Guida alla programmazione in C++

`Questa guida è per C++ dalla versione C++11 in poi utilizzando come IDE Visual Studio 2022 e come compilatore MSVC (il compilatore di Visual Studio)`

Livello Principiante

I commenti

I commenti sono delle parti di testo inserite nel codice il cui unico scopo è quello di fornire informazioni sul codice, vengono ignorati dal compilatore e perciò non influiscono sull'esecuzione del programma.

Per scrivere un commento si può precedere la frase dal simbolo `//` oppure racchiudere la frase tra `\*` e `*\`.

Esempio:

```
// questo è un commento su una riga
/*
    questo è un commento
    su più righe
*/
/* questo è un altro commento */
```

Le basi del linguaggio C++

Hello World

Ecco un semplice programma che scrive “Hello World!”:

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!";

    return 0;
}
```

Spiegazione

`iostream` (input output stream) è un file esterno, che è necessario includere per usare `std::cout`.

`main` rappresenta il programma, e contiene ciò che deve essere eseguito, `int` segnala che il tipo restituito dalla funzione è un numero intero.

`return 0` termina il programma con codice 0 che significa che non si sono verificati errori

`std` è uno spazio di nomi che contiene l'oggetto `cout`, si scrive `std::` prima di `cout` per indicare che `cout` appartiene allo spazio di nomi std (standard).

`std::cout` è utilizzato per scrivere nella console, in questo caso `Hello, World!`, le cose da scrivere sono concatenate con l'operatore `<<`.

Il punto e virgola dopo `Hello, World!` indica che l'istruzione è terminata, infatti il compilatore distingue le istruzioni grazie al punto e virgola e non al carattere di nuova linea.

`#include <iostream>` non ha un punto e virgola alla fine perché è una direttiva, si possono riconoscere le direttive perché iniziano con `#`

using namespace std

Esiste un modo per evitare di scrivere `std::` prima di ogni funzione\oggetto\classe\struttura dello spazio di nomi standard:

```
using namespace std;
```

In questo caso il programma diventa

```
#include <iostream>
using namespace std;
int main()

{
    cout << "Hello, World!";
    return 0;
}
```

In questa guida non useremo `using namespace std` perché così è più chiaro

un'alternativa preferibile a `using namespace std` potrebbe essere questa

```
using std::cout;
```

oppure

```
using std::cout, std::cin;
```

chiaramente scrivendo ciò la regola vale SOLO per gli oggetti\funzioni\strutture\classi riportati dopo `using`

L'output (o stampa a video)

In C++ esistono quattro oggetti principali per eseguire l'output: **`cout`**, **`cerr`**, **`wcout`**, **`wcerr`**.
(`cout` = c output, `cerr` = c error, `wcout` = wide c output, `wcerr` = wide c error)

`cout` e `wcout` sono usati per l'output normale mentre `cerr` e `wcerr` sono usati per i messaggi di errore,

`cout` e `cerr` sono usati con le stringhe normali e `wcout` e `wcerr` con stringhe larghe, vale a dire stringhe che supporatano una maggiore quantità di caratteri (come ad esempio lettere accentate).

Esempio:

```
#include <iostream>
int main()
{
    std::cout <<  "questa e' una stringa normale"        << std::endl;
    std::cerr <<  "questa e' una stringa di errore"      << std::endl;

    setlocale(0, "");

    std::wcout << L"questa è una stringa wide"           << std::endl;
    std::wcerr << L"questa è una stringa wide di errore" << std::endl;

