Linguaggi di Programmazione - Base

C++

Alex Ergasti, PhD student.
alex.ergasti@unipr.it

11 settembre 2025

Sommario

Sezione attuale

C++: un linguaggio strutturato e potente

C++ è considerato un linguaggio di programmazione ben strutturato, ed è proprio per questo che è così popolare.

Essendo un linguaggio di alto livello, **consente ai programmatori di concentrarsi sul problema da risolvere**, senza preoccuparsi del sistema su cui verrà eseguito il programma.

Molti linguaggi si dichiarano indipendenti dalla macchina, ma C++ è tra i migliori in questo ambito.

Le origini del linguaggio C++

Come molti altri linguaggi, **C++** deriva fondamentalmente da **ALGOL**, il primo linguaggio con una struttura definita.

Sviluppato nei primi anni ’60, **ALGOL** ha aperto la strada alla programmazione strutturata.

Il primo lavoro teorico fu condotto da **Corrado Böhm** e **Giuseppe Jacopini**, che nel 1960 pubblicarono una ricerca fondamentale sui concetti di programmazione strutturata.

Le origini del linguaggio C++

Nel 1967, **Martin Richards** progettò il linguaggio **BCPL** (Basic Combined Programming Language).

Nel 1970, **Ken Thompson** sviluppò il linguaggio **B**.

Nel 1972, **Dennis Ritchie** introdusse il linguaggio **C**.

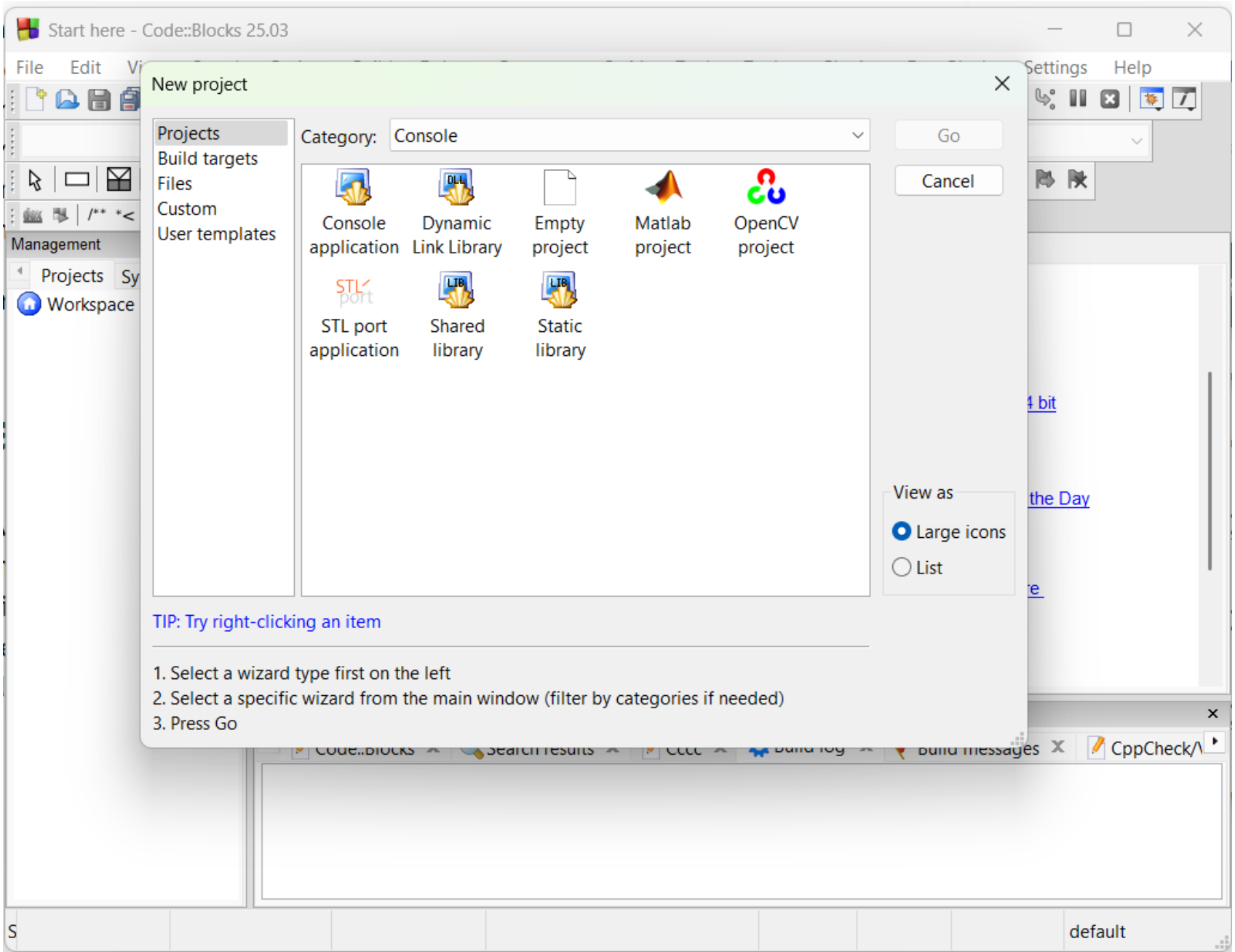
Ispirandosi ai linguaggi **ALGOL**, **BCPL**, **B** e **C**, **Bjarne Stroustrup** sviluppò **C++** a metà degli anni '80.

Linguaggi imperativi

C++ è un linguaggio imperativo, ovvero le istruzioni vengono eseguite secondo l'ordine imposto.

> In informatica, la **programmazione imperativa** è un paradigma di programmazione secondo cui un programma viene inteso come un **insieme di istruzioni** (dette anche direttive o comandi), ciascuna delle quali può essere pensata come un "**ordine**" che viene impartito alla macchina virtuale del linguaggio di programmazione utilizzato.

Hello world, creare il progetto



Hello world

```cpp
#include <iostream>
int main() {
    std::cout << "Ciao mondo!" << std::endl;
    return 0;
}
```

- `#include <iostream>`: include la libreria per input/output
- `int main()`: funzione principale, punto di ingresso del programma
- `std::cout`: oggetto per l'output su console
- `<<`: operatore di inserimento (inserisce dati nello stream)
- `std::endl`: inserisce una nuova riga e svuota il buffer
- `return 0`: indica che il programma è terminato correttamente

