Programmazione

⌨️ Programmazione

▼ **0.0 - Informazioni generali**

Ore di studio

Totale: **300**

Lezione: **132**

Casa: **168** (**3** al giorno - 3,5 giorni a settimana)

Esame

- Scritto
  - Durata: 2h 30 - 3h.
  - Punteggio: 14 (min.) - 24 (max.).
  - Struttura: 3 esercizi di programmazione su carta.
    1. Algoritmico, ricorsione o no, iterativo
    2. Liste
    3. Programmazione a oggetti
- Progetto + orale
  - Punteggio: 8 (max.).
  - Struttura: Sviluppo progetto in 3/4 studenti + discussione orale progetto con il gruppo.

Ricevimento

- Cosimo Laneve
  - Di solito il Giovedì dalle 12.00 alle 13.30: prendere un appuntamento via email.
- Giuseppe Lisanti
  - Giovedì pomeriggio dalle 14 alle 18. Da concordare insieme via e-mail.

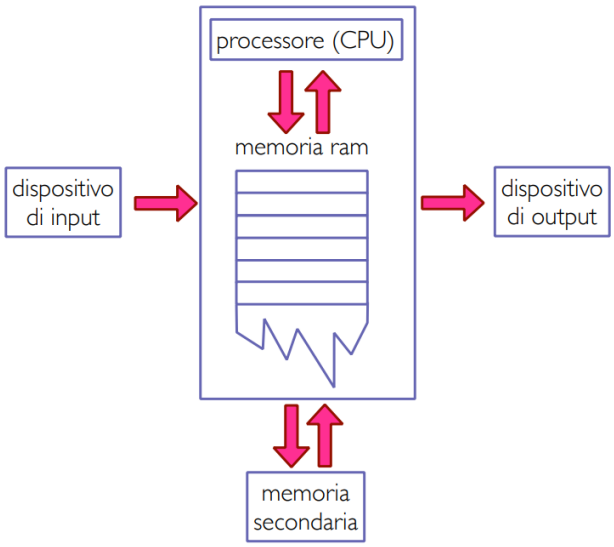
▼ **1.0 - Introduzione ai calcolatori e alla programmazione**

▼ **1.1 - Struttura di un sistema di calcolo**

Cos’è un sistema di calcolo?

Un **sistema di calcolo** rappresenta l’unione tra hardware e software, ovvero tra l’insieme dei componenti fisici che costituiscono un computer e l’insieme dei programmi (insieme di istruzioni da eseguire) utilizzati da questo.

Organizzazione di un calcolatore



Architettura di Von-Neumann

Un calcolatore è costituito da **5 componenti principali**:

1. **Dispositivi di input**
2. **Dispositivi di output**
3. **Processore**

   Anche chiamato CPU (Central Processing Unit), esegue le istruzione di un programma.

4. **Memoria principale** (RAM, Random Access Memory)

   La memoria principale è utile a memorizzare dati e programmi in esecuzione in maniera **temporanea** ed è costituita da una lunga sequenza di **locazioni**, ognuna da 8 bit (1 byte), con un identificativo.

   Per memorizzare dati più grandi di 8 bit vengono utilizzati più blocchi di locazione contigui e come identificativo quello del primo byte.

   Come capire se una successione di numeri binari rappresenta un’istruzione, un numero o una stringa?

   Una sequenza di bit possono rappresentare sia un’istruzione, che un numero, che una lettera o altri **tipi di dato**, per questo viene sempre utilizzata la prima

locazione di un dato, nella quale si trova l'istruzione, per decidere di che tipologia saranno i byte che seguono.

Qual è la differenza tra locazioni e registri?

Mentre **i registri** si trovano all'interno del processore ed hanno una latenza di accesso nulla, le **locazioni** si trovano al di fuori della CPU.

Perchè 8 bit?

Esistono due ragioni per cui 1 byte corrisponde ad 8 bit:

- 8 è una potenza di 2, dunque è molto semplice da gestire da un elaboratore, il quale lavora utilizzando un sistema binario.
- 8 bit sono abbastanza per rappresentare un singolo carattere della tastiera

5. **Memoria secondaria**

   La memoria secondaria è utile a memorizzare dati e programmi in maniera **permanente** e può essere di diverso tipo (hard disk, cd, dvd, flash drive ecc.).

▼ 1.2 - Linguaggi di programmazione e compilazione

Un sistema di calcolo è in grado di eseguire istruzioni scritte solamente in **linguaggio macchina**, il quale però non è facilmente comprensibile dall'umano. Per risolvere questo problemi sono stati creati dei linguaggi che fanno da intermediari tra la macchina e l'umano al fine aiutare la programmazione. Questi linguaggi si dividono in due categorie:

- Linguaggi a **basso livello**

  Linguaggi facilmente comprensibili dagli elaboratori. Un esempio è il linguaggio Assembly, il quale ha una forte corrispondenza tra le istruzioni che utilizza e le singole istruzioni che vengono eseguite dal computer.

- Linguaggi ad **alto livello**

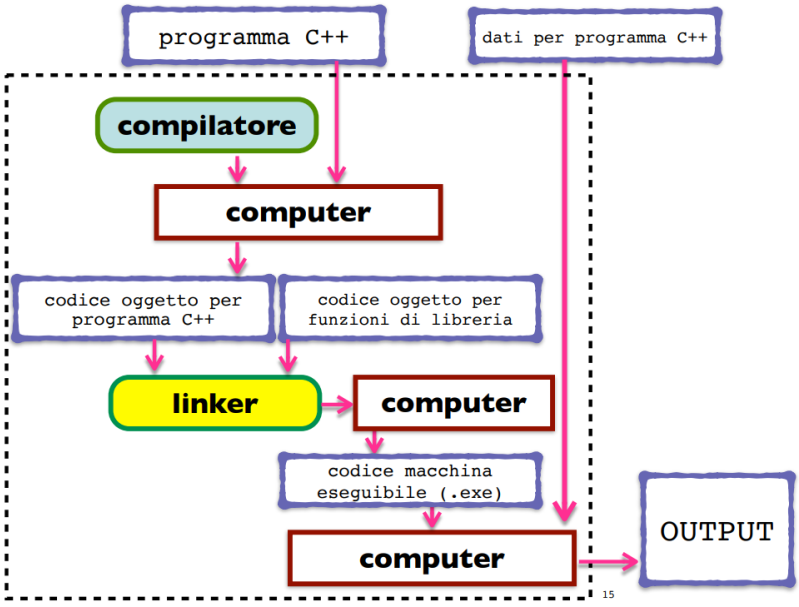
  Linguaggi facilmente comprensibili dall'essere umano in quanto somigliano ad un linguaggio naturale.

