2 - Programmazione modulare

La programmazione modulare è un paradigma fondamentale nell'informatica che si basa sull'idea di suddividere un programma complesso in blocchi più piccoli e indipendenti, chiamati moduli. Questi moduli, spesso implementati come funzioni o procedure, permettono di gestire la complessità del codice, migliorarne la leggibilità e favorire il riutilizzo.

Cos'è la Programmazione Modulare?

La programmazione modulare rappresenta un approccio strutturato allo sviluppo software, dove un programma viene scomposto in sotto-programmi o frammenti di codice distinti dal programma principale (spesso il `main()`). Questi frammenti, noti come funzioni o procedure, sono definiti e invocati attraverso meccanismi specifici forniti dai linguaggi di programmazione.

L'astrazione è un concetto chiave in questo contesto, si riferisce alla capacità di isolare un concetto o un oggetto dal suo contesto per analizzarlo in modo indipendente.
In informatica, l'astrazione permette di definire oggetti generali a partire da casi particolari, nascondendo i dettagli implementativi e focalizzandosi su "cosa" fa un modulo piuttosto che su "come" lo fa.
Le funzioni e le procedure sono un meccanismo di astrazione tra loro, infatti:

- Le funzioni sono un esempio di astrazione sui dati, perchè ci permettono di estendere gli operatori disponibili nel linguaggio
- Le procedure sono un esempio di astrazione sulle istruzioni, perchè ci permettono di estendere le istruzioni primitive disponibili nel linguaggio

Vantaggi della Programmazione Modulare

La programmazione modulare presenta diversi vantaggi come:

1. **Leggibilità**: Un codice suddiviso in moduli è più facile da comprendere. Ad esempio, una funzione `calcolaBMI()` è più intuitiva rispetto a una serie di operazioni matematiche direttamente nel `main()`.
2. **Astrazione**: Nasconde i dettagli implementativi, permettendo agli sviluppatori di concentrarsi sulla logica generale.
3. **Riusabilità**: Le funzioni possono essere riutilizzate in diversi contesti o progetti, evitando duplicazioni di codice.
4. **Modularità**: Ogni modulo svolge un compito specifico e può essere testato autonomamente.

Paradigma Divide-et-Impera

La programmazione modulare implementa il principio "divide et impera", che consiste nel suddividere un problema complesso in sotto-problemi più semplici. Questo approccio può essere applicato in due modi:

- **Top-down**: Si parte dal problema generale e lo si scompone in sotto-problemi, implementando un modulo per ciascuno.
- **Bottom-up**: Si parte da moduli già esistenti e li si combina per risolvere problemi più complessi.

Principi della Programmazione Modulare

1. **Suddivisione in moduli**: Un programma deve essere organizzato in moduli indipendenti.
2. **Indipendenza**: I moduli devono essere il più possibile autonomi, con interfacce ben definite.
3. **Testabilità**: Ogni modulo deve poter essere testato singolarmente.
4. **Interfaccia chiara**: Ogni modulo deve specificare chiaramente input e output.

Esempi Pratici

Un esempio classico è il calcolo del BMI (Body Mass Index). Invece di scrivere tutto il codice nel `main()`, è preferibile definire una funzione dedicata:

```
float calcolaBMI(float altezza, float peso) {
    return peso / ((altezza / 100) * (altezza / 100));
}
```

Questa funzione può essere riutilizzata in contesti diversi, come un sistema gestionale per una palestra o un'applicazione medica, dimostrando la flessibilità della programmazione modulare.

Progettazione Modulare

La progettazione modulare richiede un'attenta analisi per identificare i moduli necessari e definirne le interfacce. È fondamentale considerare:

- Lo scopo di ogni modulo.
- I dati di input e output.
- La generalità della soluzione (ad esempio, parametrizzare una soglia nel calcolo di un voto).
  Un esempio di programmazione modulare sono le istruzioni #include, perché stiamo riutilizzando funzioni non previste nel C standard che sono state scritte da altri programmatori.

Information Hiding e Separazione delle Competenze

L'information hiding è un principio che consiste nel nascondere i dettagli implementativi di un modulo, esponendo solo ciò che è necessario. Ad esempio, la funzione `calcolaBMI()` non deve occuparsi di acquisire input o stampare output, ma solo di eseguire il calcolo.

La separazione delle competenze implica che ogni modulo deve avere un ruolo ben definito. Una funzione che calcola il BMI non deve anche stamparlo a schermo, poiché ciò violerebbe questo principio.