Ogni riga di codice deve terminare con il `;`

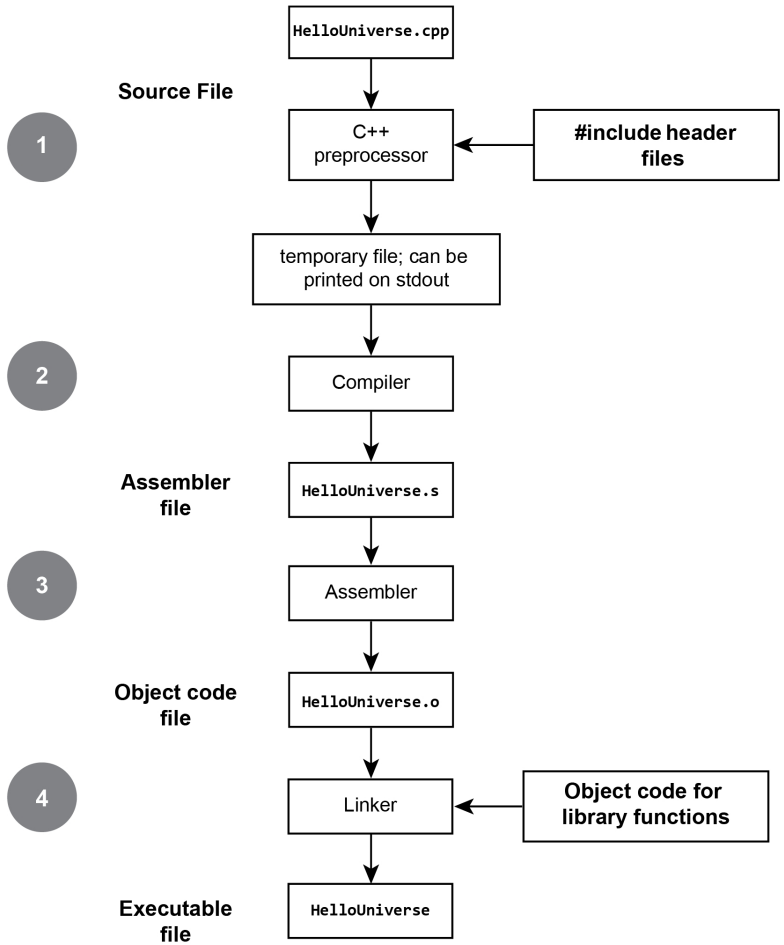
Include

L'utilizzo di `include` dice al compilatore di importare le dichiarazione di una determinata libreria. In questo modo è possibile utilizzare senza conoscere nel dettaglio l'implementazione.

Sarà il compilatore poi ad aggiungere l'effettiva implementazione nel file compilato.

Come si esegue il codice?

Il codice in se è solo un file di testo senza un grosso signficiato, per questo è necessario il compilatore!

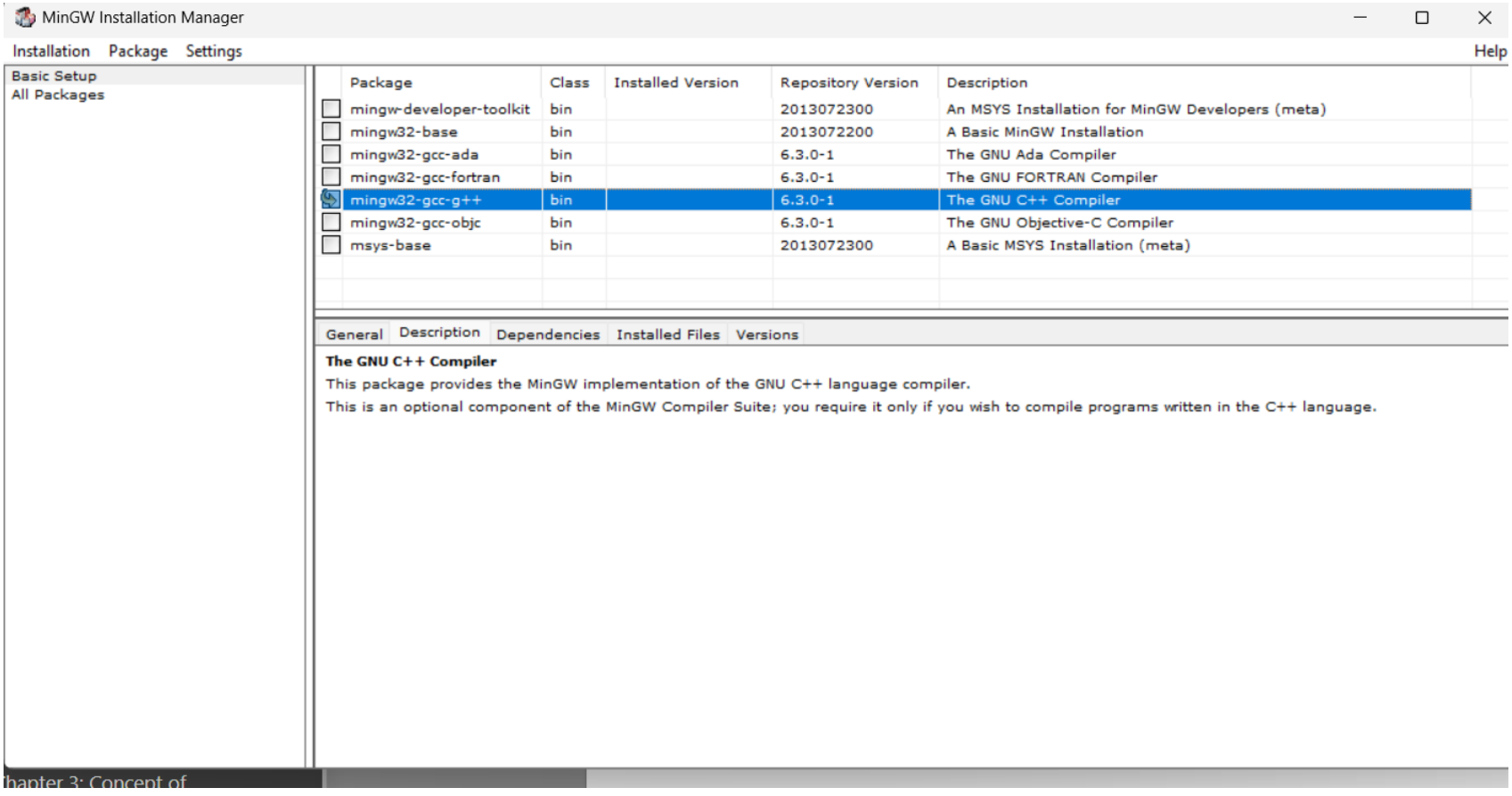


Code Block

In questo corso useremo l'IDE CodeBlock. Un'IDE è solo un lettore di testo che aiuta nella scrittura del codice, non è in grado di compilare senza un compilatore esterno!