  Per essere eseguiti devono essere tradotti in codice macchina dai **compilatori**. In questo modo si vengono a creare due versioni dello stesso codice:

  - **Source code**: codice originario scritto in linguaggio ad alto livello.
  - **Object code**: codice in linguaggio macchina tradotto dal compilatore.

  Molti dei programmi, applicativi o funzioni di libreria sono già tradotti in codice oggetto per essere eseguiti più velocemente. Per questo motivo esistono i

linker, i quali combinano il codice oggetto del programma che il programmatore scrive con il codice oggetto pre-compilato delle funzioni di libreria.



Compilazione ed esecuzione di un programma

▼ 1.3 - Algoritmi e programmi

- **Algoritmo**: insieme di istruzioni per risolvere un problema.
- **Programma**: algoritmo espresso in un linguaggio di programmazione.

La programmazione è un processo composto da due fasi:

- **Problem solving**: si individua un algoritmo per risolvere un dato problema.

  L'esperienza dice che definire un algoritmo prima dell'impolementazione ci fa guadagnare tempo.

  Uno dei tanti metodi per definire un algoritmo è quello di utilizzare un diagramma di flusso.

- **Implementazione**: si traduce l'algoritmo in un programma.

  1. **Traduzione** dell'algoritmo in un linguaggio di programmazione.
  2. **Compilazione** del codice sorgente e risoluzione degli errori segnalati dal compilatore.
  3. **Esecuzione** del codice su alcuni casi di test
  4. **Risoluzione** dei bug (nome che deriva da una falena che causò un guasto in un relay del calcolatore Mark I1)

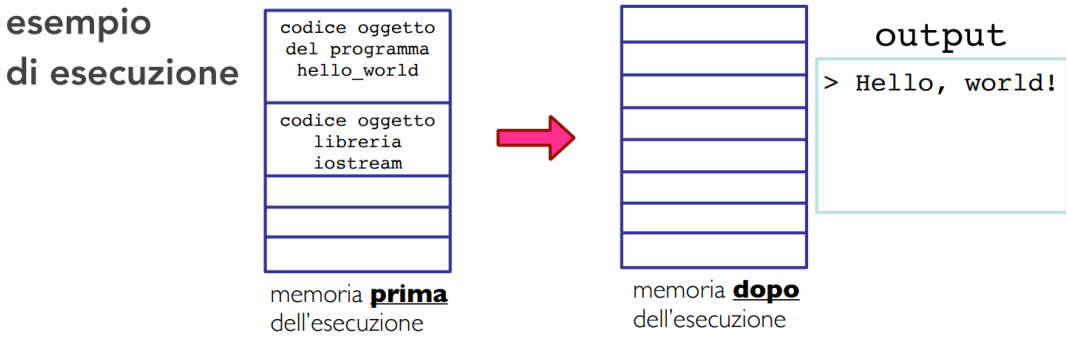
     Tipi di errori:

- Di **sintassi**: violazione delle regole grammaticali del linguaggio, vengono scoperti dal compilatore.
- A **run-time**: si verificano durante l'esecuzione (Es. out-of-memory)
- **Logici**: errori nella logica del programma.

Esecuzione di un programma

L'esecuzione di un programma è un processo che si divide in 3 passi.

1. Il compilatore traduce le istruzioni in linguaggio macchina.
2. Il linker include il codice binario delle librerie incluse.
3. L'elaboratore esegue la versione in linguaggio macchina del programma.



Esempio dell'utilizzo della memoria prima e dopo l'esecuzione di un programma.

▼ 1.4 - Funzioni di libreria

cmath

Librerie per calcolare le funzioni matematiche di uso più comune.

- double abs(double)
- double sqrt(double)
- double pow(double, double)
- double cos(double)
- double sin(double)

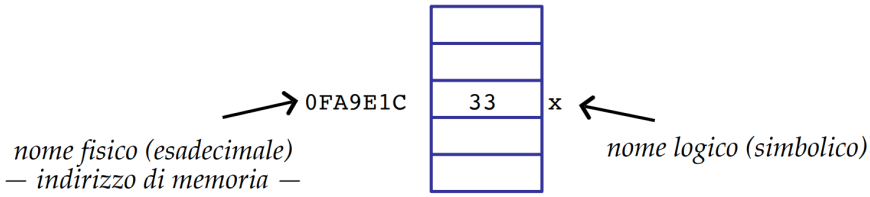
cstdlib

- int rand()
- int sran(int)
- RAND_MAX

▼ 2.0 - Identificatori e tipi di dato

▼ 2.1 - Identificatori

Gli **identificatori** sono dei nomi simbolici creati dal programmatore ed associati ad un valore. Nella pratica, un identificatore è un **nome logico** associato ad una cella della memoria. Una cella di memoria viene però anche identificata da un **nome fisico** chiamato **indirizzo di memoria**, solitamente in binario o esadecimale, il quale non viene utilizzato dal programmatore ma solo dall'elaboratore durante l'esecuzione del programma.



Nome fisico e logico.

Essi in molti linguaggi di programmazione, incluso C++, sono **case-sensitive**, ovvero lettere maiuscole e minuscole non rappresentano la stessa cosa. Non è possibile inoltre utilizzare le **parole chiave** del linguaggio di programmazione (es. int, float, double, main ecc.) per creare identificatori.