    return 0;
}
```

Notare l'aggiunta di `L` prima delle stringhe wide, in generale bisogna aggiungere **`setlocale(0, "");`** solo una volta prima di usare stringhe wide, ed è buona pratica non mescolare cout con `wcout` o `cerr` con `wcerr`.

`std::endl` (end of line) serve per andare a capo.

Le sequenze di escape

Le sequenze di escape rappresentano dei caratteri speciali non stampabili:

- `\n`: aggiunge una nuova riga (va a capo)
- `\r`: sposta il cursore all'inizio della riga
- `\b`: carattere backspace
- `\a`: suono di errore
- `\t`: tab
- `\`: carattere backslash
- `\"`: virgolette (per distinguerle da quelle che delimitano la stringa)
- `\'`: apice (viene è usato per delimitare un singolo carattere invece di una stringa)
- `\0`: carattere nullo (non uno spazio)

Ha particolare importanza `\n`, che è un'alternativa in genere preferibile a `std::endl`

La tabella ASCII

La tabella ASCII assegna un numero a ogni carattere:

Byte	Cod.	Char	Byte	Cod.	Char	Byte	Cod.	Char	Byte	Cod.	Char
00000000	0	Null	00100000	32	Spc	01000000	64	@	01100000	96	`
00000001	1	Start of heading	00100001	33	!	01000001	65	A	01100001	97	a
00000010	2	Start of text	00100010	34	"	01000010	66	B	01100010	98	b
00000011	3	End of text	00100011	35	#	01000011	67	C	01100011	99	c
00000100	4	End of transmit	00100100	36	$	01000100	68	D	01100100	100	d
00000101	5	Enquiry	00100101	37	%	01000101	69	E	01100101	101	e
00000110	6	Acknowledge	00100110	38	&	01000110	70	F	01100110	102	f
00000111	7	Audible bell	00100111	39	'	01000111	71	G	01100111	103	g
00001000	8	Backspace	00101000	40	(	01001000	72	H	01101000	104	h
00001001	9	Horizontal tab	00101001	41	)	01001001	73	I	01101001	105	i
00001010	10	Line feed	00101010	42	*	01001010	74	J	01101010	106	j
00001011	11	Vertical tab	00101011	43	+	01001011	75	K	01101011	107	k
00001100	12	Form Feed	00101100	44	,	01001100	76	L	01101100	108	l
00001101	13	Carriage return	00101101	45	-	01001101	77	M	01101101	109	m
00001110	14	Shift out	00101110	46	.	01001110	78	N	01101110	110	n
00001111	15	Shift in	00101111	47	/	01001111	79	O	01101111	111	o
00010000	16	Data link escape	00110000	48	0	01010000	80	P	01110000	112	p
00010001	17	Device control 1	00110001	49	1	01010001	81	Q	01110001	113	q
00010010	18	Device control 2	00110010	50	2	01010010	82	R	01110010	114	r
00010011	19	Device control 3	00110011	51	3	01010011	83	S	01110011	115	s
00010100	20	Device control 4	00110100	52	4	01010100	84	T	01110100	116	t
00010101	21	Neg. acknowledge	00110101	53	5	01010101	85	U	01110101	117	u
00010110	22	Synchronous idle	00110110	54	6	01010110	86	V	01110110	118	v
00010111	23	End trans. block	00110111	55	7	01010111	87	W	01110111	119	w
00011000	24	Cancel	00111000	56	8	01011000	88	X	01111000	120	x
00011001	25	End of medium	00111001	57	9	01011001	89	Y	01111001	121	y
00011010	26	Substitution	00111010	58	:	01011010	90	Z	01111010	122	z
00011011	27	Escape	00111011	59	;	01011011	91	[	01111011	123	{
00011100	28	File separator	00111100	60	<	01011100	92	\	01111100	124	|
00011101	29	Group separator	00111101	61	=	01011101	93	]	01111101	125	}
00011110	30	Record Separator	00111110	62	>	01011110	94	^	01111110	126	~
00011111	31	Unit separator	00111111	63	?	01011111	95	_	01111111	127	Del

---

I datatype

I datatype sono i tipi tra cui bisogna scegliere quando si dichiara una variabile, esistono dei datatype primari, forniti dal linguaggio C++, e datatype secondari che possono essere definiti nei file header esterni oppure direttamente nel codice.

I datatype primari

Tipi interi:

- `short`: un intero compreso tra -32.768 e 32.767
- `int`: un intero compreso tra -2.147.483.648 e 2.147.483.647
- `long` (o `__int32`): a seconda del sistema può comportarsi come `int` o `long long`
- `long long` (o `__int64`): un intero compreso tra -2^63 e 2^63 - 1

Tipi decimali:

- `float`: un numero decimale con precisione di 6 o 7 cifre decimali
- `double`: un numero decimale con precisione di 15-16 cifre decimali

- `long double`: varia, ma in genere ha una precisione decimale di 18-19 cifre

Altro:

- `char`: un carattere
- `bool`: può essere solo vero (`true` o 1) oppure falso (`false` o 0)

E' possibile aggiungere la parola chiave `unsigned` prima del tipo, in questo modo il datatype non potrà mai contenere valori negativi, ma può contenere valori positivi con un range doppio

Esempio:

`__int64` va da **-9.223.372.036.854.775.808** a **9.223.372.036.854.775.807**,
ma `unsigned __int64` va da **0** a **18.446.744.073.709.551.615**.

I `typedef`

Per definire un datatype secondario è possibile utilizzare un `typedef`:
typedef <*datatype*> <*nome*>;

Esempio:

```
typedef unsigned long long size_t;
```

Qui si definisce `size_t` come `unsigned long long`.

Esiste un altro modo per definire un datatype secondario con la parola chiave `using`:
using <*nome*> = <*datatype*>;

L'esempio di prima diventa

```
using size_t = unsigned long long;
```

Le variabili e le costanti

Che cosa sono

In C++ le variabili vengono dichiarate con questa sintassi:
<datatype> <nome>;

In questo modo si dichiara una variabile che però non ha un valore, che si può assegnare con l'inizializzazione:
<datatype> <nome> = <espressione>;

Il nome di una qualsiasi cosa (non solo una variabile) può contenere solo lettere, numeri o underscore (_) e non può iniziare con un numero.
Il C++ è un linguaggio **case sensitive** cioè esiste una differenza tra le lettere minuscole e quelle maiuscole.

Per dichiarare una costante bisogna aggiungere la parola chiave `const` all'inizio ed è obbligatorio assegnare un valore:
const <*datatype*> <*nome*> = <*espressione*>;

Le variabili possono cambiare il loro valore durante l'esecuzione del programma, al contrario delle constanti.

Inizializzazione

Si può usare la parola chiave `auto` al posto del datatype, dove il compilatore proverà a dedurre il tipo dall'espressione, perciò è obbligatorio assegnare un valore, ad esempio è errato scrivere

```
auto Variable;
const double Pi;
```

Ma è corretto scrivere

```
auto Variable = 1;
const double Pi = 3.1415926535;
int NewVariable = Variable;
long lines;
```

Se due o più variabili sono dichiarate con lo stesso datatype si può usare una virgola per separarle e scrivere una sola volta il datatype:

```
int A = 1, B = 2, _i, __i;
```

Esistono altri due modi per inizializzare le variabili, l'inizializzazione per copia e uniforme, ma il datatype dell'espressione deve essere uguale a quello della variabile:

Inizializzazione per copia

```
int A(1), B(2);
```

Inizializzazione uniforme

```
int A{ 1 }, B{ 2 };
```

Nell'inizializzazione uniforme si può omettere il valore tra le parentesi:

```
int x{}; // stessa cosa di int x{ 0 };
```

Cambiare il valore

Per cambiare il valore di una variabile si può usare l'operatore `=`.
Esempio:

```
x = 1; // vuol dire "x è impostato a 1" non "x è uguale a 1"
```

Vediamo come è possibile scambiare il valore di due variabili:

```
int main()
{
    int x = 5, y = 10;

    auto temp = x; // x = 5   y = 10  temp = 5
    x = y;         // x = 10  y = 10  temp = 5
    y = temp;      // x = 10  y = 5   temp = 5

    // x = 10   y = 5

    return 0;
}
```

L'input utente

In C++ l'input viene eseguito con `std::cin`, necessario includere `<iostream>`:

```
#include <iostream>

int main()
{
    int Number1, Number2;
    std::cin >> Number1 >> Number2;
}
```

In questo modo l'utente inserisce da console due valori per le variabili Number1 e Number2, tuttavia l'input si blocca se l'utente inserisce dei caratteri, ma per risolvere questo si può eseguire l'input con una stringa e poi convertirla in numero (vedi stringhe).

E' buona pratica inserire sempre un output prima di ogni input per far capire all'utente cosa deve fare:

```
#include <iostream>

int main()
{
```

```
    int Number1;
    std::cout << "inserisci un numero\n";
    std::cin >> Number1;
    std::cout << "hai inserito: " << Number1;
    return 0;
}
```

L'ambito delle variabili

In C++ una variabile viene creata con una dichiarazione e distrutta quando esce dal proprio ambito, che è definito dalle parentesi graffe, una variabile dichiarata fuori da una funzione si dice **globale** mentre una variabile dichiarata in una funzione si dice **locale**.

Se due variabili, una locale e una globale hanno lo stesso nome ma ambiti diversi, è possibile utilizzare l'**operatore di risoluzione dell'ambito** `::` per accedere a quella globale, ciò funziona anche con le funzioni, o con una funzione e una variabile:

```
#include <iostream>
const double Variable = 10.2; // variabile globale

int main()
{
    int Variable = 7;
    std::cout << Variable << '\n'; // output = 7
    std::cout << ::Variable << '\n'; // output = 10.2

    return 0;
}
```

In questo caso abbiamo due variabili con lo stesso nome, una locale e una globale, possiamo accedere a quella locale con `variable` e a quella globale con `::variable`.

Due variabili non possono avere lo stesso nome all'interno dello stesso ambito.

Gli operatori matematici

In C++ ci sono 5 operatori matematici **primari (`+`, `-`, `*`, `/`, `%`)**, 5 operatori matematici **composti (`+=`, `-=`, `*=`, `/=`, `%=`)** e 2 operatori **unari (`++` e `--`)**.

Gli operatori matematici primari

Gli operatori **(`+`, `-`, `*`, `/`)** indicano rispettivamente addizione, sottrazione, moltiplicazione e divisione.
L'operatore `%` indica il resto della divisione intera.

Gli operatori `*` `/` e `%` hanno la precedenza sugli operatori `+` e `-`.

Esempio di utilizzo:

```
#include <iostream>

int main()
{
    int A = 10, B = 5;

    std::cout << A + B << '\n'; // output = 15
    std::cout << A - B << '\n'; // output = 5
    std::cout << A * B << '\n'; // output = 50
    std::cout << A / B << '\n'; // output = 2
    std::cout << A % B << '\n'; // output = 0

    return 0;
}
```

Gli operatori composti e unari abbreviano alcune espressioni:

Gli operatori matematici composti

```
    Var += i;   // Var = Var + i
    Var -= i;   // Var = Var - i
    Var *= i;   // Var = Var * i
    Var /= i;   // Var = Var / i
    Var %= i;   // Var = Var % i
    Var++;      // Var = Var + 1
    Var--;      // Var = Var - 1
    ++Var;      // Var = Var + 1
    --Var;      // Var = Var - 1
```

Gli operatori unari

Tuttavia esiste una leggera differenza tra `++Variabile` e `Variabile++` e tra `--Variabile` e `Variabile--`:

```
    int Var, i;

    // prima si incrementa i, quindi si assegna il risultato a Var
    // i = 3 --> i = 4 --> Var = 4
    i = 3;
    Var = ++i;

    // prima si assegna il risultato a Var, quindi si incrementa i
    // i = 3 --> Var = 3 --> i = 4
    i = 3;
    Var = i++;
```

La conversione tra datatype

Alcune volte può essere necessario effettuare la conversione tra due datatype, in C++ esistono vari modi per fare ciò, ma qui ne vedremo solo due: il **type cast** e lo **static cast**.

Il type cast

Il type cast può effettuare qualsiasi tipo di conversione.
(<*datatype*>)<*espressione*>
oppure
*<datatype>(<espressione>)***

Esempio: divisione decimale tra due interi

```
#include <iostream>

int main()
{
    const int A = 7, B = 5;
    std::cout << A / B << '\n'; // output = 1
    std::cout << double(A) / B; // output = 1.4

    return 0;
}
```

Nel primo output viene visualizzato 1 (un intero) perché entrambi i numeri sono interi; ma nel secondo, A viene convertito in double, e quindi anche il risultato sarà double: 1.4.

Lo static cast

E' preferibile utilizzare lo **static cast** quando possibile, perché è più sicuro e consente solo conversioni ben definite.
static_cast< <*datatype*> >(<*espressione*>);