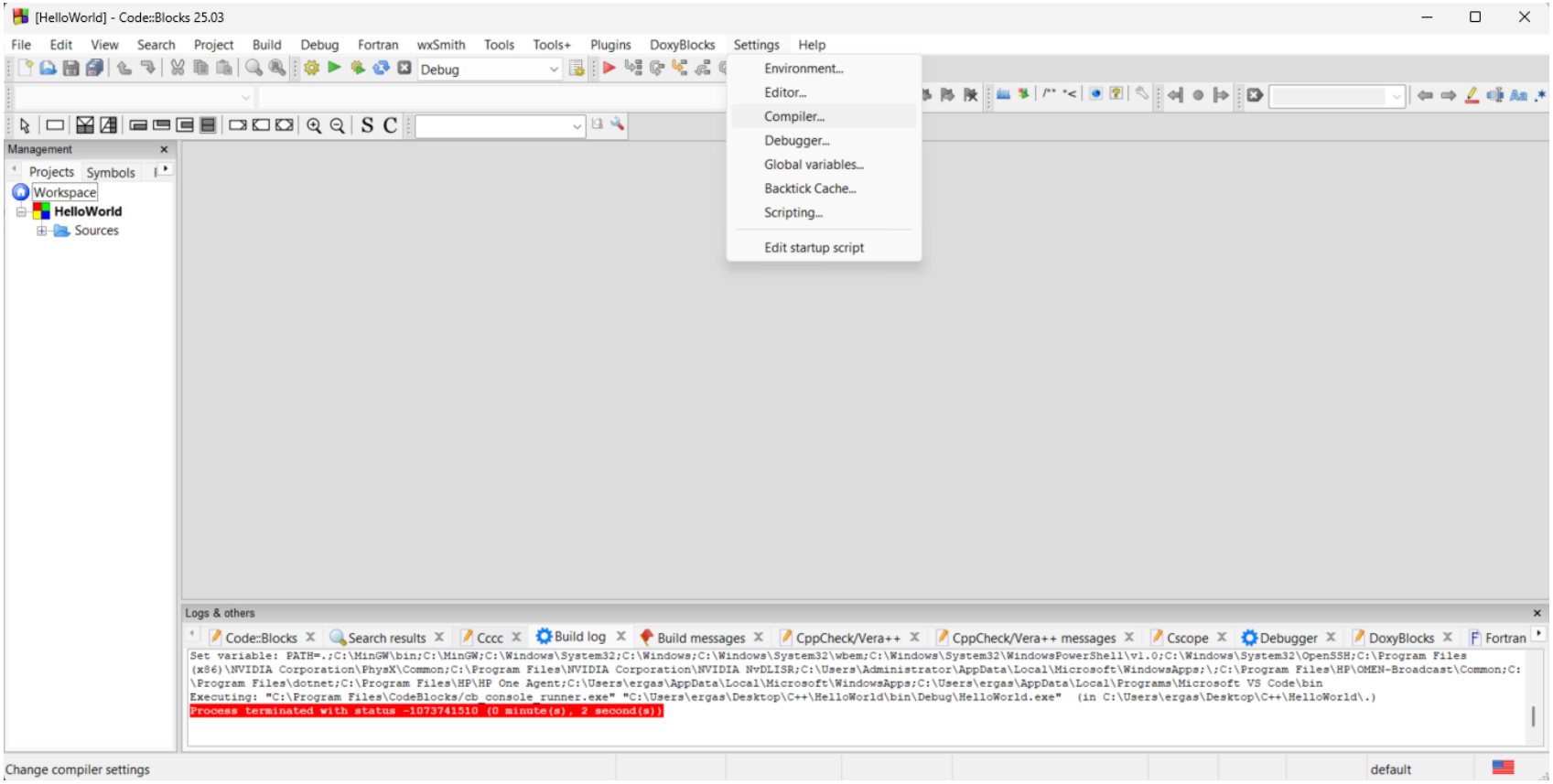
Per questo, sarà necessario installare anche MinGW su windows o GCC su Linux per poter compilare i nostri files.

MinGW



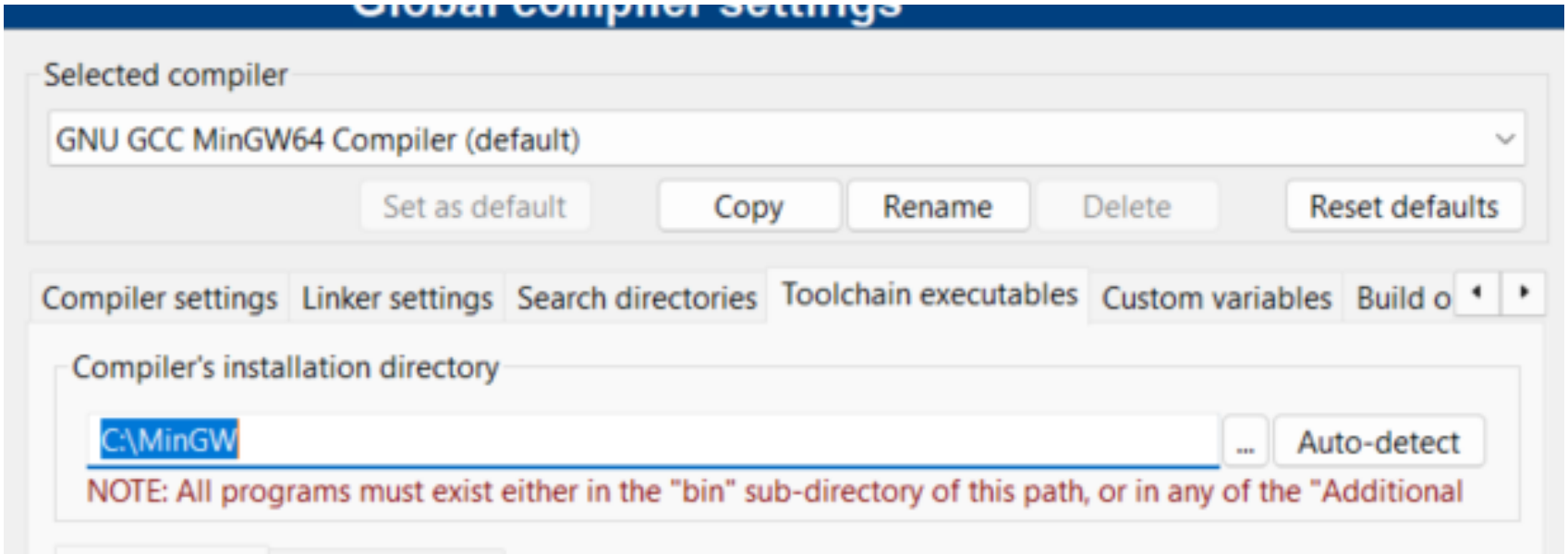
Selezionare l'opzione come nelle slide e poi in "Installation > Apply changes"

CodeBlock



Da qui si può poi selezionare MinGW64 come compiler di codeblocks

CodeBlock compiler



Sezione attuale

Tipo di variabile

Una variabile è una zona di memoria a cui viene dato un nome. Le variabili sono accessibili dal programma in base allo scope. Lo scope definisce in maniera chiara le variabili **visibili**, e quindi **utilizzabili** in un determinato punto di codice. Analizzeremo più avanti le varie tipologie di scope.

Le variabili sono caratterizzate da 2 cose, un **nome** e un **tipo**.

Il nome di una variabile è **case sensitive** e la identifica univocamente all'interno dello scope

Il tipo di una variabile definisce il contenuto di quella variabile.

Integer

Value in programming	Number	Data Type
98	+98	int
-865	-865	int
-68495L	-68495	long int
984325LU	984325	unsigned long int

```
sizeof (long int) => sizeof (int) => sizeof
(short int)
```

Tipi di interi

```
short int l; //2 byte
unsigned short int a; //2 byte
int i; //4 byte
unsigned int j; //4 byte
long int k; //sizeof(z) >= 4 byte
long long int z; //sizeof(z) >= 8 byte
unsigned long long int y; //sizeof(y) >= 8 byte
```

La size di `long long` dipende dal sistema, `long int` è 4 byte in sistemi a 32 bit, 8 byte in sistemi a 64 bit.

Character

Un singolo carattere, rappresentato tra ”, è codificato dalla tipologia **char**. Un carattere ha per altro un mapping 1:1 con un uint8 (da 0 a 255), definito dalla tabella ASCII

```
char c = 'a'; //97
```