La **dichiarazione** di un identificatore è necessaria ad allocare la memoria sufficiente a contenere i valori utilizzati dal programma.

```
int number;
```

L'istruzione di **assegnamento** permette di assegnare il valore di un espressione ad un identificatore.

```
[identifier] = [expression];
```

Esistono identificatori particolari, chiamati **costanti**, i quali devono essere assegnati al momento della dichiarazione e non possono essere modificati. È possibile dichiarare una constante in due modi:

- Aggiungendo il prefisso const.

  ```
  const double pi = 3.14, e = 2.71;
  ```

- Definendola ad inizio file.

```
#define KMS_PER_MILE 1.609
```

▼ 2.2 - Tipi di dato

I valori utilizzati dal programma vengono distinti in base al **tipo di dato**. Questo permette di ottimizzare l'**uso della memoria**, allocando la giusta memoria al giusto dato.

Esempio di tipo di dato e memoria allocata:

- **int**: 4 byte
- **double**: 8 byte
- **char**: 1 byte

I tipi di dato servono anche ad ottimizzare le **prestazioni del processore**, il quale suddivide le sue operazioni in base al tipo.

Alcune operazioni, inoltre, possono essere effettuate solo tra certi tipi di dato, come l'operazione **resto** (%), la quale può essere effettuata solo tra interi.

Cast

È possibile anche effettuare conversioni, dette **cast**, tra tipi di dato compatibili, come tra i double e gli interi.

```
(int) 4.6 // -> 4
(double) 4 // -> 4.0
```

I cast possono anche essere effettuati implicitamente, ad esempio in ogni operazione che utilizza dei float può essere inserito come parametro un intero, il quale viene convertito implicitamente a float:

```
4 / 2.3 -> 4.0 / 2.3
log(4) -> log(4.0)
```

Type safety

I tipi di dato all'interno di un linguaggio di programmazione vengono usati non solo per ottimizzare l'utilizzo della memoria, ma anche per la **type safety**, una proprietà che garantisce che i dati utilizzati dal programmatore all'interno di un programma sono giusti.

In questo modo si riescono ad evitare diversi errori in quanto il compilatore rileva in anticipo errori di tipo senza eseguire le eventuali operazioni, le quali sono

sbagliate (es. somma tra intero e stringa) ma sono comunque fattibili, visto che tutte le informazioni vengono memorizzate in binario.

▼ 2.3 - Espressioni e input/output

Espressioni

Un'**espressione** è una sequenza di operazioni che restituiscono un valore e può essere composta da:

- Una **variabile/costante**.
- Una **chiamata di funzione**.
- Una **combinazione** di variabili, costanti e chiamate di funzione connesse da operatori.

L'ordine di valutazione delle espressioni è fissato da:

1. Parentesi.
2. Precedenza tra operatori.

- Operatori aventi la stessa precedenza vengono valutati da sinistra verso destra se sono binari, da destra verso sinistra se unari.

Input/output

Uno dei tanti metodi per fare **input/output** in C++ consiste nell'utilizzo degli stream **cin** e **cout** (ricordarsi di includere <iostream>), con i quali si interagisce tramite gli operatori **<<** e **>>**.

```
cout << [expression1] << [expression2]; // output
cin >> [identifier1] >> [identifier2]; // input
```

È importante ricordarsi che l'operatore << associa a sinistra e il suo livello di precedenza è inferiore rispetto a quello degli operatori aritmentici, ma non di quelli logici.

L'operatore >> invece si comporta in maniera diversa a seconda del tipo di dato da inserire nell'identificatore. Ad esempio se l'identificatore è di tipo char, cin legge il primo carattere che incontra, mentre se è un intero legge finchè trova caratteri numerici validi. Eventuali spazi o a capo non vengono considerati.

Sequenze di escape

Le **sequenze di escape** consentono di inserire caratteri speciali all'interno di stringhe, esse sono composte da un **backslash** (\) seguito da un **codice speciale**. Le sequenze di escape più importanti sono:

- **\n**: nuova linea.
- **\t**: tab.
- ****: backslash.

La keyword **endl** consente di inserire la sequenza di escape \n nello stream.

▼ 3.0 - Comandi condizionali e iterativi

▼ 3.1 - Comandi condizionali

I **comandi condizionali** consentono di effettuare una scelta, in base ad una condizione, tra diversi comandi alternativi da eseguire.

In C++ è possibile scrivere un comando condizionale in questo modo:

```
if ([cond1]) {
  // code1
} else if ([cond2]) {
  // code2
} else {
  // code3
}
```

La parte else if ed else del comando condizionale è facoltativa.

▼ 3.2 - Comandi iterativi

I **comandi iterativi** sono comandi che consentono di eseguire uno stesso blocco di istruzioni più volte.

Queste tipologie di comandi sono composti da una **guardia del ciclo**, ovvero la condizione che controlla il numero di iterazioni, e dal **corpo del ciclo**, ovvero l'insieme delle operazioni che vengono ripetute.

I comandi iterativi più importanti sono 2, uno è il while mentre l'altro è il for. Il secondo viene utilizzato quando si conosce preventivamente il numero di iterazioni da eseguire, mentre il primo viene utilizzato in tutti gli altri casi. La sintassi in C++ è la seguente:

```
while ([condition]) {
  // code
}
```

```
for (int i = [num]; [condition]; i++/--) {
  // code
}
```

All'interno del corpo del ciclo while ci deve sempre essere almeno una istruzione che invalida la guarda del ciclo.

Esiste un altro tipo di comando iterativo, ovvero il **do while**, il quale consente di effettuare un ciclo while almeno una volta:

```
do {
  // code
} while ([condition]);
```

Guardie dei cicli

Le guardie dei cicli devono sempre essere prima o poi invalidate al fine di non creare cicli infiniti. I due metodi utilizzati per invalidare le guardie dei cicli sono 2:

- Contatori

  Un numero che varia ad ogni esecuzione del ciclo e che consente di calcolare in anticipo il numero di iterazioni del ciclo in base alla condizione e all'incremento/decremento inserito.

