PROGRAMMAZIONE LAB

https://www.antoninofurnari.it/proglab1/2122/

❌ Appunti prog1 lab

Credenziali:
Username: `prog1`
Password: `$prog_lab$`

Ciclo for basato su range
GOTO

SIGNIFICATO SIMBOLI FONDAMENTALI C++

```
#include <iostream>

using namespace std; //commento a caso

int main () {
      cout << "Hello world" << endl;
}
```

- #è una direttiva
- *using namespace std;* si utilizza il 'namespace' STandarD che contiene gli oggetti cout, endl, cin ecc...;
- *main* è obbligatoria se vogliamo che il programma faccia qualcosa quando lo eseguiamo. Le parentesi graffe di int main () { } sono viste come un contenitore delle istruzioni che si vogliono eseguire;
- *cout* è un oggetto che stampa a schermo un numero, un'operazione o delle stringhe;
- << è un **operatore** binario e gli operandi sono cout e 'hello world'
  Prende quelll che c'è a destra e lo spinge << in cout;
- *endl* indica un ' a capo' o la fine della riga;
- ***Spazi e righe vuote*** sono ignorate dal compilatore. Si usano per la leggibilità del codice e si parla di **identizzazione.*;***
- // sono i commenti su una riga;
- */* sono i commenti*

  fra le

  righe

  /

VARIABILI E TIPI

- Ogni variabile deve avere un tipo.
- I nomi di variabili devono iniziare con una **lettera** o con un "_" e può contenere(dopo la prima lettera) solo **lettere**, **numeri** e "_". Inoltre, i nomi delle variabili non possono essere uguali a delle **parole chiave che sono “riservate” in C++.**
- Quando il nome di una variabile è composto da più parole, è comune seprare le diverse parole con un *underscore*: `int variabile_composta_da_diverse_parole;`

- Questo schema è noto come **snake case**. Alternativamente, è possibile usare il **camel case**, che non separa le parole, ma rende maiuscola la prima lettera di ciascuna parola: `int variabileCompostaDaDiverseParole`

`int` variabile di tipo intero

`float` una variabile di in virgola mobile.

`double` variabile più lunga di float.

`short int` variabile intera più corta in numero di bit di int.

`long` variabile di tipo intero più lunga in bit di int.

`long long` variabile di tipo intero più lunga in bit di long.

`char` contiene un singolo carattere, è formato da 8 bit e non ha segno.

`signed char` utilizza un bit per rappresentare il segno 1 o 0.

`string` contiene una serie di caratteri.

`auto` deduzione di tipo, è il compilatore stesso a capire il tipo di variabile.

`decltype ()` considera il TIPO della variabile indicata fra parentesi e lo applica alla variabile che si sta definendo.

ecc..

DICHIARAZIONE DI VARIABILE

```
auto x=2;
auto z=2.2;
auto y="Hello";

  cout << x << z << y;
// OUTPUT: 2 2.2 Hello
```

```
// esempio con decltype()
int x;
decltype(x) y = 2; //equivalente a int y = 2 visto che la 'x' è un intero

std::cout << "y= " <<y;
```

ESCAPE CODES

Gli escape codes sono dei comandi che servono per formattare il testo in output e si inseriscono direttamente nelle stringhe o vicino alle variabili.

`\n` newline

`\t` tab

`\v` vertical tab

ecc...

SOTTOSTRINGHE

`#include <string>` ← Libreria da includere

`.substr(x,y)` x=indice punto inizio sottostringa, y=indice fine sottostringa
Se la y viene omessa, il comando prenderà in considerazione tutto quello che segue la x.
`.length()` ha come valore la lunghezza della stringa, cioè il numero di caratteri, inclusi spazi.

COSTANTI `const` e `#define`

Queste costanti **DEVONO** essere inizializzate sennò avremo un errore di compilazione.