ASCII Table

Dec	Hex	Char	Name	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	NUL	null	32	20	Space	64	40	@	96	60	`
1	1	SOH	start of heading	33	21	!	65	41	A	97	61	a
2	2	STX	start of text	34	22	"	66	42	B	98	62	b
3	3	ETX	end of text	35	23	#	67	43	C	99	63	c
4	4	EOT	end of transmission	36	24	$	68	44	D	100	64	d
5	5	ENQ	enquiry	37	25	%	69	45	E	101	65	e
6	6	ACK	acknowledge	38	26	&	70	46	F	102	66	f
7	7	BEL	bell	39	27	'	71	47	G	103	67	g
8	8	BS	backspace	40	28	(	72	48	H	104	68	h
9	9	HT	horizontal tab	41	29	)	73	49	I	105	69	i
10	A	LF	line feed	42	2A	*	74	4A	J	106	6A	j
11	B	VT	vertical tab	43	2B	+	75	4B	K	107	6B	k
12	C	FF	form feed	44	2C	,	76	4C	L	108	6C	l
13	D	CR	carriage return	45	2D	-	77	4D	M	109	6D	m
14	E	SO	shift out	46	2E	.	78	4E	N	110	6E	n
15	F	SI	shift in	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	data link escape	48	30	0	80	50	P	112	70	p
17	11	DC1	device control 1	49	31	1	81	51	Q	113	71	q
18	12	DC2	device control 2	50	32	2	82	52	R	114	72	r
19	13	DC3	device control 3	51	33	3	83	53	S	115	73	s
20	14	DC4	device control 4	52	34	4	84	54	T	116	74	t
21	15	NAK	negative acknowledge	53	35	5	85	55	U	117	75	u
22	16	SYN	synchronous idle	54	36	6	86	56	V	118	76	v
23	17	ETB	end of transmission block	55	37	7	87	57	W	119	77	w
24	18	CAN	cancel	56	38	8	88	58	X	120	78	x
25	19	EM	end of medium	57	39	9	89	59	Y	121	79	y
26	1A	SUB	substitute	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	escape	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	file separator	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	US	unit separator	63	3F	?	95	5F	_	127	7F	DEL

From https://ajsmith.org/tools/ascii-table/

Character

I character internamente sono rappresentati come interi, è quindi agevole passare da char a int seguendo la tabella ASCII.

```
#include <iostream>
int main(){
    char c = 'a'+3;
    std::cout << static_cast<int>(c) << ": " << c;
    //100: d
}
```

Float

I numeri con la virgola si definiscono float, double o long double e varia in base alla loro precisione.

```
sizeof (long double) => sizeof (double) =>
sizeof (float)
```

Boolean

I booleani sono variabili che contengono un valore di verità, codificano l'informazione di vero o falso.

Costanti

Le costanti sono variabili il cui valore rimane fisso sempre e viene deciso a compile-time. C++ permette due modi di definire le costanti. Tramite `#DEFINE` e tramite la keyword `const` davanti al tipo di variabile.

```
#define COSTANTE 42
const int costante = 42;
```

La differenza è che il define viene sostituito dal compilatore, la keyword const invece crea una vera e proprio variabile in memoria il cui però valore non può essere modificato.

Inizializzazione una variabile

Ogni variabile in C++ deve essere **dichiarata** e **inizializzata** prima di essere utilizzata.

> Dichiarare una variabile significa dare un tipo e un nome e assegnare una zona di memoria. Inizializzare una variabile significa dare un valore preciso ad una variabile.

```
int main()
{
    //Dichiarazione
    int pippo;

    //Inizializzazione
    pippo = 15;

    //Dichiarazione e inizializzazione
    int pluto = 32;
}
```

Casting di variabili

Si può convertire una variabile da un tipo ad un'altra in 2 maniere, **implicito** o **esplicito**. La differenza è che quando è implicito il programmatore non deve esplicitare la conversione.

```
int main()
{
    float pi = 3.14;
    int pi_greco = (int)pi;
    int pi_greco2 = static_cast<int>(pi);

    char c = 'A';
    int ascii = static_cast<int>(c); //ascii contiene 65
}
```

Quale casting usare?

Meglio usare `static_cast<T>(variabile)` poichè il compilatore verifica se la conversione è effettivamente fattibile, se non lo è, il compilatore da un errore.

Usare `(T)variabile` è una conversione non sicura che non da errori se la conversione non è fattibile. Quello che avviene è particolare.

IOstream

La direttiva `#include <iostream>` include la libreria per la gestione dell'**input/output standard** in C++.

Questa libreria consente di:

- Usare `std::cout` per stampare dati sulla console (output)
- Usare `std::cin` per leggere dati da tastiera (input)
- Usare `std::endl` per andare a capo e svuotare il buffer
- Gestire stream di dati in ingresso e in uscita in modo efficiente

std

E' possibile evitare di scrivere `std::` importando il namespace con `using namespace std`, in questo modo si sta dicendo al compilatore di cercare le varie operazioni anche all'interno della libreria standard di C++, in particolare nel namespace `std`. C'è da stare attenti però!

> Una libreria è un insieme di funzioni (vedremo poi cosa sono, per ora consideratele del codice) scritto da altre persone che noi possiamo riutilizzare senza reinventare la ruota

Possibili problemi

- Inquinamento del namespace: se varie librerie hanno funzioni con lo stesso nome, il compilatore non sa quale deve usare
- Difficile capire da quale libreria arriva una determinata funzione

```cpp
#include <iostream>

int main() {
  int numero;
  std::cout << "Inserisci un numero: ";
  std::cin >> numero;
  std::cout << "Hai inserito: " << numero << std::endl;
  return 0;
}
```

Un po’ di esercizi

- Leggere un carattere da standard input e stampare il valore intero relativo
- Fare un programma che stampa
  A
  AA
  AAA
- Leggere un intero, un float e un char e stamparli in questo ordine

Esercizio 1

```cpp
#include <iostream>

int main() {
    char c;
    std::cout << "Inserisci un numero: ";
    std::cin >> c;
    std::cout<<"Hai inserito: "<< static_cast<int>(c);
    return 0;
}
```

Esercizio 2

```cpp
#include <iostream>

int main() {
    std::cout << "A"   << std::endl;
    std::cout << "AA"  << std::endl;
    std::cout << "AAA" << std::endl;
    return 0;
}
```

Sezione attuale

Operatori

Operatori aritmetici

Il linguaggio C++ è dotato di tutti i comuni operatori aritmetici:

- somma `+`
- differenza `-`
- moltiplicazione `*`
- divisione `/`
- modulo `%`

La divisione

La divisione è un'operazione particolare, se si divide tra 2 interi si ottiene sempre un intero, per avere una divisione con la virgola serve almeno un float.

```
#include <iostream>

int main() {
    int x = 7;
    int y = 3;

    float k = x / y;
    std::cout << k <<std::endl;

    float z = static_cast<float>(x) / y;
    std::cout << z <<std::endl;
}
```

Esempi del modulo

```cpp
#include <iostream>

int main() {
    int x = 7;
    int y = 16;
    int z = 0;
    int k;
    k = y % x; // restituisce 2
    k = x % y; // restituisce 16
    k = y % z; // restituisce un messaggio di errore.
}
```

Operatori unari

Gli operatori unari permettono di incrementare o decrementare il valore di una variabile in una sola riga di codice

```
#include <iostream>
using namespace std;
int main(){
    int i=10;
    i++; //i=i+1
    cout << i << endl; //11
    ++i; //i=i+1
    cout << i << endl; //12

    i--; //i=i-1
    cout << i << endl; //11
    --i; //i=i-1
    cout << i << endl; //10
}
```

Operatori unari e assegnamento

```cpp
#include <iostream>
using namespace std;
int main(){
    int i=10;
    int x=++i;
    cout << "x: " << x << " i: " << i << endl; //x: 11 i: 11

    int y=--i;
    cout << "y: " << y << " i: " << i << endl; //y: 10 i: 10

    int z=i++;
    cout << "z: " << z << " i: " << i << endl; //z: 10 i: 11

    int k=i--;
    cout << "k: " << k << " i: " << i << endl; //k: 11 i: 10
}
```

Risultato

L'operatore `i++` e `++i` (come `i--` e `--i`) funzionano allo stesso modo se usati per incrementare **una variabile**.
Se usati in combinazione con un **assegnamento** si ha:

- `x=++i`, prima incrementa `i` e poi assegna il valore di `i` a `x`.
- `x=i++`, prima assegna il valore di `i` a `x` e poi incrementa `i`