```
#include <iostream>

int main()
{
    const int A = 7, B = 5;
    std::cout << A / B << '\n';
    std::cout << static_cast<double>(A) / B;

    return 0;
}
```

I concetti di base delle stringhe

In C++ una stringa è un datatype secondario per una variabile che contiene del testo, esistono due classi per questo dette `std::string` e `std::wstring`, per usare le stringhe bisogna includere `<string>`

Output

Esempio:

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

int main()
{
    std::string str = "questa e' una stringa";
    std::wstring wstr = L"questa è una stringa wide";

    std::cout << str << '\n';

    setlocale(0, "");
    std::wcout << wstr << '\n';
}
```

Se si vuole una stringa di caratteri uguali e lunghezza finita si può fare così:

```
std::string str(5, ',');   // stessa cosa di std::string str = ",,,,,";
std::string wstr(9, L','); // stessa cosa di std::string str = L",,,,,,,,,";
```

Input

Si esegue l'input di una stringa con `std::cin` e quello di una stringa wide con `std::wcin`:

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

int main()
{
    setlocale(0, "");

    std::string str;
    std::wstring wstr;

    std::cout << "inserisci due stringhe\n";
    std::cin >> str;
    std::wcin >> wstr;

    std::cout  << str  << '\n';
```

```
    std::wcout << wstr << '\n';
}
```

Tuttavia c'è un problema: `std::cin` e `std::wcin` fermano l'input non appena leggono un carattere di spazio, ma questo si risolve con la funzione `std::getline`:

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

int main()
{
    setlocale(0, "");

    std::string str;
    std::wstring wstr;

    std::cout << "inserisci due stringhe\n";
    std::getline(std::cin, str);
    std::getline(std::wcin, wstr);

    std::cout  << str  << '\n';
    std::wcout << wstr << '\n';
}
```

Input senza `cin`

Esiste un altro modo per eseguire l'input di un carattere, che è strettamente legato alle stringhe, con il file header `<conio.h>`:

- `_getch()` legge un carattere dalla tastiera senza scriverlo sullo schermo (al contrario di `_getche()`), e non un carattere wide al contrario di `_getwch()`:

```
#include <conio.h>

int main()
{
    _getch(); // ferma il programma fino a quando un carattere viene premuto
    return 0;
}
```

```
#include <conio.h>

int main()
{
    char c = _getch(); // legge un carattere e lo assegna a c
```

```
    return 0;
}
```

Molto importante anche la funzione `_kbhit()` (non `_kbhit`), che resituisce `true` se un carattere è stato premuto ed è in attesa di essere letto da una funzione di input:

```
#include <conio.h>
#include <iostream>

int main()
{
    if (_kbhit()) {
        char c = _getch();
        std::cout << "hai premuto " << c;
    }

    return 0;
}
```

Le istruzioni IF-ELSE

Che cos'è

Un'istruzione if-else è un modo per eseguire determinate parti di codice in base alla verità di una condizione:

if (*<condizione>*) {
<istruzioni da eseguire se la condizione è vera>
}
else {
<istruzioni da eseguire se la condizione è falsa>
}

Ecco un esempio di if-else, un programma per indicare il segno di un numero:

```
#include <iostream>

int main()
{
    long long Number;
    std::cin >> Number;

    if (Number > 0) {
        std::cout << "il numero e' positivo\n";
    }
    else if (Number < 0)
    {
        std::cout << "il numero e' negativo\n";
```

```
    }
    else std::cout << "il numero e' 0\n";

    return 0;
}
```

Le parentesi graffe

Qui possiamo vedere i 3 principali stili di un if-else quanto a parentesi graffe, che valgono anche per i **cicli**, che saranno descritti in seguito:

- con la parentesi graffa aperta davanti alla condizione
- con le parentesi graffe a capo
- senza parentesi graffe, tuttavia deve essere presente una sola istruzione

E' anche possibile annidare due istruzioni if in questo modo:

```
    if (Number > 0)
        if (Number % 2 == 0)
        {
            Number++;
            std::cout << Number;
        }
```

Qui vediamo che il primo if non ha le parentesi graffe, questo perché al suo interno c'è un solo if, nonostante le due istruzioni all'interno di quest'ultimo if

Gli operatori

Negli if vengono usati gli **operatori di confronto** e gli **operatori logici**:

Operatori di confronto:

- `<`: minore di,
- `<=`: minore o uguale di,
- `==`: uguale a,
- `>=`: maggiore o uguale di,
- `>`: maggiore di,
- `!=`: diverso da.