Funzioni e procedure

Le funzioni e le procedure sono gli elementi costitutivi della programmazione modulare, permettendo di organizzare il codice in blocchi logici e riutilizzabili. Entrambe rappresentano sotto-programmi che svolgono compiti specifici, ma differiscono nel modo in cui restituiscono i risultati.

Differenza tra Funzioni e Procedure

Le **funzioni** sono sotto-programmi che accettano input (parametri), elaborano dati e restituiscono un valore di output. Ad esempio, una funzione che calcola il quadrato di un numero restituisce il risultato del calcolo.

Le **procedure**, invece, sono funzioni che non restituiscono alcun valore. In linguaggio C, questo viene indicato con il tipo di ritorno `void`. Un esempio è una procedura che stampa un messaggio a schermo senza restituire dati al chiamante.

Struttura di una Funzione

Ogni funzione è caratterizzata da quattro elementi principali:

1. **Nome**: Identificatore univoco utilizzato per invocare la funzione.
2. **Parametri**: Variabili di input necessarie per l'esecuzione della funzione.
3. **Valore di ritorno**: Risultato restituito dopo l'elaborazione.
4. **Implementazione**: Blocco di codice che definisce le operazioni da eseguire.

I primi tre elementi costituiscono il **prototipo della funzione**, che ne descrive l'interfaccia senza entrare nei dettagli implementativi.

Esempio di Funzione

```c
int somma(int a, int b) {
    return a + b;
}
```

In questo esempio:

- `somma` è il nome della funzione.
- `a` e `b` sono i parametri di input.
- `int` è il tipo di dato restituito.
- `return a + b` è l'implementazione.

Prototipi e Chiamate a Funzione

Il **prototipo** di una funzione ne dichiara la firma (nome, parametri e tipo di ritorno) prima della sua implementazione. Questo permette al compilatore di verificare la correttezza delle chiamate.

```c
int somma(int a, int b);  // Prototipo

int main() {
    int risultato = somma(3, 5);  // Chiamata
    printf("%d", risultato);
    return 0;
}

int somma(int a, int b) {  // Implementazione
    return a + b;
}
```

La **chiamata a funzione** trasferisce il controllo dal programma principale alla funzione, legando i parametri attuali (valori passati) a quelli formali (definiti nel prototipo).

I parametri quindi si suddividono in questa maniera nel codice:

- **Parametro formale**: Variabile definita nell'intestazione della funzione (es. `int x` in `void raddoppia(int x)`).
- **Parametro attuale**: Valore concreto passato durante la chiamata (es. `num` in `raddoppia(num)`).

Best Practices

1. **Nomi significativi**: Scegliere nomi descrittivi per funzioni e parametri (es. `calcolaMedia` invece di `func1`).
2. **Commenti**: Documentare lo scopo di ogni funzione e il significato dei parametri.
3. **Coesione**: Ogni funzione dovrebbe svolgere un solo compito ben definito.

Passaggi di valore

Nella programmazione modulare, il meccanismo con cui i dati vengono trasferiti tra il programma chiamante e le funzioni costituisce un aspetto cruciale per comprendere il comportamento del codice. In C esistono due modalità distinte di passaggio parametri che

determinano radicalmente l'interazione tra le diverse parti del programma: il passaggio per valore e il passaggio per riferimento.

Passaggio per Valore

Il passaggio per valore rappresenta la modalità predefinita in linguaggio C e si caratterizza per la creazione di copie locali dei parametri all'interno della funzione chiamata. Quando si invoca una funzione con parametri passati per valore, il sistema alloca nuova memoria nello stack per le variabili locali della funzione, copiando in queste locazioni i valori degli argomenti forniti.

Questo approccio garantisce un completo isolamento tra il contesto della funzione chiamata e quello del chiamante. Le modifiche apportate ai parametri all'interno della funzione rimangono confinate allo scope locale e non hanno alcun effetto sulle variabili originali del programma chiamante. Questo comportamento è particolarmente utile quando si vogliono proteggere i dati originali da modifiche accidentali.

Un esempio pratico dimostra chiaramente questo concetto:

```c
void modificaValore(int x) {
    x = x * 2;
    printf("Dentro la funzione: %d\n", x);
}

int main() {
    int num = 5;
    modificaValore(num);
    printf("Fuori la funzione: %d\n", num);
    return 0;
}
```

L'output di questo programma mostrerà che il valore di `num` nel main rimane invariato nonostante la modifica all'interno della funzione, confermando che le due variabili, seppure con lo stesso nome, occupano locazioni di memoria distinte.

Passaggio per Riferimento

Diversamente dal passaggio per valore, il passaggio per riferimento permette alla funzione chiamata di operare direttamente sulla memoria originale del chiamante. In C questo risultato si ottiene utilizzando i puntatori, che consentono di lavorare con gli indirizzi di memoria anziché con copie dei valori.