- Flag

  Variabili il cui cambiamento di valore indica che un particolare evento è avvenuto.

Esempio:

- Combinazione contatore e flag:

  ```
  bool is_prime(int n) {
      bool prime = true;
      int i = 2;
      while (i <= (n / 2) && prime) {
  ```

```
            if (n % i == 0) prime = false;
            else i++;
        }

        return prime;
    }
```

Errori nei cicli

Gli errori più comuni che riguardano i cicli sono:

- **Errori off-by-one**: il ciclo esegue una volta in più o in meno di quanto dovrebbe.

  Per risolvere errori di questo tipo occorre verificare la guardia del ciclo (es. < o <=), verificare l'inizializzazione della variabile di controllo e verificare se il ciclo prende in considerazione il caso di 0 iterazioni.

- **Cicli infiniti**: il ciclo esegue all'infinito, a causa di errori della guardia.

  Per risolvere errori causati da cicli infiniti occorre controllare la guardia del ciclo (es. < o >), oppure utilizzare < o > nel caso in cui si utilizzi == nella guardia (ad esempio bisogna infatti ricordarsi che i double sono approssimati).

Ogni volta che un programma viene modificato occorre **ritestarlo**, in quanto il cambiamento di una parte del codice può causare il malfunzionamento di un'altra parte. Un ciclo deve essere testato con diversi input che consentono di verificare i seguenti casi:

- **Zero** iterazioni del corpo.
- **Una** iterazione del corpo.
- Un numero di iterazioni del corpo **immediatamente inferiore al massimo**.
- Il **numero massimo** di iterazioni.

▼ 4.0 - Funzioni, portata di una dichiarazione e librerie

Portata di una dichiarazione

La **portata di una dichiarazione** di un identificatore indica la parte di programma in cui essa è valida.

La portata di una dichiarazione segue le seguenti regole:

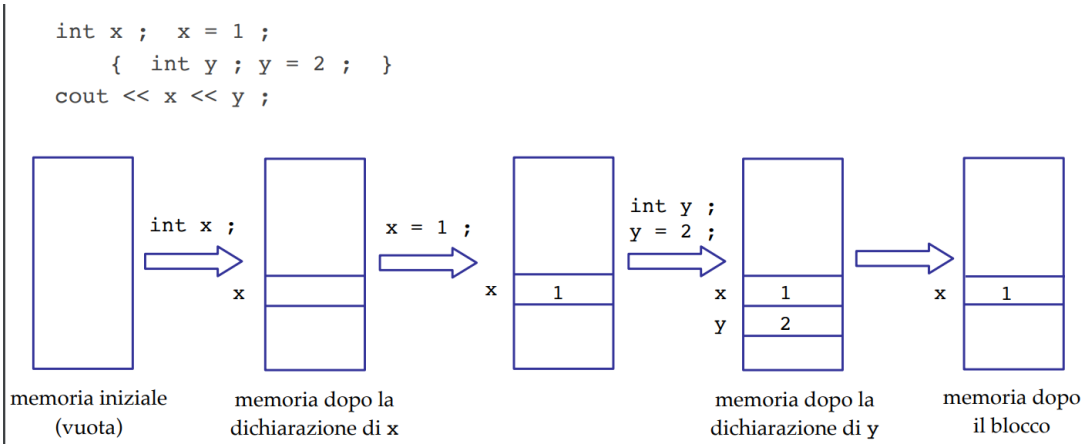
- Questa si limita al **blocco**, delimitato da parentesi graffe **{}**, in cui occorre la dichiarazione.
- Un **blocco interno** può utilizzare identificatori definiti nei blocchi che lo contiene, nonostante ciò il blocco interno e i suoi contenitori non fanno parte della stessa

portata, dunque è possibile ridefinire identificatori nel blocco interno senza modificare quelli definiti nel blocco esterno.

L'identificatore di una variabile fa riferimento alla variabile definita nel blocco più vicino a lui.

- Non è possibile **ridefinire lo stesso identificatore** all'interno del medesimo blocco.

Dopo che un blocco viene chiuso, in memoria vengono eliminati i dati inizializzati all'interno di quel blocco.



Esempio di utilizzo della memoria e portata di una dichiarazione.

Funzioni

Una **funzione** rappresenta un blocco di codice al quale viene assegnato un nome. Essa può avere degli argomenti e restituire dei valori.

Benefici delle funzioni

L'utilizzo di funzioni all'interno di un programma consente di **riutilizzare codice già esistente**. Questo porta dei benefit sia dal punto di vista della leggibilità del programma che nella modifica del programma, in quanto è sufficiente farlo una sola volta per tutte, evitando il meccanismo del cut & paste.

Definizione, dichiarazione e chiamata di funzione

```
[returnType] [functionName] ([type1] [par1], [type2] [par2]) {
  [code];
}
```

Una funzione non può essere definita all'interno di un'altra funzione, e la **portata della sua definizione** è dal momento in cui si trova in tutto il resto del programma.

Nonostante ciò è possibile utilizzare una funzione prima della sua definizione utilizzando una **dichiarazione**, la quale comprende solamente l'intestazione della funzione e informa il compilatore del nome della funzione.

```
[functionName]([arg1], [arg2]);
```

Parametri e argomenti

Quando viene definita una funzione occorre elencare i **parametri formali** di questa, ovvero identificatori associati a un tipo ai quali devono essere assegnati dei valori al momento della chiamata di funzione.

I reali valori che vengono assegnati ai parametri al momento della chiamata vengono invece chiamati **argomenti** o **parametri attuali** della funzione. **Il numero e il tipo** di parametri attuali che vengono passati ad una funzione devono essere uguali a quelli dei parametri formali, ad accezione del fatto che a volte il compilatore può effettuare conversioni tra tipi di dato simili (es. int → real, real → int, chat → int).

I parametri formali di una funzione sono visibili solo all'interno di questa.