Espressioni o valori che hanno un valore fisso. Questo valore non può essere cambiato durante il programma.
Le costanti vengono dichiarate:

- mediante la parola `const` prima del tipo della variabile, all'interno dell'int main ();
- mediante la direttiva `#define` al di fuori del main, quindi è valida **globalmente**.

```
// sintassi const e #define
#include <iostream>

#define PI 3.14
using namespace std;

int main () {
     ...
}
    // oppure
#include <iostream>

using namespace std;

int main () {
      const float pi=3.14;
}
```

LETTERALI

Le costanti letterali hanno un **tipo**(float,int, double,string...) e possono essere dei numeri, caratteri, simboli o stringhe.

> `int x = 5` 5 è una costante letterale
> `char c='c'` è anche un letterale

Inoltre è possibile convertire il tipo di una variabile aggiungendo dei **suffissi** durante la sua dichiarazione, per esempio:

```
75         // int
75f        // float
75u        // unsigned int
```

```
75l        // long
75ul       // unsigned long
75lu       // unsigned long

int a=75f;
// equivale a
int a=(float)75;
```

Stringhe multiriga (`\`)

E' possibile inizializzare una stringa sulla stessa riga oppure fra più righe utilizzando il ***backslash "\"***.

```
#include <string>

using namespace std;
int main () {
        string s = "Stringa\
        multiriga\
        fine";
}
// \ si usa per mettere variabili in multiriga.
```

Prefissi `char`

I prefissi principali di `char` vengono utlizzati per decidere la quantità di bit che occuperà il char. In particolare è possibile distinguere:

Prefix	Character type
u	char16_t
U	char32_t
L	wchar_t

```
u'c'
U'c'
L'c'
```

I prefissi valgono anche per le stringhe:

`string s2=u"stringa";`

`R` come prefisso di una stringa mi stampa la stringa senza escapes, cioè in formato **grezzo**.

```
string s = R"(String with \backslash and no \n escape)";
cout << s;

// OUT: String with \backslash and no \n escape
```

> *Letterale nullptr (visto piu avanti)*

ESPRESSIONI COSTANTI CON UN NOME

Le epsressioni con costanti possono essere **inizializzate solo una volta.**

```
int h = 5;
h=0; //si puo fare

const int h = 5;
h=0 // non si puo fare piu perchè h è una costante
```

```
const float pi=3.14;
float r = 12.2;

float area = pi * r * r;
cout << area;
// Output: valore dell'area di un cerchio.
```

> Possono servire in `#include <cmath>` quando alcuni simboli vengono usati come costanti. Tipo il valore del pi greco già salvato in libreria ripescabile con la notazione `M_PI`

OVERFLOW E UNDERFLOW

`#include <climits>` ← Libreria da includere

```
#include <climits>

cout << INT_MAX; //Valore massimo intero rappresentabile
// output 2147483647 che è l'intero massimo.
```

Per **overflow** intendo un valore che ***supera il massimo valore rappresentabile.***
Per **underflow** si intende un valore che sta ***sotto il valore minimo rappresentabile.***

> *Se provo a sommare 1 mi dà errore di overflow e mi dà il valore minimo possibile. E' come azzerare i bit perchè comunque gli interi si ripetono in loop.*

Questo overflow o underflow può essere raggirato eseguendo una conversione del tipo:

```
cout << INT_MAX +1LL;
// cout << (long long) INT_MAX+1; è equivalente

avviene una conversione in long long
```

OPERAZIONI MATEMATICHE

`#include <cmath>` ← Libreria da includere

```
#include<iostream>
#include<cmath>

using namespace std;

int main(){
  cout << sqrt(2) << endl; //radice quadrata di due
  cout << cbrt(3) << endl; //radice cubica di tre
  cout << pow(2,3) << endl; //2 elevato 3
  cout << sin(M_PI) << endl; //seno di Pi greca
  cout << cos(2*M_PI); //coseno di 2*Pi greca
}
```

Arrotondamenti

`#include <cmath>` ← Libreria da includere

L'arrotondamento, in C++, può avvenire in vari modi mediante 3 comandi:

- `round` arrotonda per **eccesso**(*n,5+)* o per **difetto** (*n,4-)*, proprio come sappiamo fare noi;
- `floor` arrotonda **SEMPRE** per difetto, quindi elimina la parte decimale, qualsiasi esso sia il valore;
- `ceil` arrotonda **SEMPRE** per eccesso, qualsiasi esso sia il valore.

```
#include<cmath>
#include<iostream>

std::cout << round(2.2) << std::endl; //2
std::cout << round(2.6) << std::endl << std::endl; //3

std::cout << floor(2.2) << std::endl; //2
std::cout << floor(2.6) << std::endl << std::endl; //2

std::cout << ceil(2.2) << std::endl; //3
std::cout << ceil(2.6) << std::endl; //3
```

Not A Number (nan) e Infinito (inf, -inf)

`#include <cmath>` ← Libreria da includere

Non sono numeri veri e propri ma risultati di operazioni che non danno un numero reale.

Il risultato dell'espressione deriva da un'espressione **matematicamente non corretta**.

```
#include<cmath>

//NAN sta per "not a number" e indica valori numerici non validi
std::cout << NAN << std::endl;
//indica +infinito
```

```
std::cout << +INFINITY << std::endl;
//indica -infinito
std::cout << -INFINITY << std::endl;
```

```
std::cout << 0.0/0 << std::endl; //nan
std::cout << 1.0/0 << std::endl; //infinity
std::cout << -1.0/0 << std::endl; //-infinity
std::cout << 1/INFINITY << std::endl; //0
```

NAN è associato ad un float.

INPUT/OUTPUT DI BASE

`#include <iostream>` ← Libreria da includere

I **flussi** (**streams**) sono:

`cout` stampa a schermo numeri, stringhe o caratteri. quindi gestisce valori eterogenei.

`cin` permette all'utente di scrivere su terminale e mettere su una variabile l'input

`cerr` permette di presentare errori all'utente

`clog` permette all'utente di effettuare logging

Precisione di stampa (cout) e manipolazione output

`#include<iomanip>` ← Libreria da includere

E' possibile utilizzare dei comandi per decidere nella fase di stampa:

- il numero di **cifre totali** che dovrà avere un numero con la virgola utilizzando `setprecision()`;
- il numero di **cifre decimali** che dovrà avere un numero con la virgola utilizzando `fixed` prima di *setprecision();*
- se il numero da stampare dovrà essere scritto in **formato scientifico** utilizzando `scientific`;
- annullare la notazione scientifica utilizzando `defaultfloat`;
- annullare il *setprecision()* utilizzando `setprecision(-1)`
- quanti spazi stampare prima della stampa effettiva, utilizzando `setw()`.

```
#include <iomanip>
float x = 1.0/3;
std::cout << std::setprecision(0) << x << std::endl; //stampa comunque una cifra dopo la virgola
std::cout << std::setprecision(1) << x << std::endl; //una cifra dopo la virgola
std::cout << std::setprecision(3) << x << std::endl; //tre cifre dopo la virgola
std::cout << std::setprecision(20) << x << std::endl; //venti cifre dopo la virgola
std::cout << std::setprecision(1000) << x << std::endl; //ci fermiamo alla massima precisione possibile
```

```
std::cout << std::setprecision(2) << std::scientific << 1e-2 << std::endl;
std::cout << std::setprecision(2) << std::scientific << 1.2 << std::endl;
std::cout << std::setprecision(2) << std::scientific << 90.2 << std::endl;
```

```
std::cout << std::defaultfloat; //riporta alla visualizzazione di default
std::cout << std::setw(8) << 1e-2 << std::endl;
std::cout << std::setw(8) << 1.2 << std::endl;
std::cout << std::setw(8) << 90.2 << std::endl;
/*OUTPUT
0.01
     1.2
    90.2
*/
```

Input standard cin

`#include <iostream>` ← Libreria da includere

E' possibile dichiarare inizializzare una variabile mediante l'input da tastiera tramite:

- `cin` prende in considerazione quello che scrive l'utente **ESCLUSI** gli spazi;
- `getline (cin, variabile )` serve per **comprendere gli spazi** che l'utente inserisce.

```
#include<iostream>
using namespace std;

int main(){
    string s;
    cout << "Inserisci il tuo nome: ";
    cin >> s;
    cout << "Hello " << s << "!" << endl;
}
```

```
#include<iostream>
using namespace std;

int main(){
    string s;
    cout << "Inserisci il tuo nome: ";
  getline(cin, s);
    cout << "Hello " << s << "!" << endl;
}
```

Stringstream

`#include <sstream>` ← Libreria da includere

E' possibile concatenare stringhe, numeri e altri simboli e, inoltre, convertire numeri in stringhe e viceversa.

`ss` è una sorta di contenitore virtuale che può essere usato come cin e cout, cioè:

- `ss << x`
- `ss >> s`

Possiamo convertire uno `stringstream` in stringa anche usando il metodo `str`.

Il metodo `str` può anche essere utilizzato per cambiare il valore dello `stringstream`.

`ss.str()` si usa solitamente per inizializzare ss.
Es: `ss.str(x)` *ss* avrà dentro la variabile *x.*

```
#include<iostream>
#include<string>
#include<sstream>
using namespace std;

int main(){
  stringstream ss;
  string s;
  int x = 5;

  ss << x;
  ss >> s; //la stringa s adesso contiene il numero 5
  cout << s;
}
```

```
stringstream ss;
ss << "Hello " << 5;
string s = ss.str();
cout << s;
```

oppure

```
#include<iostream>
#include<sstream>

using namespace std;

int main(){
  stringstream s;
  s << "ciao";
  cout << s.str() << endl;
  s.str("help"); //cambia il valore in "help"
  cout << s.str() << endl;
  s.str(""); //"svuota" lo stream
  cout << s.str() << endl;
}
```

COSTRUTTI DI BASE

If then else

Verifica se una condizione è valida o no. Se valida si fa una cosa, altrimenti un'altra cosa.
La condizione può essere anche un booleano (true o false) o un intero.
Se è un intero(o qualsiasi altro tipo di numero), allora la condizione sarà **SEMPRE VERA** finchè il numero è diverso da 0.

```
#include<iostream>
using namespace std;

int main(){
  int x;
  cout << "Indovina il numero: ";
  cin >> x;
  if (x == 5)
    cout << "Complimenti, il numero inserito e\' corretto!";
}
```

Ciclo *while*

`while` si ripete finchè la condizione è vera e si ferma quando la condizione è falsa. Solitamente si utilizza un contatore *i* per tenere conto delle iterazioni che dovranno essere eseguite.

```
#include<iostream>
using namespace std;

int main(){
    int x = 1;

    while(x<=10) {
      cout << x << endl;
      x++;
  }
}
```

Ciclo *do while*

Fa una cosa simile a while ma la prima istruzione venga **SEMPRE** eseguita.

```
#include<iostream>
using namespace std;

int main(){
    int x;

    do {
        cout << "Indovina il numero: " << endl;
        cin >> x;
    } while(x!=5)
}
```

ASSEGNAMENTO COMPOSTO

E' possibile fare delle operazioni di assegnamento nei seguenti modi:

```
float x = 1;
x+=1; //equivalente a x = x+1
x*=5; //equivalente a x = x*5
x-=3; //equivalente a x = x-3
x/=2; //equivalente a x = x/2
```

OPERATORI LOGICI || E &&

```
Sintassi: a && b
          a || b

int x=5;
bool y= true || (x++);

cout << x; // OUTPUT: 5
```

OPERATORE CONDIZIONALE TERNARIO (?)

```
Sintassi: (a < b ? a : b)
int x=5;
int y=7;
int z;

if (x>y){
    z=x;
} else {
  z=y;
}

z= (x>z) ? x : y; //OUTPUT 7
```

OPERATORE VIRGOLA (,)

L'operatore virgola (`,`) permette di inserire più espressioni laddove ne è richiesta solo una. L'espressione restituisce il valore più a destra.

```
int x = 2;
int y = 3;
//questa espressione prima incrementa x, poi decrementa y e infine restituisce x*y
int c = (x++, y--, x*y);
std::cout << c;
```

OPERATORI BIT A BIT

Gli operatori bit a bit confrontano i singoli bit con i propri corrispondenti e fanno l'operazione logica richiesta di AND, NOT, OR ecc...