Operatori

Operatori di uguaglianza

Un operatore di uguaglianza è un operatore che verifica determinate condizioni come: ”è minore di” oppure ”è maggiore di” oppure ancora ”è uguale a”. Un operatore di uguaglianza viene utilizzato per determinare il tipo di azione da intraprendere al verificarsi di certe situazioni. Nella seguente tabella, è riportato l’elenco completo degli operatori di uguaglianza:

Operatore	Simbolo	Esempio d’uso	Risultato
Minore	<	bool ris = (3 < 8)	true
Maggiore	>	bool ris = (3 > 8)	false
Uguale	==	bool ris = (5 == 5)	true
Minore Uguale	<=	bool ris = (3 <= 5)	true
Maggiore Uguale	>=	bool ris = (5 >= 9)	false
Diverso	!=	bool ris = (4 != 9)	true

Operatori

Regole di precedenza degli operatori

Le regole di precedenza degli operatori indicano l'ordine con cui gli operatori vengonoeseguiti nell'ambito di una espressione. Un operatore con precedenza maggiore verrà valutato per prima rispetto ad un operatore con precedenza minore, anche se quest'ultimo figura prima dell'operatore con precedenza maggiore.

```
int risultato = 4 + 5 * 7 + 3;
```

L'operatore di moltiplicazione (*) ha precedenza rispetto all'operatore addizione (+). Ciò vuol dire che la moltiplicazione `5 * 7` avverrà prima di tutte le altre addizioni. Avremo quindi, al termine:

```
int risultato = 4 + 35 + 3; //42
```

Volendo variare questo ordine di interpretazione delle operazioni, è necessario ricorrere all'uso delle arentesi tonde, la cui funzione è di raggruppare le operazioni:

```
int risultato = (4 + 5) * (7 + 3); // 90
```

in questo caso l'operazione di moltiplicazione verrà effettuata dopo aver eseguito le espressioni contenute nelle due paretesi (che sono diventate gli operandi della moltiplicazione)

Esercizi con le operazioni

Chiedere 2 numeri all'utente e stampare il risultato delle 5 operazioni (`+`, `-`, `*`, `/`, `%`)

```cpp
#include <iostream>
using namespace std;
int main()
{
    int num1, num2;
    cout << "Inserisci due numeri: ";
    cin >> num1 >> num2;
    int somma = num1 + num2;
    int sottrazione = num1 - num2;
    int moltiplicazione = num1 * num2;
    float divisione = static_cast<float>(num1)/num2;
    cout << "Somma: " << somma << endl;
    cout << "Sott: " << sottrazione << endl;
    cout << "Molt: " << moltiplicazione << endl;
    cout << "Div: " << divisione << endl;
    return 0;
}
```

Sezione attuale

Istruzioni Condizionali

Le istruzioni condizionali permettono di eseguire una parte di codice o un'altra a seconda della necessittà.

Una prima divisione comprende le istruzioni:

- `if-else`
- `switch`

If

Le istruzioni if ed else, analizzando il significato in Inglese corrispondono a Se ed Allora. Intuitivamente, si utilizza questo costrutto per eseguire un gruppo di istruzioni al verificarsi di determinate condizione.

```
if (<condizione>)
{
    istruzione 1;
    istruzione 2;
}
```

> `<condizione>` è un espressione che viene valutata dal programma come vera o falsa, formata quindi da operatori di uguaglianza

If-else

```
if (<condizione>)
{
    istruzione 1;
    istruzione 2;
}
else{
    istruzione 3;
    istruzione 4;
}
```

If-else if

```
if (<condizione 1>)
{
    istruzione 1;
    istruzione 2;
}
else if(<condizione 2>){
    istruzione 3;
    istruzione 4;
}
```

If-else if-else

```
if (<condizione 1>)
{
    istruzione 1;
    istruzione 2;
}
else if (<condizione 2>){
    istruzione 3;
    istruzione 4;
}
else{
    istruzione 5;
    istruzione 6;
}
```

Esericizi if

- Dato un numero inserito dall'utente, stampare se positivo o negativo
- Chiedi all'utente la sua età. Se ha almeno 18 anni, può guidare. Altrimenti no.
- Chiedi all'utente un numero e stampa se è pari o dispari.
- Chiedi due numeri all'utente e stampa il maggiore (o se sono uguali).
- Inserisci un voto da 0 a 10 e stampa una valutazione (tipo sufficiente, buono, ottimo).

Operatori logici

La condizione non per forza è definita da una singola condizione, ma può essere formata da 2 o + condizioni, concatenate tramite operatori logici.

Operator	**Demonstration**
!	not
&&	Logical and
||	Logical or

Operatore AND (&&)

L'operatore AND valuta a vero se e solo se entrambe le condizioni sono vere

A	B	Ris
0	0	0
0	1	0
1	0	0
1	1	1

Operatore OR (||)

L'operatore OR valuta a vero quando anche solo una delle condizioni è vera

A	B	Ris
0	0	0
0	1	1
1	0	1
1	1	1

Operatore NOT (!)

L'operatore NOT cambia il valore di verità

A	Ris
0	1
1	0

Esempi

```
#include <iostream>
using namespace std;
int main() {
    int n = 5;
    int m = 20;
    if(n >= 0 && m >= 0)
    {
        cout << "Numeri positivi"<< endl ;
    }
    else if(n < 0 && m < 0)
    {
        cout << "Numeri negativi"<< endl ;
    }
    else{
        cout << "Uno positivo e uno negativo" << endl;
    }
    return 0;
}
```

Esercizi

- **Divisibilità per 3 o per 5**
  Chiedi all'utente un numero e controlla se è divisibile per 3 **o** per 5. Usa l'operatore `or`. Se sì, stampa "Divisibile", altrimenti "Non divisibile".
- **Verifica voto valido**
  Chiedi all'utente di inserire un voto tra 0 e 10. Se il valore non è in questo intervallo, mostra "Valore non valido". Usa l'operatore `not` con una condizione composta.
- **Temperatura accettabile**
  Chiedi all'utente di inserire una temperatura. Se è compresa tra 18 e 26 gradi inclusi, stampa "Temperatura ideale", altrimenti "Temperatura fuori intervallo". Usa l'operatore **and**.
- **Numero maggiore tra 3** Dati 3 numeri dall'utente, stampare il maggiore

Switch

Alternativa a una serie di `if-else if` per confronti su valori interi o caratteri.

```
switch (variabile) {
  case valore1:
    // codice
    break;
  case valore2:
    // codice
    break;
  default:
    // codice di default
}
```

Ogni `case` deve concludersi con `break` per evitare il "fall-through". Il blocco `default` è facoltativo ma consigliato.

Esempio: Giorno della settimana

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

int main() {
    int giorno;
    cout << "Inserisci un numero (1-7): ";
    cin >> giorno;

    switch (giorno) {
        case 1: cout << "Lunedì"; break;
        case 2: cout << "Martedì"; break;
        case 3: cout << "Mercoledì"; break;
        case 4: cout << "Giovedì"; break;
        case 5: cout << "Venerdì"; break;
        case 6: cout << "Sabato"; break;
        case 7: cout << "Domenica"; break;
        default: cout << "Numero non valido";
    }

    return 0;
}
```

Esercizio operazioni

```cpp
#include <iostream>

using namespace std;

int main()
{
    float oper1;
    float oper2;
    char operazione;

    cout << "Inserisci l'operazione che vuoi fare:" << endl;
    cin >> oper1 >> operazione >> oper2;

    switch (operazione)
    {
        case '+':
            cout << oper1+oper2;
            break;
        case '-':
            cout << oper1-oper2;
            break;
        case '*':
            cout << oper1*oper2;
            break;
        case '/':
            cout << oper1/oper2;
            break;
        default:
            cout << "Simbolo non valido";
    }

    return 0;
```

Esempio: `switch` senza `break` (Fall-through)

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

int main() {
    int voto;
    cout << "Inserisci un voto (1-3): ";
    cin >> voto;

    switch (voto) {
        case 1:
            cout << "Insufficiente\n";
        case 2:
            cout << "Sufficiente\n";
        case 3:
            cout << "Buono\n";
        default:
            cout << "Valutazione terminata\n";
    }
```

Esercizi con `switch`

- Scrivi un programma che stampi il nome del mese dato un numero da 1 a 12.
- Crea una calcolatrice semplice che, dato un operatore `+ - * /`, esegua l'operazione tra due numeri.
- Chiedi all'utente un voto da A a F e stampa il giudizio:
  - A: Eccellente, B: Buono, C: Sufficiente, D: Insufficiente, F: Gravemente insufficiente.