C++ offre 2 modalità di passaggio dei parametri:

- **Valore**

  Il valore dei parametri attuali viene memorizzato in variabili locali alla funzione, e ogni modifica all'interno del corpo della funzione riguarderà esclusivamente le variabili locali.

- **Riferimento**

  Viene passato l'indirizzo della cella di memoria che contiene il valore passato come parametro attuale, dunque ogni modifica all'interno del corpo della funzione riferita ai parametri riguarderà le variabili passate come parametri, e non delle variabili locali ad essa. In questo caso non è possibile passare un'espressione come parametro attuale.

  Per fare ciò occorre aggiungere un '&' affianco al tipo del parametro formale nella dichiarazione della funzione:

  ```
  [returnType] [functionName] ([type]& [par]) {
    [code];
  }
  ```

  Vantaggio: consente di ottimizzare l'uso della memoria in quanto non crea variabili locali alla funzione dove inserire i valori passati come argomento.

  Svantaggio: rende i programmi meno comprensibili.

Quando una funzione termina la sua esecuzione i **blocchi di memoria** utilizzati per i parametri e gli identificatori inizializzati all'interno di essa vengono rilasciati.

Ritorno di valori

Per far **ritornare** alla funzione un valore al chiamante occorre specificare un tipo **non-void** nella sua intestazione ed inserire nel suo corpo l'istruzione **return** [expression]. Il risultato dell'[expression] indicata a fianco di return deve essere dello stesso tipo di quello specificato nell'instestazine di funzione, ad accezione del fatto che a volte il compilatore può effettuare conversioni tra tipi di dato simili (es. int → real, real → int, chat → int).

Quando viene eseguita l'itruzione return all'interno di una funzione l'esecuzione della funzione si conclude ed il controllo ritorna al chiamante, dunque i comandi dopo il return non vengono mai eseguiti.

Una funzione con tipo **void** può avere l'istruzione return al suo interno, senza però possibilità di inserire alcuna espressione da ritornare. Nonostante ciò è una pratica sconsigliata.

Al contrario, se una funzione è stata definita con tipo non-void ma non presenta l'espressione return allora sta al compilatore decidere se fornire un errore o fornire un risultato scelto da lui. Questo tipo di funzioni sono in ogni caso considerate erronee.

Variabili globali nelle funzioni

Variabili definite fuori dalle funzioni vengono dette **variabili globali**. All'interno di una funzione è anche possibile modificare il valore di queste variabili, nonostante ciò è una pratica sconsigliata in quanto complica la comprensione del programma e ne può rendere ambiguo il significato.

Nonostante ciò, è possibile accedere in lettura e in scrittura alle variabili globali da una funzione rendendo il codice più leggibile nei seguenti modi:

- Accesso in **lettura**: aumentare il numero di parametri formali della funzione e, ogni volta che la si chiama, passare la variabile globale come argomento.
- Accesso in **scrittura**: aumentare il numero di parametri formali della funzione con un parametro passato per reference e, ogni volta che la si chiama, si passa la variabile globale come argomento. Questa pratica rimane comunque da evitare se possibile.

Librerie

Procedural abstraction e information hiding

L'utilizzo di funzioni all'interno del codice permette di programmare seguendo la pratica della **procedural abstraction**, la quale prevede che il programmatore che usa una funzione deve soltanto conoscere il suo scopo e come invocarla, mentre non gli

deve interessare, una volta che la funzione è stata costruita nella maniera corretta, il suo funzionamento.

L'**infomation hiding** prevede l'utilizzo della pratica procedural abstraction e consiste nell'implementare le funzioni utilizzate nel programma in un altro file. In questo modo il codice diventa più leggibile e inoltre è possibile modificare le funzioni senza che gli utilizzatori ne siano a conoscenza.

In questo modo per utilizzare una funzione il programmatore deve solamente leggere i relativi commenti che ne spiegano il funzionamento, il quali si strutturano solitamente in **precondizioni**, che ne stabiliscono le condizioni per gli argomenti della funzione, e **postcondizioni**, che ne descrivono il valore ritornato.

```
int foo(int x, double& z);
// Precondition: x > 0
// Postcondition: z memorizza la metà di x
```

Creazione e utilizzo di librerie

Per creare una libreria di funzioni occorre creare almeno due nuovi file, uno con estensione **.h** e l'altro **.cpp**. Nel primo bisogna inserire solo le dichiarazioni delle funzioni accompagnate da commenti che ne chiariscono l'utilizzo, mentre nel secondo se ne inserisce implementazione. I file .cpp che definiscono le funzioni dichiarate nel file .h possono essere più di uno, l'importante è includere in ognuno di questi il file .h.

```
// file library.h
int min(int x, int y);
```

```
// file library.cpp
#include "library.h"

int min(int x, int y) {
  if (x < y) return x;
  else return y;
}
```

Namespace

I namespace permettono di includere più di una funzione avente lo stesso nome e di riferirsi ad una di queste in maniera chiara.

È possibile creare un namespace inserendo uno scope con un identificativo all'interno dei due file .**h** e **.cpp** che definiscono la libreria, per poi utilizzarli nella chiamata a funzione utilizzando l'identificativo utilizzato e **::**.

```
// file library.h
namespace one {
  int min(int x, int y);
}
```

```
// file library.cpp
#include "library.h"

namespace one {
  int min(int x, int y) {
    if (x < y) return x;
    else return y;
  }
}
```

```
// file main.cpp

int main() {
  int a = 2, b = 1;
  one::min(a, b);

  return 0;
}
```

Al fine di non dover scrivere sempre il nome dell'identificatore seguito dai :: è possibile utilizzare l'espressione **using namespace** [namespace] per dichiarare quale identificatore utilizzare di default da quel punto del programma in poi.

```
//file main.cpp
#include "library.h"
using namespace one;

int main() {
  int a = 2, b = 1;
  min(a, b);

  return 0;
}
```

▼ 5.0 - Array

▼ 7.0 - Stringhe

In c++ esistono due metodi per utilizzare le **stringhe**, una prevede l'utilizzo di array mentre l'altra l'utilizzo della classe string, la quale, nonostante sia più potente della

prima, non verrà utilizzata in questo corso in quanto richiede la conoscenza delle classi, che verranno trattate verso il termine.