Operatori logici:

- `&&` o `and`: AND logico (il risultato è vero se e solo se entrambi gli input sono veri)
- `||` o `or`: OR logico (il risultato è vero se uno dei due input è vero)
- `!` o `not`: NOT logico (il risultato è vero se e solo se l'input è falso) per usare lo XOR logico si può usare l'operatore `!=`

Grazie agli operatori logici i due if precedenti si possono compattare così:

```
    if (Number > 0 && Number % 2 == 0)
    {
        Number++;
        std::cout << Number;
    }
```

Gli IF-ELSE e le eccezioni

Tuttavia esistono dei casi in cui è meglio usare due if invece di uno solo, come in questo esempio:

```
#include <iostream>

int main()
{
    int A, B;
    std::cin >> A >> B;

    if (B != 0) if ((A / B) % 2 == 1)
    {
        A--;
        B++;
    }

    return 0;
}
```

L'espressione `(A / B) % 2 == 1` non può essere eseguita se B è 0, e solleverebbe un'**eccezione** `std::invalid_argument`, per prevenire ciò bisogna controllare che B sia diverso da 0 PRIMA dell'elaborazione dell'espressione e non nella stessa istruzione if.

Gli IF-ELSE e l'assegnazione

Nella condizione degli if si può anche assegnare un valore a una variabile, la variabile non è nulla:

```
#include <iostream>

int main()
{
    int x{};

    if (x = 5)  // x viene assegnato a 5, poi si verifica se non è nullo
        std::cout << "x è stato assegnato a 5\n";

    return 0;
}
```

Per questo bisogna fare attenzione a non confondere gli operatori == e = perché non ci sarà nessun errore di compilazione in caso di assegnazione accidentale dentro un if. Per risolvere ciò può aiutare scrivere

```
    if (1 == x) // if (1 = x) è un errore: non si può assegnare a un numero
```

Invece di

```
    if (x == 1) // if (x = 1) non genera l'errore
```

L'operatore ternario

In C++ esiste un'alternativa rispetto alle istruzioni if-else detta operatore ternario.
<condizione> ? <se vero> : <se falso>;

Senza assegnazione

Esempio: programma per trovare l'inverso di un numero

```
#include <iostream>

int main()
{
    long long number;
    std::cin >> number;

    number == 0 ?
        std::cout << "non e' possibile dividere per 0\n" :
        std::cout << "l'inverso e' " << 1.0 / number << '\n';

    return 0;
}
```

Con l'assegnazione

E' possibile utilizzare l'operatore ternario per assegnare un valore, qui viene riportato un programma per calcolare il massimo tra due numeri:

```
#include <iostream>

int main()
{
    int A, B;
    std::cout << "inserisci due numeri\n";
```

```
    std::cin >> A >> B;

    int max = A > B ? A : B;
    std::cout << "il massimo e' " << max;

    return 0;
}
```

Gli operatori bitwise

In C++ esistono degli operatori detti **bitwise** che eseguono delle operazioni logiche su ogni bit di uno o due numeri scritti in codice binario.

Supponiamo di avere due variabili A = 9 e B = 10
Riscrivendoli in binario otteniamo A = 1001 e B = 1010

Vediamo come si comporta ciascun operatore bitwise con A e B:

BITWISE AND
Si esegue l'AND logico su ogni bit rispettivamente:
`A & B` = `1001 & 1010` = 1000 = 8 (in base 10)

BITWISE OR
Si esegue l'OR logico su ogni bit rispettivamente:
`A | B` = `1001 | 1010` = 1011 = 11 (in base 10)

BITWISE NOT
Si cambia ogni bit di un numero:
`~A` = `~1001` = 0110 = 6 (in base 10)

BITWISE XOR
Si esegue l'XOR logico su ogni bit rispettivamente:
`A ^ B` = `1001 ^ 1010` = 0011 = 3 (in base 10)

LEFT SHIFT
Si spostano tutti i bit verso sinistra di tante posizioni quanto lo shift:
`A << 2` = `001001 << 2` = 100100 = 36 (in base 10)

RIGHT SHIFT Si spostano tutti i bit verso destra di tante posizioni quanto lo shift:
`B >> 1` = `1010 >> 1` = 0101 = 5 (in base 10)

Ovviamente esistono anche gli operatori `&=`, `|=`, `^=`, `<<=` **e** `>>=`:
A = A <*operatore*> B → A <*operatore*>= B

L'istruzione GOTO

In C++ esiste un modo per trasferire il controllo a una certa istruzione quando ci si trova a un certo punto del codice, per fare ciò si utilizza un `goto`.