Sezione attuale

Variabili Complesse

Array

Array in C++

- Un **array** è una collezione di variabili dello stesso tipo, memorizzate in posizioni contigue di memoria.
- Ogni elemento ha un **indice**, che parte da 0.
- Gli array hanno dimensione fissa nota durante l'inizializzazione.
- Esempi:

```
int numeri[5];  // dichiarazione
int voti[3] = {18, 24, 30}; // inizializzazione
```

Accesso agli elementi

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

int main() {
    int voti[3];

    voti[0] = 20;
    voti[1] = 25;
    voti[2] = 30;

    cout << "Voto 1: " << voti[0] << endl;
    cout << "Voto 2: " << voti[1] << endl;
    cout << "Voto 3: " << voti[2] << endl;

    return 0;
}
```

Come ottenere la lunghezza di un array in C++

- Gli array in C++ non hanno un metodo ‘.length()‘ come in altri linguaggi.
- Si usa l’operatore **`sizeof`** per calcolare:

```
sizeof(array) / sizeof(array[0])
```

- Esempio pratico:

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

int main() {
    int numeri[] = {5, 10, 15, 20};

    int dimensione = sizeof(numeri) / sizeof(numeri[0]);

    cout << "Dimensione array: " << sizeof(numeri) <<endl;
    cout << "Dimensione elemento array: " << sizeof(numeri[0]) << endl;

    cout << "Lunghezza dell'array: " << dimensione << endl;

    return 0;
}
```

Funziona solo per array **statici** (non passati a funzione). Vedremo poi cosa è una funzione.

Errori comuni con gli array

- ✗ Accesso fuori dai limiti:

```
int a[3];
a[3] = 10; // Errore: indice fuori limite (0, 1, 2)
```

- ✗ Uso di valori non inizializzati:

```
int b[4];
cout << b[0]; // Comportamento indefinito
```

- ✓ Soluzione: inizializzare gli array

```
int c[3] = {0}; // Tutti gli elementi = 0
```

Variabili Complesse

Vector

Cosa sono i vector

Un grosso problema degli array è che sono di dimensione fissa, in C++ (rispetto a C) sono stato aggiunti i `vector`, i quali sono array ridimensionabili e che non soffrono dei vari problemi che vedremo poi degli array. Per utilizzare i vector è necessario includere `include <vector>`.

Per dichiarare un vector si usa la seguente sintassi `vector<type> name;`

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;
int main(){
    // Create a vector called cars that will store strings
    vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

    // Print vector elements
    for (string car : cars) {
      cout << car << "\n";
    }
}
```

Inizializzazione di un vector

```
//Vettore lungo 5 vuoto
vector<string> myVector(5);
//Vettore lungo 7 con "hello" ovunque
vector<string> myVector(7,"hello");
```

Accesso ad un vector

L'accesso agli elementi di un vector avviene esattamente come un array

```
// Create a vector called cars that will store strings
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Get the first element
cout << cars[0];  // Outputs Volvo

// Get the second element
cout << cars[1];  // Outputs BMW
```

Metodi utili

L'implementazione dei vector permette inoltre di utilizzare una serie di metodi utili, ne vediamo alcuni.

Front e back

```
// Create a vector called cars that will store strings
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Get the first element
cout << cars.front();

// Get the last element
cout << cars.back();
```

At

L'utilizzo di `.at()` va preferito rispetto all'utilizzo delle `[]` poichè più sicuro e permette di sapere se si sta accedendo ad un elemento fuori dall'array tramite un errore.

```
// Create a vector called cars that will store strings
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Get the second element
cout << cars.at(1);

// Get the third element
cout << cars.at(2);

// Try to access an element that does not exist (throws an error message)
cout << cars.at(6);

// Change the value of the first element
cars.at(0) = "Opel";

cout << cars.at(0);  // Now outputs Opel instead of Volvo
//si può fare anche cars[0]="Opel"
```

Push back

`push_back()` aggiunge un nuovo elemento alla fine del vector

```
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars.push_back("Tesla");
cars.push_back("VW");
cars.push_back("Mitsubishi");
cars.push_back("Mini");
```

Pop back

`pop_back()` rimuove un elemento alla fine del vector

```
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars.pop_back();
```

Gli elementi vengono tendenzialmente aggiunti e tolti dalla fine del vector, esistono altre strutture dati che permettono manipolazioni ulteriori, ma non sono nello scopo del corso.

Rimozione in qualsiasi posto

```cpp
// Online C++ compiler to run C++ program online
#include <iostream>
#include <vector>

using namespace std;


int main() {
    // Write C++ code here
    vector<int> v={1,2,3};

    v.erase(v.begin() + 1);
    cout << v.at(1);

    return 0;
}
```

Size

```
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
cout << cars.size();  // Outputs 4
```

Loop di un vector

```
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (int i = 0; i < cars.size(); i++) {
  cout << cars[i] << "\n";
}

for (string car : cars) {
  cout << car << "\n";
}
```

Variabili Complesse

Array/Vector Multidimensionali

Array multidimensionali

```
int matrix[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
    };
```

Vector Multidimensionali

```
// Method 1: Empty 2D vector
vector<vector<int>> v1;

// Method 2: Predefined size with default values
vector<vector<int>> v2(3, vector<int>(4, 0)); // 3 rows,

// Method 3: Initialize with specific values
vector<vector<int>> v3 = {
{1, 2, 3},
{4, 5, 6},
{7, 8, 9}
};

// Accessing and printing elements
for (const auto& row : v3) {
for (const auto& elem : row) {
cout << elem << " ";
```

Variabili Complesse

String

Stringhe

Il tipo `string` non è un tipo **built-in**, ma si comporta come tale nel suo utilizzo base. Serve per contenere una sequenza di caratteri (testo).