Dichiarazione

```
char stringName[length];
```

Una stringa dichiarata come array ha un numero di caratteri uguale a **length - 1**, poichè l'ultimo carattere dell'array è il **carattere null '\0'** che indica la fine della stringa.

Ogni carattere dopo il null non è significativo, dunque la lunghezza della stringa varia in base alla posizione del '\0'.

```
char stringName[length] = "[string]";
```

Con questo tipo di assegnamento il carattere null '\0' viene aggiunto in automatico. Se la stringa data in input è più lunga dell'array essa viene troncata o viene restituito un erroe a seconda del compilatore.

Non è possibile utilizzare questo tipo di assegnamento dopo la dichiarazione di una stringa, il metodo standard per cambiarne il valore è tramite l'utilizzo della funzione strncpy.

```
char stringName[] = "[string]";
```

É importante fare attenzione a non uscire dalla dimensione dell'array con l'input, in quanto si rischia il **buffer overflow**.

Input e output

É possibile fare sia input che output di stringhe tramite le funzioni di libreria **cin** e **cout.**

La funzione cin legge fino al primo spazio bianco, dunque occorre utilizzare la seguente funzione per leggere anche gli spazi bianchi.

```
cin.getline([stringName], [length])
```

Funzioni di libreria

```
sizeof([stringname]);
```

Se la stringa è implementata tramite un array di caratteri ritorna la lunghezza dell’array, non della stringa in esso contenuta. Nota: sizeof non riesce a fornire la lunghezza di una stringa passata come parametro di funzione.

▼ **cstring**

```
strlen([stringName]);
```

```
strcat([string1], [string2]);
```

```
strncat([string1], [string2], [num]);
```

```
strcpy([string1], [string2];
```

```
strncpy([string1], [string2], [num]);
```

Se [num] è minore della lunghezza della seconda stringa aggiunge il carattere null finale, altrimenti non lo aggiunge e in questi casi occorre inserirlo manualmente. É più sicuro di strcpy perchè permette di controllare il numero massimo di caratteri da copiare nella prima stringa.

```
strcmp([string1], [string2]);
```

Ritorna 0 se le due stringhe sono uguali, altrimenti al primo carattere differente ritorna un numero positivo se il codice del carattere del primo parametro è maggiore, e ritorna un numero negativo se il codice del carattere del primo parametro è minore.

▼ **cstdlib**

```
atoi([stringName]);
```

```
atol([stringName]);
```

```
atof([stringName])
```

▼ 8**.0 - Strutture**

Una **struttura**, anche detta **record**, è un dato composto da un insieme di elementi eterogenei, i quali vengono raggruppati sotto un unico nome.

Dichiarazione

```
struct [structName] {
  [type1] [itemName1];
  [type2] [itemName2];
  [type3] [itemName3];
};
```

Nella dichiarazione di una struttura occorre definire il suo nome, le tipologie e i nomi degli elementi, detti **campi**, che contiene, al fine di quantificare lo spazio in memoria necessario per le variabili di quel tipo.

Inizializzazione

```
[structName] [varName] = {[val], [], []};
```

Operazioni sulle strutture

Accesso agli elementi

```
[varName].[itemName];
```

Per **accedere ai campi** di una struttura si utilizza la **dot-notation**.

Copia di una struttura

```
[varName1] = [varName2];
```

Tramite l’operazione di **copia** è possibile copiare i campi di una struttura nei campi di un’altra struttura.

Strutture nelle funzioni

Le strutture possono sia essere passate come **argomenti** di funzioni che essere **restituite** come valori.

Il passaggio di una struttura come parametro di una funzione avviene **per valore**, ma è comunque possibile passarla per riferimento utilizzando la parola chiave **&**.

Tipologie di strutture

Stack/pila

La struttura **stack/pila** è una struttura dati di tipo LIFO (Last In First Out), ovvero nella quale è solo possibile inserire un elemento in ultima posizione o togliere l'elemento in ultima posizione.

Queue/coda

La struttura queue/coda è una struttura dati di tipo FIFO (First In First Out), ovvero una sequenza di elementi con una testa e una coda, nella quale è solo possibile inserire un elemento nella coda o eliminare l'elemento nella testa.

▼ **13.0 - Progetto**

▼ **13.1 - Informazioni generali**

Consegna

Il progetto deve essere consegnato almeno **una settimana** prima della discussione.

Le consegna deve essere fatta tramite mail a giuseppe.lisanti@unibo.it; adele.veschetti2@unibo.it e bianca.raimondi3@unibo.it e deve contenere:

- **Codici** (sorgente e binari).
- File **README**.
- **Screen recording** che mostri l'esecuzione del gioco.
- Breve **relazione** di 3/4 pagine in cui si descrivono le principali scelte nell'implementazione del progetto.

Domande

- Cosa significa traguardi, nel livello o nel gioco generale?
- Cosa significa platform game, grafica dal lato o anche da sopra?
- È possibile tornare indietro solo al livello precedente?
- Il game over nel livello corrente o al primo livello

Idee

- Doppio salto in aria

- Piattaforme instabili
- Piattaforme che si muovono o che scompaiono e riappaiono
- Pistoni per tirare giù piattaforme con carrucola
- Scale
- Armi da vicino (es. martello) e da lontano (es. pistola)
- Monete da raccogliere nel percorso
- Mercato dentro alcune mappe (premi un pulsante per entrare)
- Personaggio scende dall'alto per iniziare il livello
- Portale alla fine e all'inizio per livello precedente e successivo
- Mercato in luoghi strategici per andare avanti
- Uccisione nemici: salto e sparo