In output mandano il risultato della somma (OR), prodotto (AND) ecc..

- AND → &

- NOT → ~ (ALT + 0126)
- XOR → ^
- OR → |
- SHIFT RIGHT → >>
- SHIFT LEFT → <<

```
int x=3; //011=3
int y=4; //100=4

// 011 & 100 = 000

// AND '&'
// 011 &
// 100
// 000

// XOR ^
// 011^
// 101
// 110 -> 6

// NOT ~
// ~011=100 -> 4 (?)

// (3<<1) 3->011 -> 110 ->6
// (3<<2) 3->011 -> 1100 -> 12
// (3>>2) 3->011 .> 001
cout << (x & y); // output: 0
cout << (x ^ y); // output: 6
cout << (x | y); // output: 7
cout << (x & y); // output: 0
cout << (3<<1); // output: 6
cout << (3<<2); // O: 12
cout << (3>>2); // o: 1
```

OPERATORE DI CASTING DI TIPO ESPLICITO

Permette di **convertire** una variabile da un tipo ad un altro tipo:

```
float x = 2.2;
int y = (int) x; // (cast)
std::cout << y; // Output 2

float x = 2.5;
int y = (int) x; // (cast)
std::cout << y; // Output 3

float x = 2.49;
int y = (int) x; // (cast) MODO 1
int z= int(x) // int(cast) MODO 2
int w=static_cast<int>(x); // static_cast<int> MODO 3
// y = z = w
std::cout << y; // Output 2
std::cout << z; // Output 2
std::cout << w; // Output 2
std::cout << float(3)/2; // output: 1.5
```

sizeof()

Serve per vedere il numero di byte che occupa un TIPO di variabile.

```
cout << sizeof(int); // output: 4
cout << sizeof(long double); // output: 12 BYTE di un long double
cout << sizeof(2); // O: 4 BYTE di un int
cout << sizeof(2.0); // O: 8 byte di un double
cout << sizeof(2.0f); // O: 4 BYTE di un float
```

PRECEDENZA DEGLI OPERATORI

Il calcolatore effettua le operazioni valutando i vari operatori e le proprie precedenze in un'espressione.

Per agevolare i calcoli e la semantica **conviene sempre utilizzare le parentesi tonde.**

```
2+3*5+7 //precedenza del prodotto sulla somma
int x=(2+3)*3 // diverso da 2+3*3
// 2+(3*3) -> 2+3*3
```

OPERAZIONI TRA STRINGHE

```
std::string x = "Hello";
std::string y = "world";
std::string message = x + " " + y + "!";
std::cout << message;
// O: Hello world

std::string x = "Hello";
std::string y = "Goodbye";

std::cout << "x==y -> " << (x==y) << std::endl;
std::cout << "x!=y -> " << (x!=y) << std::endl;
std::cout << "x>y -> " << (x>y) << std::endl;
std::cout << "x<y -> " << (x<y) << std::endl;

/*
   OUTPUT
  x==y -> 0
  x!=y -> 1
  x>y -> 1
  x<y -> 0
*/
```

CICLI

Ciclo for basato su range

```
for (dichiarazione: range)
    istruzione;

std::string s = "Hello World!";
for(char c: s)
    std::cout << c << " - ";
// O: H - e - l - l - o -   - W - o - r - l - d - ! -
```

GOTO

Goto serve per **saltare** e andare in un punto del codice denotato con **un'etichetta**.

```
#include <iostream>
using namespace std;

int main () {
    int x=1;
etichetta:
    cout << x << endl;
    x++;
    if (x<=10) goto etichetta;

  // EQUIVALE
#include <iostream>
using namespace std;

int main () {
  for(int x = 1; x<=10; x++)
    cout << x << endl;
}
}
```