Quando si passa un parametro per riferimento, si fornisce alla funzione non il valore della variabile, ma il suo indirizzo in memoria. Questo approccio permette alla funzione di modificare direttamente il contenuto della locazione di memoria originale, con effetti permanenti che persistono dopo il ritorno della funzione.

L'implementazione tipica prevede l'uso dell'operatore `&` per ottenere l'indirizzo della variabile al momento della chiamata e dell'operatore `*` per dereferenziare il puntatore all'interno della funzione:

```c
void modificaRiferimento(int *x) {
    *x = *x * 2;
    printf("Dentro la funzione: %d\n", *x);
}

int main() {
    int num = 5;
    modificaRiferimento(&num);
    printf("Fuori la funzione: %d\n", num);
    return 0;
}
```

In questo caso l'output mostrerà che il valore di `num` è stato effettivamente modificato dalla funzione, dimostrando che le operazioni sono state eseguite sulla stessa locazione di memoria.

Il Caso Particolare degli Array

Un'eccezione significativa alle regole generali del passaggio parametri in C riguarda gli array. Quando un array viene passato a una funzione, il linguaggio C adotta automaticamente un comportamento simile al passaggio per riferimento, anche senza l'uso esplicito di puntatori. Questo avviene perché il nome di un array rappresenta di per sé l'indirizzo del suo primo elemento.

La conseguenza pratica è che qualsiasi modifica apportata agli elementi dell'array all'interno della funzione si rifletterà sull'array originale del chiamante:

```c
void modificaArray(int arr[], int size) {
    for(int i = 0; i < size; i++) {
        arr[i] *= 2;
    }
}

int main() {
    int numeri[] = {1, 2, 3};
    modificaArray(numeri, 3);
    printf("%d %d %d\n", numeri[0], numeri[1], numeri[2]);
    return 0;
}
```

L'output mostrerà che tutti gli elementi dell'array sono stati raddoppiati, confermando che le modifiche sono state applicate alla struttura dati originale.

Gestione della Memoria

La differenza fondamentale tra i due meccanismi di passaggio risiede nella gestione della memoria. Nel passaggio per valore, il sistema alloca nuova memoria per le variabili locali della funzione, mentre nel passaggio per riferimento si lavora direttamente sulla memoria esistente.

Questa distinzione ha importanti implicazioni per:

- L'efficienza: il passaggio per riferimento evita la copia di grandi strutture dati
- La sicurezza: il passaggio per valore protegge i dati originali
- Le prestazioni: per grandi quantità di dati, il passaggio per riferimento è più efficiente
- La flessibilità: il passaggio per riferimento permette di restituire più valori

Scelta del Meccanismo Appropriato

La decisione tra passaggio per valore e passaggio per riferimento dipende dai requisiti specifici del programma. Come regola generale:

1. Si preferisce il passaggio per valore quando:
    - I dati di input non devono essere modificati
    - Si lavora con tipi primitivi di piccole dimensioni
    - Si vuole garantire l'isolamento della funzione
2. Si preferisce il passaggio per riferimento quando:
    - È necessario modificare i parametri originali
    - Si lavora con strutture dati complesse o di grandi dimensioni
    - Si vogliono ottimizzare le prestazioni evitando copie inutili

Classi di memoria

La classe di memoria si applica ad un identificatore e determina:

- **La sua permanenza in memoria**, ovvero il periodo in quale l'identificatore esiste in memoria (può essere statica o automatica)
- Il suo **campo d'azione o scope**, praticamente dove l'identificatore può essere menzionato in un programma
- **Il suo collegamento**, quando un programma è composto da più file, viene riconosciuto se l'identificatore è conosciuto solo nel file sorgente corrente o in qualunque file sorgente ad esso collegato

Permanenza in memoria

In C, esistono degli attributi per definire la permanenza in memoria

- **Auto**: valore di default per tutte le variabili locali, imposta la permanenza in memoria pari al ciclo di vita del blocco
- **Extern**: valore di default per le variabili globali, permanenza pari all'intera esecuzione del file
- **Static**: la permanenza è pari all'esecuzione della funzione ma il suo valore non viene distrutto quando termina l'esecuzione
- **Register (Obsoleto)**: Memorizza la variabile in registri ad alta velocità, per velocizzarne l'accesso.