1. Pac-man
2. Stile pac-man ma con obiettivo moneta nel centro
    - Il personaggio può sparare in orizzontale o in verticale, dunque si può girare nelle 4 direzioni per sparare.
    - Aumentano le monete sia uccidendo che raggiungendo l’obiettivo nel centro
3. Nemici dai lati, bisogna resistere un tot di secondi
4. Super mario

▼ 13.2 - Ncurses

```
#include <ncurses.h> or #include <ncursesw/ncurses.h>

// [win] = stdscr (standard screen/terminal)

// General
curs_set(0); // sets the cursor invisible
wtimeout([win], 0); // imposta [num] millicondi entro i quali l'input deve essere dato
noecho(); // don't print when input is given

// Window
initscr();
endwin();
WINDOW *win = newwin([height], [width], [start_y], [start_x]);
box([win], [left_right_char], [top_bottom_char]);
wborder([win], [left], [right], [top], [bottom], [tlc], [trc], [blc], [brc]);
refresh();
wrefresh([win]);
clean();
cbreak();
```

```
// Input (every time we ask for an input the all screen get refreshed)
getch();
wgetch([win]);
keypad([win], [bool]);

/*
  Keys:
  - KEY_UP
  - KEY_DOWN
  - KEY_LEFT
  - KEY_RIGHT
  - KEY_F([num])
  - 10 = enter_key
*/

// Coordinates
move([y], [x]);
wmove([win], [y], [x]);
getyx([win], [y_var], [x_var]); // get position of cursor of the window
getbegyx([win], [y_var], [x_var]); // get beginning y and x position of the window
getmaxyx([win], [y_var], [x_var]); // get length and height of the window

// Output
printw([format_string], [var1], [var2], ...);
mvprintw([y], [x], [format_string], [var1], [var2], ...);
wprintw([win], [format_string], [var1], [var2], ...);
mvwprintw([win], [y], [x], [format_string], [var1], [var2], ...);
addch([char]);
waddch([win], [char]);
mvwaddch([win], [y], [x], [char]);

// Attributes and colors
attron([attr1] | [attr2]);
wattron([win], [attr1] | [attr2]);
attroff([attr1] | [attr2]);
wattron([win], [attr1] | [attr2]);
has_colors();
init_pair([num], [fore_color], [back_color]);
/*
  Attributes:
  - A_NORMAL
  - A_REVERSE
  - A_INVIS
  - A_BOLD
*/
/*
  Colors:
  - COLOR_PAIR([num])
  - COLOR_BLACK
  - COLOR_WHITE
  - COLOR_RED
  - COLOR_BLUE
  - COLOR_GREEN
  - COLOR_YELLOW
  - COLOR_MAGENTA
  - COLOR_CYAN
*/
```

- Schermata iniziale

- Controllare tutto il codice
- Scegliere il font del nome
- Cambiare struttura blocchi mappa
- Cambiare animazione nel display_map()
- Metodo che passa al prossimo livello (fa il display di un livello casuale/market, aumenta il numero di nemici, aumenta la loro velocità)
- Metodo che torna al livello precedente (prima fare il salvataggio)
- Metodo che toglie una moneta
- Game over/morte del protagonista
- Uscita dal gioco con un tasto (sei sicuro di voler uscire?)
    - Resume game
- Controllare i // TODO

☐ Prova a usare una struttura con un solo array e ritornarlo da una funzione

In un assegnazione ciò che si trova a sinistra va inteso come indirizzo, mentre ciò che si trova a sinistra va inteso come contenuto. (x = y → all'indirizzo x inserire il contenuto di y)

Comandi condizionali e iterativi

Evitare sempre di interrompere una iterazione con un return, in quanto la leggibilità del programma aumenta quando l'unica condizione di terminazione di una iterazione è la guardia del comando iterativo.

Array

Gli elementi dell'array vengono memorizzati in celle di memoria contigue.

```
int arrayName[20];
```

La lunghezza dell'array deve essere una costante (su c++ è possibile creare anche array dinamici ma è meglio non farlo).

```
arrayName[index]
```

Gli out of bound negli array non vengono segnalati come errore in c++. è possibile inserire qualunque numero al posto dell'indice dell'array, verranno lette le celle successive ad esso. Non farlo.

```
int [functionName](int arrayName[])
```

L’array nelle funzioni vengono passati per riferimento, nella pratica viene passato l’indirizzo di memoria del primo elemento dell’array, tramite il quale poi sarà possibile accedere alle seguenti locazioni di memoria dell’array. Le funzioni non conoscono la lunghezza dell’array, a volte è utile passare anch’essa come parametro.

Algoritmi di ordinamento di array

Ordinare prima un array consente di effettuare molte operazioni su di esso in maniera più ottimizzata.

- Selection sort

  Viene ciclato tutto l’array ed inserito nella posizione i il numero minimo compreso tra i e length - 1.

- Bubble sort

  Viene ciclato tutto l’array più volte ogni volta scambiato il numero in posizione i con quello in posizione i + 1 se non ordinati.

Ricercare un elemento all’interno di array ordinato perette di effettuare molte meno operazioni rispetto a un array non ordinato. La computazione cambia da l a $\log_2 l$.

Testa: primo elemento

Coda: ultimo elemento

Strutture dati

Una **struttura** è un dato, anche chiamato **record**, composto da elementi che possono essere eterogenei e che vengono raggruppati sotto un unico nome.

Implementazione C++

```
struct structId {
  type fieldid1;
  type fieldId2;
};
```

Per accedere ai campi di una struttura: variabile.campo.

Le strutture possono essere copiate in altre variabili, passate come parametri (sia come valori che per riferimento) e ritornate come valore nelle funzioni.

Pile

Una **sequenza** su un insieme A è una successione di elementi di A.

- Sequenza vuota: ϵ
- Lunghezza della sequenza: $|\sigma|$ = numero di elementi di σ
- Concatenazione: $\sigma * \sigma' = e_1 * ... * e_n * e'_1 * ... * e'_n$

Pile, code e liste sono particolari sequenze.

