratteri di dimensione 20
```

Una volta che hai dichiarato un array, puoi accedere ai singoli elementi dell'array usando l'operatore di indice. L'indice di un elemento dell'array inizia sempre da 0, quindi per accedere al primo elemento dell'array interi dichiarato sopra, ad esempio, useresti interi[0].

Ecco un esempio di come potresti utilizzare un array di interi per memorizzare e stampare una serie di numeri:

```
int interi[10]; // Dichiara un array di interi di dimensione 10

// Assegna alcuni valori all'array
interi[0] = 10;
interi[1] = 20;
interi[2] = 30;

// Stampa gli elementi dell'array
for (int i = 0; i < 10; i++) {
    printf("Elemento %d: %d\n", i, interi[i]);
}
```

Questo codice stamperà:

```
Elemento 0: 10
Elemento 1: 20
Elemento 2: 30
Elemento 3: 0
Elemento 4: 0
Elemento 5: 0
Elemento 6: 0
Elemento 7: 0
```

```
 9  Elemento 8: 0
10  Elemento 9: 0
```

10 stringhe

In linguaggio C, una stringa è una sequenza di caratteri contigui memorizzati in memoria con un carattere terminatore di stringa, il carattere NULL ('\0'). La libreria string.h fornisce funzioni per la manipolazione delle stringhe, tra cui la copia, la concatenazione, la ricerca e la comparazione.

Ecco alcuni esempi di dichiarazione di stringhe in C:

```
char str1[] = "hello"; // dichiarazione di una stringa inizializzata
char str2[10]; // dichiarazione di una stringa non inizializzata con
    una lunghezza massima di 10 caratteri
```

Per eseguire operazioni sulle stringhe, è necessario includere la libreria string.h. Ecco alcuni esempi di funzioni fornite dalla libreria:

```
strcpy(dest, src): copia la stringa src nella stringa dest.
```

```
char src[] = "hello";
char dest[10];
strcpy(dest, src);
printf("dest: %s\n", dest); // output: dest: hello
strcat(dest, src): concatena la stringa src alla fine della stringa
    dest.
```

```
char dest[10] = "hello";
char src[] = "world";
strcat(dest, src);
printf("dest: %s\n", dest); // output: dest: helloworld
```

In particolare, `strlen(str)` restituisce la lunghezza della stringa str, escluso il carattere terminatore di stringa.

```
char str[] = "hello";
int len = strlen(str);
printf("length: %d\n", len); // output: length: 5
```

`strcmp(str1, str2)`: confronta le due stringhe str1 e str2 e restituisce un valore intero che indica la relazione tra le due stringhe. Restituisce 0 se le stringhe sono uguali, un valore positivo se str1 è maggiore di str2, e un valore negativo se str1 è minore di str2.

```c
char str1[] = "hello";
char str2[] = "world";
int cmp = strcmp(str1, str2);
if (cmp == 0) {
    printf("le stringhe sono uguali\n");
} else if (cmp > 0) {
    printf("str1 è maggiore di str2\n");
} else {
    printf("str1 è minore di str2\n");
}
```

Questi sono solo alcuni esempi delle funzioni fornite dalla libreria string.h in C. Ci sono molte altre funzioni utili per la manipolazione delle stringhe, come strstr() per la ricerca di una sottostringa in una stringa, strtok() per la suddivisione di una stringa in sottostringhe, e strchr() per la ricerca di un carattere in una stringa.

11 struct

In C, una struct è una struttura dati utilizzata per organizzare dati di tipi diversi in un unico blocco.

Esempio:

```
struct persona {
  char nome[50];
  int età;
  float altezza;
};
```

In questo esempio, abbiamo definito una struct chiamata persona che contiene tre campi: nome, età e altezza. Ogni campo può contenere un valore di un tipo di dati diverso.

Per utilizzare una struct in un programma, è necessario dichiararne una variabile:

```
struct persona p1;
```

In questo caso, abbiamo dichiarato una variabile chiamata p1 che è una struct persona. Possiamo quindi accedere ai campi della struct utilizzando l'operatore di accesso ai membri (.):

```
p1.nome = "Mario Rossi";
p1.età = 30;
p1.altezza = 1.75;
```

In questo caso, abbiamo assegnato valori ai campi nome, età e altezza della struct p1. Possiamo quindi utilizzare questi valori nella nostra logica di programma:

```
printf("%s ha %d anni e %.2f di altezza\n", p1.nome, p1.età, p1.altezza
    );
```

In questo caso, stiamo utilizzando i valori delle variabili nome, età e altezza all'interno del nostro comando printf per stampare i valori della struct p1.

È inoltre possibile utilizzare una struct come tipo di dati per una funzione:

```
struct persona crea_persona(char *nome, int età, float altezza) {
  struct persona p;
  p.nome = nome;
  p.età = età;
  p.altezza = altezza;
```

```
6     return p;
7   }
```

In questo caso, abbiamo creato una funzione crea_persona che accetta tre argomenti: nome, età e altezza. La funzione crea una nuova struct persona e assegna ai suoi campi i valori degli argomenti. La funzione restituisce poi la struct persona. Possiamo quindi utilizzare i valori delle variabili nome, età e altezza all'interno del nostro comando printf per stampare i valori della struct p1.

È inoltre possibile utilizzare una struct come tipo di dati per una funzione:

```
1  struct persona crea_persona(char *nome, int età, float altezza) {
2    struct persona p;
3    p.nome = nome;
4    p.età = età;
5    p.altezza = altezza;
6    return p;
7  }
```

In questo caso, abbiamo creato una funzione crea_persona che accetta tre argomenti: nome, età e altezza. La funzione crea una nuova struct persona e assegna ai suoi campi i valori degli argomenti. La funzione restituisce poi la struct persona. Possiamo quindi utilizzare questa funzione per creare nuove struct persona:

```
1  struct persona p1 = crea_persona("Mario Rossi", 30, 1.75);
2  struct persona p2 = crea_persona("Luca Bianchi", 28, 1.80);
```

In questo caso, abbiamo creato due nuove struct persona utilizzando la funzione crea_persona e assegnando i valori restituiti alle variabili p1 e p2. Possiamo quindi utilizzare queste struct persona all'interno del nostro programma come desiderato.

12 funzioni

In C, una funzione è una porzione di codice che viene eseguita in risposta a una determinata richiesta. Le funzioni sono utilizzate per suddividere il codice in blocchi riutilizzabili che possono essere richiamati da altre parti del programma.

Per dichiarare una funzione in C, devi specificare il tipo di dati del valore restituito dalla funzione, seguito dal nome della funzione e dai suoi argomenti tra parentesi:

```
int somma(int x, int y) {
    // Codice della funzione
    return x + y;
}
```

In questo esempio, la funzione somma riceve due argomenti di tipo int (x e y) e restituisce un valore di tipo int che rappresenta la somma di x e y.

Per richiamare una funzione in C, devi semplicemente usare il suo nome seguito dai suoi argomenti tra parentesi:

```
int main() {
    int a = 5, b = 7;
    int risultato = somma(a, b); // Richiama la funzione somma
    printf("La somma di %d e %d è %d\n", a, b, risultato);
    return 0;
}
```

Questo codice stamperà:

```
La somma di 5 e 7 è 12
```

Come puoi vedere, la funzione somma viene dichiarata all'inizio del codice, specificando il tipo di dati del valore restituito (int) e i suoi argomenti (due interi chiamati x e y). Il codice all'interno delle parentesi graffe {} rappresenta il corpo della funzione, che viene eseguito quando la funzione viene richiamata.

Nel main(), la funzione somma viene richiamata passando come argomenti i valori di a e b. La funzione esegue il suo codice, che in questo caso consiste nella somma di x e y e nella restituzione del risultato, che viene assegnato alla variabile risultato. Quindi, il valore di risultato viene stampato a schermo.

13 Ricorsione

Le funzioni ricorsive in C sono funzioni che si richiamano da sole, ovvero una funzione richiama se stessa all'interno del proprio corpo.

In pratica, il funzionamento è il seguente: la funzione esegue una determinata operazione su un input, dopodiché si richiama passando come input una versione modificata dell'input originale. Questa operazione si ripete finché non si raggiunge una condizione di terminazione, ovvero un caso base che impedisce alla funzione di richiamarsi ulteriormente.

Ogni funzione ricorsiva è composta da due componenti principali:

- **Caso base**: condizione che impedisce alla funzione di richiamarsi ulteriormente. Se non viene specificata, la funzione si richiamerà all'infinito.
- **Caso ricorsivo**: chiamata ricorsiva della funzione, che passa come input una versione modificata dell'input originale.

Per esempio, consideriamo la funzione fattoriale:

```c
int fattoriale(int n) {
    if (n == 0) {  // caso base
        return 1;
    } else {
        return n * fattoriale(n-1);  // chiamata ricorsiva
    }
}
```

In questo caso, se n è uguale a 0, la funzione ritorna 1, altrimenti ritorna n moltiplicato per il fattoriale di n-1, ovvero la stessa funzione chiamata con un input diverso. Questo processo si ripete finché non si raggiunge il caso base, ovvero n = 0.

Un altro esempio potrebbe essere la funzione Fibonacci:

```c
int fibonacci(int n) {
    if (n == 0 || n == 1) {  // casi base
        return n;
    }

    else {
        return fibonacci(n-1) + fibonacci(n-2);  // chiamate ricorsive
```

```
8       }
9   }
```

In questo caso, se n è uguale a 0 o 1, la funzione ritorna direttamente n, altrimenti ritorna la somma dei due numeri di Fibonacci precedenti, ovvero la stessa funzione chiamata con n-1 e n-2. Anche qui, il processo si ripete finché non si raggiunge un caso base.

Tuttavia, è importante notare che le funzioni ricorsive possono portare a problemi di memoria se non gestite correttamente. Ogni volta che una funzione si richiama, viene creata una nuova istanza della funzione, che viene mantenuta in memoria fino alla fine della chiamata ricorsiva. Se la ricorsione è troppo profonda o se la funzione richiede molta memoria, si può arrivare a un esaurimento della memoria disponibile (stack overflow).

Quindi, è importante assicurarsi che la ricorsione termini in modo appropriato e che non ci siano chiamate ricorsive infinite o troppo profonde. Inoltre, in alcuni casi, le funzioni iterative possono essere più efficienti e meno rischiose delle funzioni ricorsive.

14 file

In **C**, come in molti altri linguaggi di programmazione, la gestione dei file viene effettuata tramite delle funzioni specifiche che permettono di aprire, leggere, scrivere e chiudere un file.

Suggerimento: puoi pensare ai file come dei "pacchi". Prima di utileizzare un pacco, è necessario aprirlo, poi leggere il suo contenuto, scrivere qualcosa all'interno e infine chiuderlo (va sempre chiuso prima di essere spedito!).

14.1 Aprire un file

Per aprire un file, si può utilizzare la funzione 'fopen(), che prende come argomenti il nome del file e la modalità di apertura (lettura, scrittura, lettura e scrittura, etc.). Ad esempio, per aprire un file chiamato "test.txt" in modalità di lettura, si può utilizzare il seguente codice:

```c
FILE *file_ptr;
file_ptr = fopen("test.txt", "r");
```

I file possono essere aperti in varie modalità:

- **lettura**: `r`, consente la lettura del file.
- **scrittura**: `w`, consente la scrittura sul file. Se il file esiste già, il suo contenuto viene sovrascritto.
- **append**: `a`, consente la scrittura sul file. Se il file esiste già, il nuovo contenuto viene aggiunto in coda al file.

Analogia: Hai aperto il pacco per poi utilizzarlo in seguito.

14.2 Leggere un file

Il **C** mette a disposizioni molte funzioni per leggere i dati da un file, di seguito ne sono illustrate alcune.

Analogia: stai guardando il contenuto del pacco.

14.2.1 `fread()`

Per leggere il contenuto di un file aperto, si può utilizzare la funzione `fread()`, che prende come argomenti il puntatore al file, la dimensione di ogni elemento da leggere, il numero di elementi da leggere e un puntatore all'area di memoria in cui verranno scritti i dati letti dal file. Ad esempio, per leggere un array di interi dal file "test.txt" aperto in precedenza, si può utilizzare il seguente codice:

```
1 int array[100];
2 fread(array, sizeof(int), 100, file_ptr);
```

14.2.2 `getline()`

Per leggere il contenuto di un file aperto riga per riga è possibile utilizzare la funzione `getline()`, che prende come argomenti il puntatore all'area di memoria in cui verrà scritta la riga letta, il numero massimo di caratteri da leggere e il puntatore al file da cui leggere. Ad esempio, per leggere il contenuto del file "test.txt" aperto in precedenza riga per riga, si può utilizzare il seguente codice:

```
1 char * line = NULL;
2 size_t len = 0;
3 ssize_t read;
4
5 // fino a quando il file non finisce
6 while ((read = getline(&line, &len, fp)) != -1) {
7         printf("%s", line); // stampa la riga letta
8 }
```

14.3 Scrivere su un file

Il **C** mette a disposizioni molte funzioni per leggere i dati da un file, di seguito ne sono illustrate alcune.

Analogia: Stai inserendo delle informazioni all'interno del pacco.

14.3.1 `fwrite()`

Per scrivere su un file aperto, si può utilizzare la funzione `fwrite()`, che prende come argomenti il puntatore all'area di memoria contenente i dati da scrivere, la dimensione di ogni elemento da scrivere, il numero di elementi da scrivere e un puntatore al file su cui scrivere. Ad esempio, per scrivere l'array di interi precedentemente letto sul file "test.txt", si può utilizzare il seguente codice:

```
1 fwrite(array, sizeof(int), 100, file_ptr);
```

14.3.2 `fprintf()`

Per scrivere su un file aperto, si può utilizzare la funzione `fprintf()`, che prende come argomenti il puntatore al file su cui scrivere e una stringa contenente il testo da scrivere. Ad esempio, per scrivere una stringa sul file "test.txt" aperto in precedenza, si può utilizzare il seguente codice:

```
1 fprintf(file_ptr, "Hello World!");
```

14.4 Chiudere un file

Per chiudere un file aperto, si può utilizzare la funzione `fclose()`, che prende come argomento il puntatore al file da chiudere. Ad esempio, per chiudere il file "test.txt" aperto in precedenza, si può utilizzare il seguente codice:

```
1 fclose(file_ptr);
```

Analogia: Stai chiudendo il pacco per poterlo spedire.

Importante: Ricorda sempre di chiudere un file dopo aver finito di utilizzarlo. Per poterlo riusare sarà poi necessario riaprilo.

15 algoritmi di ordinamento

Gli algoritmi di ordinamento vengono utilizzati per ordinare il contenuto di un vettore a seconda di un qualche criterio (ad esempio in modo crescente, decrescente ecc).

Esistono più algoritmi che consentono di fare ciò, con differenti livelli di complessità e velocità. In questo capitolo vedremo alcuni di questi algoritmi.

15.1 Selection Sort

Il **Selection Sort** è un algoritmo di ordinamento in cui si cerca il valore minimo (o massimo) dell'array e lo si scambia con il primo elemento dell'array, poi si cerca il valore minimo tra i restanti elementi dell'array e lo si scambia con il secondo elemento dell'array, e così via, fino a quando l'intero array è ordinato.

L'algoritmo *Selection Sort* ha una complessità computazionale di $O(n^2)$ nel caso peggiore e medio, dove n è la dimensione dell'array, poiché per ogni elemento dell'array vengono eseguiti n-1 confronti.

Ecco un esempio di implementazione del Selection Sort in linguaggio C:

```c
#include <stdio.h>

void selection_sort(int arr[], int n) {
    int i, j, min_idx, temp;

    for (i = 0; i < n-1; i++) {
        min_idx = i; // assume che il valore minimo sia il primo
           elemento non ordinato
        for (j = i+1; j < n; j++) {
            // cerca il valore minimo tra gli elementi non ordinati
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        // scambia il valore minimo con il primo elemento non ordinato
        temp = arr[i];
        arr[i] = arr[min_idx];
```

```
17          arr[min_idx] = temp;
18      }
19  }
20
21  int main() {
22      int arr[] = {64, 25, 12, 22, 11};
23      int n = sizeof(arr) / sizeof(arr[0]);
24
25      printf("Array non ordinato: ");
26      for (int i = 0; i < n; i++) {
27          printf("%d ", arr[i]);
28      }
29
30      selection_sort(arr, n);
31
32      printf("\nArray ordinato: ");
33      for (int i = 0; i < n; i++) {
34          printf("%d ", arr[i]);
35      }
36
37      return 0;
38  }
```

In questo esempio, l'array `{64, 25, 12, 22, 11}` viene ordinato in ordine crescente utilizzando l'algoritmo Selection Sort. Il ciclo esterno for attraversa l'array e seleziona un elemento alla volta. Il ciclo interno for cerca il valore minimo tra gli elementi non ordinati dell'array. Se viene trovato un valore minimo, il valore minimo viene scambiato con il primo elemento non ordinato. Alla fine del ciclo esterno, l'array sarà ordinato in ordine crescente.

15.2 bubbleSort

Il **Bubble Sort** è un algoritmo di ordinamento molto semplice da implementare e comprendere, ma non è il più efficiente per grandi insiemi di dati. Tuttavia, è comunque utile per scopi didattici e per ordinare piccoli insiemi di dati.

In pratica, il Bubble Sort confronta ripetutamente coppie di elementi adiacenti nell'array da ordinare, e se la coppia non è ordinata, scambia i due elementi. Questo processo viene ripetuto finché non si verificano più scambi, ovvero l'array è stato completamente ordinato.

Di seguito riporto un esempio di implementazione del Bubble Sort in C:

```
1  #include <stdio.h>
2
3  void bubble_sort(int array[], int n) {
4      int temp, i, j;
5      for(i = 0; i < n - 1; i++) {
```

```
        for(j = 0; j < n - i - 1; j++) {
            if(array[j] > array[j + 1]) {
                // Scambia gli elementi
                temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
}

int main() {
    int array[] = {10, 2, 5, 8, 1};
    int n = sizeof(array) / sizeof(int);

    printf("Array non ordinato: ");
    for(int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }

    bubble_sort(array, n);

    printf("\nArray ordinato: ");
    for(int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }

    return 0;
}
```

Nell'esempio sopra, la funzione `bubble_sort()` prende come argomenti l'array da ordinare e la sua dimensione n. All'interno della funzione, due cicli for vengono utilizzati per scorrere l'array e confrontare le coppie di elementi adiacenti. Se la coppia non è ordinata, gli elementi vengono scambiati.

Nella funzione `main()`, viene creato un array di interi non ordinato e la sua dimensione viene calcolata utilizzando l'operatore `sizeof()`. L'array viene poi stampato prima dell'ordinamento, e dopo aver chiamato la funzione bubble_sort(), viene stampato l'array ordinato.

Ecco il risultato dell'esecuzione del programma:

```
Array non ordinato: 10 2 5 8 1
Array ordinato: 1 2 5 8 10
```

la versione ottimizzata del Bubble Sort con sentinella è un'altra variante dell'algoritmo che può migliorare le prestazioni del Bubble Sort. Invece di confrontare tutti gli elementi adiacenti in ogni iterazione, questa variante utilizza una sentinella, ovvero un valore di controllo che viene inserito alla fine dell'array e viene utilizzato per interrompere il ciclo for interno quando non ci sono più scambi da fare.

L'utilizzo della sentinella può ridurre il numero di confronti effettuati e quindi migliorare le prestazioni del Bubble Sort. Ecco un esempio di implementazione del Bubble Sort con sentinella in C:

```c
#include <stdio.h>

void bubble_sort_sentinel(int array[], int n) {
    int i = 0, j, temp;
    int scambiati = 1;

    while(i < n - 1 && scambiati) {
        scambiati = 0;
        for(j = 0; j < n - i - 1; j++) {
            if(array[j] > array[j + 1]) {
                // Scambia gli elementi
                temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
                scambiati = 1;
            }
        }
        i++;
    }
}

int main() {
    int array[] = {10, 2, 5, 8, 1};
    int n = sizeof(array) / sizeof(int);

    printf("Array non ordinato: ");
    for(int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }

    bubble_sort_sentinel(array, n);

    printf("\nArray ordinato: ");
    for(int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }

    return 0;
}
```

In questa implementazione, la funzione `bubble_sort_sentinel()` prende come argomenti l'array da ordinare e la sua dimensione n. La variabile i viene utilizzata per tenere traccia del numero di passaggi fatti nell'array, mentre la variabile scambiati viene utilizzata per verificare se ci sono stati scambi nell'ultimo passaggio. La sentinella viene aggiunta alla fine dell'array utilizzando n - 1 come indice.

All'interno del ciclo while, viene eseguito il ciclo for interno solo se ci sono stati scambi nell'ultimo passaggio e se non si è ancora raggiunto la fine dell'array. Se non ci sono stati scambi nell'ultimo passaggio, la variabile scambiati viene impostata su 0 e il ciclo while termina.

Nella funzione main(), viene creato un array di interi non ordinato e la sua dimensione viene calcolata utilizzando l'operatore `sizeof()`. L'array viene poi stampato prima dell'ordinamento, e dopo aver chiamato la funzione `bubble_sort_sentinel()`, viene stampato l'array ordinato.