N.B.: **le variabili globali** possono causare effetti secondari non voluti quando una funzione che non ha necessità di accedere alla variabile la modifica accidentalmente, quindi vanno **generalmente evitate**

Campo d'azione

Lo scope (o campo di azione) di una variabile è il frammento di codice in cui è visibile (nota al compilatore e può essere utilizzata nel codice senza produrre errori)

In regola generale una variabile è visibile **nel blocco in cui è definita e in tutti i blocchi innestati a meno di redifinizioni**.
Esistono diversi tipi di campi d'azione:

- **File scope**: tutti gli identificatori definiti fuori dalle funzioni sono visibili in tutto il file, si usa per i **prototipi di funzione** (devono essere richiamati in ogni momento) e le **variabili globali** (visibili e utilizzabili in tutto il codice sorgente)
- **Function scope**: gli identificatori definiti nel corpo di una funzione sono visibili solo in quella funzione (a meno di ridefinizioni in un blocco)
- **Block scope**: Tutti gli identificatori definiti in un **blocco**, sono visibili solo in quel **blocco**
- **Function Prototype Scope**: Gli identificatori definiti nel prototipo di una funzione valgono solo in esso. (possono anche essere omessi, non è detto che serva ri-specificarlo)

Un esempio di tutti i campi d'azione è:

```
void function(int minnie) //Visibilità nel prototipo, File scope
int pluto //Visibilità globale, File scope

int main(){
int pippo=0 //Visibilità locale, Function Scope
if (pippo==0){
    int topolino=pippo //Visibilità nel blocco, Block Scope
}
}
```

Suggerimenti

È importante utilizzare il principio dell'information hiding, poiché ogni oggetto, per motivi di sicurezza del software e per facilitare la risoluzione di errori, necessita della protezione dei dati a un livello più interno e meno esposto;
Questo è il principio del minimo privilegio, **deve essere sempre garantita la quantità di privilegi necessaria** per eseguire il suo compito disegnato ma non di più.

Header files

L'utilizzo delle funzioni e delle procedure risolve **parzialmente** il problema della programmazione modulare, perchè il codice sorgente è comunque tutto aggregato in un unico file (anche se spacchettato in diverse funzioni).Per implementare totalmente i principi della programmazione modulare è necessario anche dividere **fisicamente** il codice sorgente, per poterlo farlo possiamo utilizzare gli **Header Files** e le Librerie statiche

Gli header files sono file di intestazione, hanno estensione **.h** e sono seguiti da un file **.c** con lo stesso nome, contengono le intestazioni delle funzioni e delle procedure che vogliamo separare dal programma principale.
Si creano uno o più nuovi file e si inseriscono le funzioni in questi file, le funzioni vengono aggregate in diversi header file in base al loro scopo (es. tutte le funzioni che si occupano di input/output, tutte le funzioni per operazioni matematiche, etc.), un po' come avviene funzioni della libreria standard del C (es. <string.h> <ctype.h> etc.)

Nell'header file andiamo a inserire solo i prototipi di funzione

```c
#include "function.c"
int f(int x); // prototipo
```
function.h

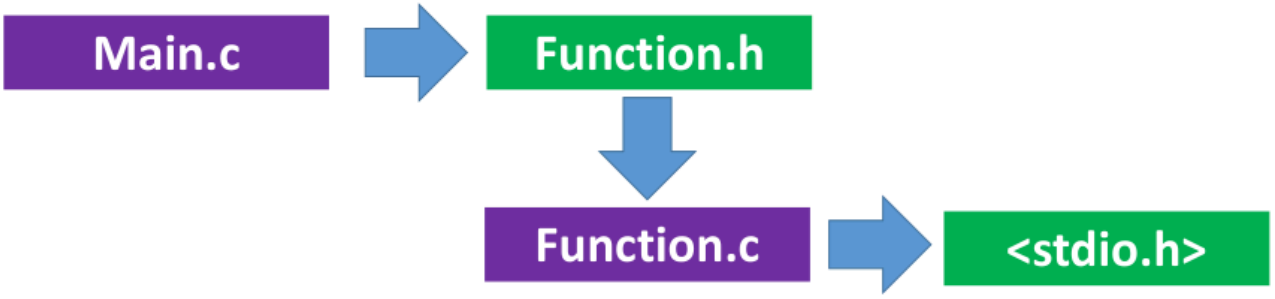
Nel file di implementazione (con lo stesso nome!) inseriamo l'implementazione dei prototipi. Se
necessario, i file di implementazione possono avere a loro volta delle direttive `#include`

```c
#include <stdio.h> // include
int f(int x) { // funzione
   static int a = 0;
   int b = 0;
   printf("Ciclo %d:%d\t%d\n", x,a,b)
}
```
function.c

Nel main includiamo il nostro nuovo header files, così come se fosse una delle librerie standard del C, e possiamo utilizzarne le funzioni.
Importante: virgolette, non parentesi angolari!