Una sequenza è un prodotto cartesiano: $e_1 * ... * e_n = (((e_1, e_2), e_3), e_4)$

Una **pila** è una sequenza avente le seguenti operazioni:

- **isEmpty(σ)**
  - true if $\sigma = \epsilon$, false altrimenti
  - Pila → bool
- **push(σ, e)**
  - e * σ
  - Pila x A → Pila
- **pop(σ)**
  - σ' if $\sigma = \sigma' e$, ϵ altrimenti
  - Pila → Pila

L'implementazione di una struttura dati non è sempre equivalente dalla definizione algebrica di questa. Ad esempio per implementare le operazioni della pila utilizzando un array su c++ occorre passare come argomento anche la lunghezza dell'array.

Code

Una coda è una sequenza di elementi con una testa e una coda.

- **isEmpty(Q):**
- **enqueue(Q, e):** aggiunge un elemento al termine della coda
- **deqeue(Q)**: elimina un elemento all'inizio della coda.

Insieme

- Creazione insieme vuoto
- Appartenenza di un elemento nell'insieme
- Intersezione
- Unione (fare attenzione se l'insieme unione è più grande della lunghezza dell'array)

Puntatori e strutture dati dinamiche

I puntatori non sono essenziali per computare qualsiasi programma, ma consentono una migliore gestione di questo.

Il programma non può conoscere a priori gli indirizzi di memoria che utilizzerà durante l'esecuzione perchè questi cambiano in base a dove viene caricato il programma.

Un puntatore è una variabile che ha in memoria l'indirizzo di un altra variabile.

Spesso la quantità di memoria utilizzata da un programma non è stimabile prima del suo avvio e in casi estremi potrebbe esaurire la memoria utilizzabile (out of memory).

Le operazioni principali tra puntatori sono == e !=.

Un certo puntatore non può contenere gli indirizzi di qualunque dato, ma solo di quelli dello stesso tipo (int, char ecc.).

```
[type]* [nomePuntatore]
```

La costante NULL rappresenta il puntatore vuoto. Ha valore 0 ed è definita in diverse librerie tra cui iostream. NULL può essere assegnato solo ai puntatori.

```
new [type]
```

Memoria programma divisa in stack (vengono allocati i record delle funzioni) e heap (vengono allocati nuovi blocchi di memoria - new). La stack si comporta come una pila, mentre la heap si comporta in base al sistema operativo.

Le variabili create con new non hanno un nome.

```
*[nomePuntatore] = [value]
```

```
&[lhs-expression]
```

```
delete [nomePuntatore];
// Non lasciare del dangling pointer, riassegna sempre a NULL un puntatore
// dopo l'invocazione di delete per evitare errori.
nomePuntatore = NULL;
```

```
[type]** [nomePuntatori];
```

```
typedef [type] [typeName];
```

```
typedef int* p_int;
p_int p, q;
```

Non è possibile ritornare una variabile locale in una funzione.

```
struct [structName] {
  [structName]* [pointerName];
};

q = new [structName];
(*q).punt = p;
p = q;
// se si ripetono queste ultime 3 righe si crea una struttura dinamica di puntatori
```

Liste

Una lista è una sequenza di nodi che contengono valori di un determinato tipo e nella quale ogni nodo è connesso al successivo tramite un puntatore. Una lista può crescere e decrescere durante l'esecuzione. L'ultimo elemento punta a null.

Solitamente viene utilizzata una struttura con due campi: un puntatore e un valore.

Viene utilizzare un puntatore che punta al primo elemento della lista.

```
struct lista {
  int val;
  p_lista next;
}

typedef lista* p_lista;

p_lista head;
head = new lista; // creare il primo elemento della lista
(*head).val = 1; // assegnare un valore al primo elemento della lista
head -> val = 1; // fa la stessa cosa della riga prima

head -> next = new lista; // creare il secondo elemento della lista
(head -> next) -> val = 2; // assegnare un valore al secondo elemento della lista
```

```
p_lista tmp = new lista;
tmp -> next = p1 -> next;
p1 -> next = tmp;
```

aggiungere un elemento alla head

Se viene passata una lista come parametro ad una funzione viene passata come valore, dunque tutto ciò che viene fatto nella funzione non modofica la lista originale.

Per passare una lista per riferimento occorre aggiungere &.

Non bisogna mai perdere il primo elemento della lista, quindi quando occorre iterarla è meglio farlo su una variabile ausiliaria, lasciando il puntatore al primo elemento.

Le operazioni sulla testa della losta sono facili, mentre quelle sulla coda no poichè la lista deve essere completamente iterata.

È possibile perdere dei nodi e dei nodi persi non sono più recuperabili, dunque stare attenti alle operazioni sulle liste.

Inserimento e rimozione di elementi dalla coda.

```
char stringName[length] = "[string]";
```

Funzioni ricorsive

Solitamente le funzioni ricorsive prendono un numero maggiore di parametri rispetto a quelle iterative.

Strutture

Altezza albero: n

Numero nodi: 2^n - 1

La forma infissa è algebricamente ambigua (servono le parentesi), mentre quelle prefissa e postfissa no.

Alberi

```
struct [tree] {
  int val;
  tree *ltree;
  tree *rtree;
}
```

La lunghezza di un albero è data dal percorso più lungo dalla radice a un nodo - 1 (un albero con solo la radice è lungo 0).

OOP

- **Encapsulation**

  Ogni oggetto contiene i propri dati e le funzioni che possono accedere/modificare i dati. Nessun altra funzione può farlo (**information hiding**).

- **Inheritance**

  Possibilità di utilizzare il codice di un oggetto per crearne un altro con la possibilità di modificare e aggiungere nuove funzioni.

- **Polimorfismo**

  Uno stesso nome di funzione può avere significati diversi in base al contesto in cui è definito.

Overloading (definire un metodo con lo stesso nome di un altro ma non con lo stesso tipo di ritorno o gli stessi parametri) e overriding (definire un metodo con la stessa signature (nome, tipo restituito e parametri) del metodo padre)