```
// Include the string library
#include <string>
// Create a string variable
string greeting = "Hello";
// Output string value
cout << greeting;
```

Per utilizzare le stringhe è necessario il `#include <string>`. Alcune dipende di `<string>` sono gia dichiarate dentro `<iostream>` ma è comunque **meglio** mettere `<string>`.

`string` è un tipo definito nella standard library, senza `using namespace std;` è quindi necessario usare `std::string`.

Inizializzazione di una string

```
string str1{"Pippo"};
string str2="Pippo";
```

Concatenazione

```
string firstName = "John";
string lastName = "Doe";
string fullName = firstName + " " + lastName;
cout << fullName; //John Doe

string firstName = "John ";
string lastName = "Doe";
string fullName = firstName.append(lastName);
cout << fullName;
```

Attenzione con la somma

```
int x = 10;
int y = 20;
int z = x + y;       // z will be 30 (an integer)

string x = "10";
string y = "20";
string z = x + y;    // z will be 1020 (a string)
```

Lunghezza di una stringa

```
string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
cout << "The length of the txt string is: " << txt.length();
```

Accesso ad una stringa

Le stringhe possono essere viste come un array di caratteri, infatti è possibile accederci posizionalmente

```
string myString = "Hello";
cout << myString[0];
```

```
string myString = "Hello";
myString[0] = 'J';
cout << myString; //Jello
```

.at()

```
string myString = "Hello";
cout << myString; // Outputs Hello

// First character
cout << myString.at(0);

// Second character
cout << myString.at(1);

// Last character
cout << myString.at(myString.length() - 1);

myString.at(0) = 'J';
cout << myString;  // Outputs Jello
```

Caratteri speciali

```
//Qui c'è un errore
string txt = "We are the so-called "Vikings" from the north.";
```

Per poter utilizzare `"` all'interno di una stringa, bisogna utilizzare il caraterre di **escape** `\`. Esistono altri caratteri speciali come `\n`, `\t` o `\'`.

Leggere una stringa

```
string firstName;
cout << "Type your first name: ";
cin >> firstName; // get user input from the keyboard
cout << "Your name is: " << firstName;

// >> John Doe
// << John

string fullName;
cout << "Type your full name: ";
getline (cin, fullName);
cout << "Your name is: " << fullName;

// >> John Doe
// << John Doe
```

Il problema di **`cin`** è che si ferma al primo spazio, con **`getline`** il programma prendere l'intera linea.

Metodi delle stringhe

Si trovano molti metodi già implementati che permettono di fare molto con le stringhe, un esempio:

```
//ritorna la posizione di una sottostringa dentro una stringa
//se non la trova, ritorna string::npos
    string s = "pippo";
    string sub1 = "pp";
    string sub2 = "aa";
    cout << s.find(sub1) << endl; //2
    cout << s.find(sub2) << endl; //npos
```

Ma ne esistono altri, basta cercare!

Sezione attuale

Loop

I loop sono istruzioni che permettono di eseguire un blocco di codice n-volte.

- Se n è noto a compile time o a run time, si usa un `for-loop`
- Se n non è noto, ma si deve eseguire un blocco di codice finchè una determinata condizione è vera (o falsa), si usa un `while-loop`

Cicli

While Loop

While

Il ciclo `while` permette di eseguire un blocco di codice racchiuso tra {} finchè una condizione rimane vera (`true`)

```
while (condition) {
  // code block to be executed
}
```

Esempio con while

```cpp
#include <iostream>

int main(){
    int i;
    std::cout << "Inserisci un numero: ";
    std::cin >> i;
    while(i % 2 == 0){
        std::cout << "Hai inserito un numero pari" << std::endl;
        std::cout << "Inserisci un numero: ";
        std::cin >> i;
    }
    std::cout << "Hai inserito un numero dispari, terminazione programma" << std::endl;

    return 0;
}
```

Countdown con while

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

int main(){
    int countdown = 3;

    while (countdown > 0) {
      cout << countdown << "\n";
      countdown--;
    }
    cout << "Happy New Year!!\n";

    return 0;
}
```

Do-while

Il blocco `do-while` è una variante del `while`. La differenza è che il blocco `do-while` esegue almeno una volta il blocco di codice prima di controllare la condizione.

```
do{
    //code block to be exectued
} while(condition);
```

Input utente

```
#include <iostream>
using namespace std;
int main(){
    int number;
    do {
      cout << "Enter a positive number: ";
      cin >> number;
    } while (number <= 0);
    cout << "Hai inserito un numero positivo";
    return 0;
}
```

Esempio

```cpp
#include <iostream>

int main(){
    int i;
    do {
        std::cout << "Inserisci un numero: ";
        std::cin >> i;
        if(i%2 == 0){
            std::cout << "Hai inserito un numero pari" << std::endl;
        }
        else{
            std::cout << "Hai inserito un numero dispari, terminazione programma" << std::endl;
        }
    } while(i % 2 == 0)

    return 0;
}
```

Cicli

For Loop

For

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

- **statement 1**: Istruzione eseguita una sola volta, prima dell'esecuzione del blocco di codice
- **statement 2**: Definisce la condizione di esecuzione del blocco
- **statement 3**: Istruzione eseguita ogni volta dopo l'esecuzione del blocco

Esempio

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

int main(){
    for (int i=0; i<5; i++) {
      cout << i << endl;
    }
}
```

- `int i=0`: Si inizializza il contatore, una e una sola volta
- `i<5`: Si definisce la condizione per cui il blocco viene eseguito
- `i++`: Si incrementa il contatore

Ciclo annidato

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

int main(){
    // Outer loop
    for (int i = 1; i <= 2; ++i) {
      cout << "Outer: " << i << "\n"; // Executes 2 times

      // Inner loop
      for (int j = 1; j <= 3; ++j) {
        cout << " Inner: " << j << "\n"; // Executes 6 times (2 * 3)
      }
    }
}
```

Cicli

Foreach Loop

Foreach

Il `Foreach` è utilizzato per ciclare all'interno di una variabile

```
for (type variableName : arrayName) {
  // code block to be executed
}
```

Esempio di foreach

```
#include <iostream>
using namespace std;
int main(){
    int myNumbers[5] = {10, 20, 30, 40, 50};
    for (int i : myNumbers) {
      cout << i << "\n";
    }
}
```

Break

L'istruzione di **break** serve per poter stoppare prematuramente un esecuzione di un ciclo.

```
#include <iostream>
using namespace std;
int main(){
    int myNumbers[5] = {10, 20, 30, 40, 50};
    int length = sizeof(myNumbers) / sizeof(myNumbers[0]);
    for(int i=0; i<length; i++) {
        if(i==2){
            break;
        }
        cout << myNumbers[i] << endl;
    }
}
```

Output:

```
10
20
```

Continue

L'istruzione di **`Continue`** serve per poter saltare uno step del ciclo.

```
#include <iostream>
using namespace std;
int main(){
    int myNumbers[5] = {10, 20, 30, 40, 50};
    int length = sizeof(myNumbers) / sizeof(myNumbers[0]);
    for(int i=0; i<length; i++) {
        if(i==2){
            continue;
        }
        cout << myNumbers[i] << endl;
    }
}
```

Output:

```
10
20
40
50
```

Sezione attuale

Cosa è il debugger

Cos'è un Debugger?

- Il debugger è uno strumento che permette di eseguire un programma passo per passo.
- Consente di osservare lo stato delle variabili e il flusso di esecuzione.
- È fondamentale per individuare e correggere errori (bug) nel codice.

Sezione attuale

Esercizio 1

Realizzare un programma in C++ che permetta all'utente di inserire un numero positivo ed in seguito stampi a video tutti i numeri da 1 a quel numero.

Esercizio 2

Realizzare un programma in C++ che permetta all'utente di inserire un numero n (maggiore di 0) ed effettua la somma dei primi n numeri naturali.

Esercizio 3

Realizzare un programma che permetta all'utente di inserire un numero positivo ed in seguito riesca a comprendere se tale numero sia pari o dispari utilizzando solamente l'operazione di sottrazione.

Suggerimento

Continuare a sottrarre due, se rimane 0 è pari, se rimane 1 è dispari

Esercizio 4

Realizzare un programma che chieda all'utente di inserire 10 numeri e stampi quanti di essi sono positivi.

Esercizio 5

Realizzare un programma che chieda all'utente di inserire un numero positivo e stampi la sua tabellina fino a 10.

Esercizio 6

Realizzare un programma che chieda all'utente di inserire una parola e stampi ciascun carattere su una riga diversa.