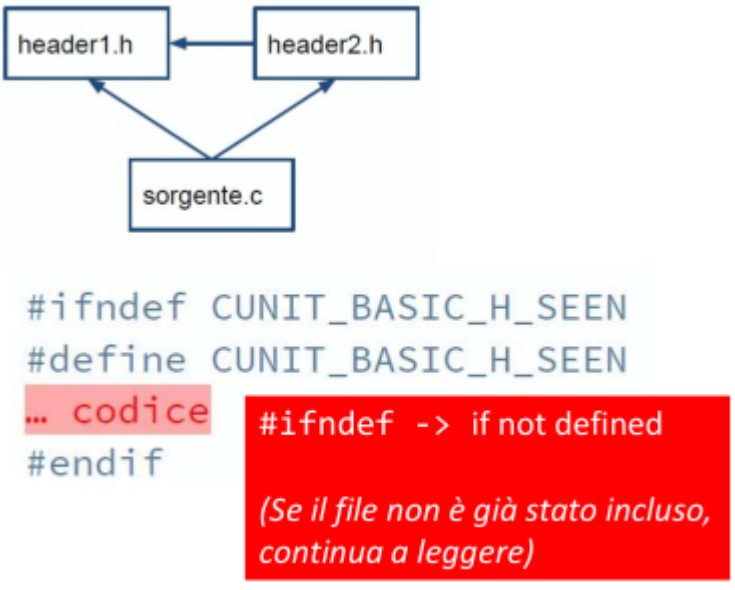
```c
#include "function.h" // virgolette dopp

int main() {  // main
  for(int i=1; i<=10; i++) {
    f(i); // invocazione
  }
}
```
main.c

L'utilizzo degli header files cambia la struttura dei programmi. Il **main** invoca le funzioni implementate in **function.h**, che a sua volta nella sua implementazione ha bisogno delle funzioni della libreria standard.
Il processo di esecuzione dei programmi può essere rappresentato attraverso un **grafo aciclico** (perché non ci possono essere dipendenze circolari tra librerie)



Parti di un Header File

1. **Prologo**: Si tratta di un «commento» che descrive a cosa serve il file e che tipo di informazioni contiene, può includere: autori, data, numero di versione, riferimenti e link esterni, informazioni su licenze e copyright.
2. **Guardia**
   - La struttura a grafo delle dipendenze può portare ad avere dipendenze multiple.
   - La guardia serve a inserire delle condizioni nella definizione delle informazioni contenute nell'header file, per evitare definizioni multiple.
   - Uso della direttiva `#ifndef` (if not defined) per controllare se il file è già stato incluso



3. **Direttive di inclusione**
   - Un modulo può implementare funzioni che necessitano di altre librerie.
   - Ad esempio, includere `<stdio.h>` per poter utilizzare la funzione `printf()`.

4. **Costanti**
    - Un header file deve includere anche le definizioni di costanti e macro
    - È possibile usare `#ifndef` per evitare ridefinizioni di costanti già definite nel programma.
5. **Tipi di Dato** Un header file deve includere anche le definizioni di nuovi tipi di dato.
    - Si definisce con il solito comando `typedef`.
    - Vantaggi:
        - Astrazione
        - Leggibilità
6. **Variabili Globali**: Le variabili definite negli header file hanno visibilità globale, visibili da tutte le funzioni o da tutti i moduli.
    - A volte può essere utile avere variabili globali accessibili da tutti.
7. **Prototipi di Funzione**: a livello minimale, un header file deve contenere almeno un prototipo di funzione.

Linker

Il processo di **compilazione e linking** in C permette di gestire un programma suddiviso in più file in modo modulare ed efficiente.

Quando scriviamo un programma con più file, ad esempio un `main.c` che utilizza funzioni definite in `function.c`, il compilatore traduce separatamente ogni file `.c` in un file oggetto `.o`. Questo significa che `main.o` sa che esiste una funzione `f()`, perché la sua dichiarazione è presente nell'header file `function.h`, ma **non contiene l'implementazione della funzione**. L'implementazione vera e propria si trova in `function.o`.

A questo punto interviene il **linker**, il cui compito è collegare questi file oggetto, risolvendo i riferimenti tra di essi. Il linker prende `main.o`, cerca l'implementazione della funzione `f()` in `function.o` e collega tutto insieme per creare l'eseguibile finale. Se il linker non trova l'implementazione di una funzione dichiarata, restituirà un errore.