Esercizio 7

Realizzare un programma che chieda all'utente di inserire un numero positivo e calcoli il fattoriale di quel numero utilizzando un ciclo `for`.

Fattoriale

Il fattoriale di un numero n è uguale a $n \cdot (n-1) \cdot (n-2) \ldots 1$
Esempio, il fattoriale di 10! è $10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 3,628,800$

Esercizio 8

Realizzare un programma che chieda all'utente di inserire una stringa e ne calcoli la lunghezza senza usare funzioni predefinite.

Esercizio 9

Realizzare un programma che chieda all'utente di inserire un numero positivo e ne stampi la rappresentazione in binario (senza usare funzioni di libreria).

Esercizio 10

Realizzare un programma che chieda all'utente di inserire una parola e verifichi se è palindroma.

Esercizio 11

Realizzare un programma che chieda all'utente di inserire 5 numeri e li memorizzi in un array, quindi stampi il valore massimo e il valore minimo.

Esercizio 12

Realizzare un programma che chieda all'utente di inserire un numero **n** e poi **n** valori interi. Il programma deve stampare la somma dei soli valori pari.

Esercizio 13

Un professore deve fare la media dei voti dei suoi studenti. Realizzare un programma che permetta al docente di inserire il numero `numVoti` di voti di cui vuole calcolare la media. Successivamente il docente deve inserire i voti (nell'intervallo [0, 10]).

Esercizio 14

Realizzare un programma che chieda all'utente di inserire una stringa e stampi la stringa al contrario.

Esercizio 15

Realizzare un programma che chieda all'utente di inserire una lista di numeri positivi, terminando l'inserimento quando viene inserito un numero negativo. Stampare la media dei numeri inseriti.

Esercizio 16

Realizzare un programma che chieda all'utente di inserire due array di 5 elementi ciascuno e crei un terzo array contenente la somma elemento per elemento.

Esercizio 17

Realizzare un programma che chieda all'utente di inserire un testo e conti quante vocali e quante consonanti sono presenti.

Esercizio 18

Scrivere un programma che stampa tutti gli elementi di un vettore che sono strettamente minori dei loro vicini.

```
Esempio: 1 2 5 0 3 1 7
<< 0 1
```

Esercizio 19

Scrivere un programma che data una stringa stampa una nuova stringa con le tutte le parole che iniziano con le maiuscole

```
<< red green black white Pink
>> Red Green Black White Pink
```

Esercizio 20

Scrivere un programma in C++ che verifichi che tutte le lettere presenti nella seconda stringa compaiano anche nella prima stringa. Restituire `true` se la condizione è soddisfatta, altrimenti `false`.

```
<< Python Py
>> true
```

Esercizio 21

Scrivere un programma in C++ che, dato un vettore di stringhe, ritorni solo quelle stringhe che contengono almeno un numero. Restituire un vettore vuoto se nessuna stringa contiene numeri.

```
<< red green23 1black white
>> green23 1black
```

Esercizio 22

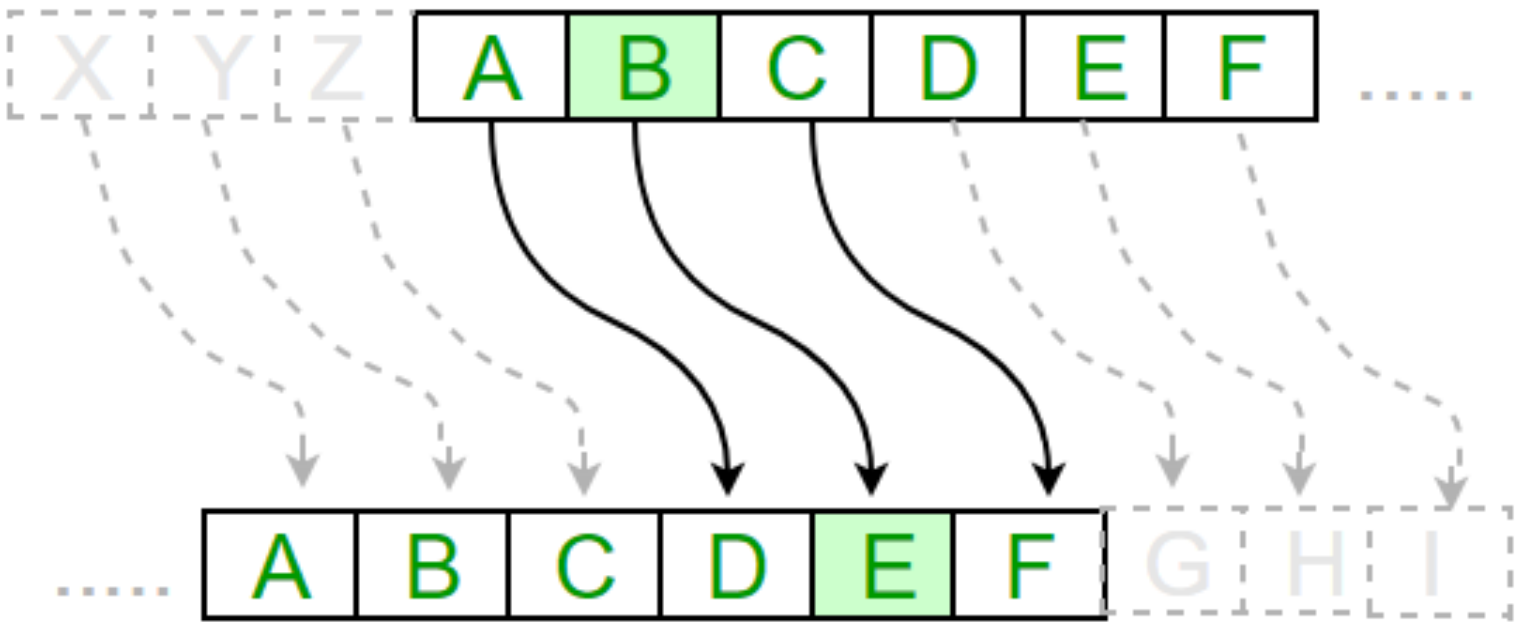
Realizzare un programma che chieda all'utente di inserire una frase e verifichi se è un pangramma (cioè contiene tutte le lettere dell'alfabeto almeno una volta).

Esercizio 23

Implementare il cifrario di atbash. Il cifrario di Atbash inverte l'ordine delle lettere dell'alfabeto. Data la stringa "ciao" essa diventa "xrzl".

Esercizio 24

Implementare il cifrario di cesare. Il cifrario di cesare sposta ogni lettere di 3 posizioni.



Data la stringa ”ciao” essa diventa ”fldr”

Esercizio 25

Implementare il cifrario di cesare che permette di decidere di quante posizioni spostare, sia in avanti che indietro.

Esercizio 26

L'esercizio prevede l'implementazione di un ATM con le seguenti operazioni:

- Deposito di soldi
- Ritiro di soldi
- Stampare saldo medio
- Exit

Utilizzare un `while(true)` per simulare la continuità di esecuzione del programma e chiedere all'utente di inserire un numero che modellizzi le singole operazioni

Esercizio 27

Implementare un programma che permette a due giocatori di giocare a tris, il programma deve:

1. Chiedere al giocatore ”X” in quale cella vuole giocare (numero da 0 a 8)
2. Mettere la ”X” verificando se la posizione è valida, altrimenti richiedere al giocatore una posizione
3. Verificare se qualcuno ha vinto o il gioco è finito, se si, si esce dal programma
4. Ripetere dal punto 1 ma con il giocatore ”O”

```
012
345
678
```