Attenzione: utilizzare molti goto può rendere il codice poco leggibile.

Esempio di goto:

```
#include <iostream>

int main()
{
    size_t some_number{};
    std::cin >> some_number;

    if (some_number == 0) goto stop;
    else {
        some_number <<= 3;
        std::cout << "eseguito left shift 3 volte\n";
        std::cout << "il numero e' " << some_number;
        return 0;
    }

stop:
    std::cout << "il programma e' stato interrotto\n";
    return 1;
}
```

In questo caso se il numero è 0, il controllo viene trasferito al punto `stop` da cui termina il programma con codice di errore 1, chiaramente questo è un esempio di goto utilizzato in modo errato, ma era giusto per capire come funziona questa istruzione.

Se tra il goto e l'istruzione del goto, viene inizializzata una variabile, si otterrà un errore di compilazione perché se si trasferisce il controllo viene ignorata l'inizializzazione di quella variabile.

Il ciclo WHILE

Un ciclo while è una sezione di codice che viene ripetuta finché una condizione rimane vera:

while (<*condizione*>)
{
<istruzioni da ripetere>
}

Può essere utilizzato per creare del codice che si ripete all'infinito, come questo:

```
#include <iostream>

int main()
{
    while (true)
    {
```

```
        size_t some_number{};
        std::cin >> some_number;

        std::cout << "hai inserito il numero ";
        std::cout << some_number << '\n';
    }
}
```

Questo codice ripeterà all'infinito all'utente di inserire un numero, perché la condizione all'interno del ciclo è sempre vera.

Ecco un codice che calcola i primi numeri della sequenza di Fibonacci:

```
#include <iostream>

int main()
{
    long long x = 0, y = 1, i{};
    while (y >= 0) {
        std::cout << "numero di Fibonacci #" << i;
        std::cout << " = " << x << '\n';

        auto temp = x + y;
        x = y;
        y = temp;
    }
    std::cout << "numero di Fibonacci #" << i + 1;
    std::cout << " = " << x << '\n';
    std::cout << "si e' verificato un overflow\n";
    return 0;
}
```

Poiché la variabile y è intera, ha un limite, quando il limite viene superato si riparte a contare dal minimo e quindi la variabile diventa negativa, rendendo falsa la condizione del ciclo e di conseguenza si esce dal ciclo.

Il ciclo DO WHILE

Questo è un ciclo simile al while, con la differenza che qui si controlla se una condizione è vera dopo aver eseguito il codice e non prima come nel while.

do {
<istruzioni da ripetere>
} while (*<condizione>*);

Questo ciclo può essere usato per il controllo dell'input, dove si ripete l'input fino a quando è corretto, ecco un codice che esegue l'input controllato di un numero positivo:

```cpp
#include <iostream>

int main()
{
    int pos_number;
    do {
        std::cout << "inserisci un numero\n";
        std::cin >> pos_number;

        if (pos_number <= 0)
        {
            std::cout << "hai inserito un numero sbagliato!";
        }
    } while (pos_number <= 0);

    return 0;
}
```

Il ciclo FOR

Questo è il ciclo più importante tra quelli visti finora, permette di ripetere un blocco di codice finché una condizione è vera ed eseguendo una certa operazione su di una variabile a ogni iterazione.

for (<*inizializzazione variabile*>; <*condizione*>; <*operazione*>)
{
<istruzioni da ripetere>
}

Si può lasciare vuota qualsiasi informazione tra le tre presenti.
Ecco un semplice esempio di uso del for:

```cpp
#include <iostream>

int main()
{
    int low, high;
    do {
        std::cout << "inserire il limite inferiore\n";
        std::cin >> low;
        std::cout << "inserire il limite superiore\n";
        std::cin >> high;
    } while (low >= high);

    for (int i = low; i <= high; ++i) std::cout << i << '\n';
    return 0;
}
```

L'iterazione inizia con la variabile i inizializzata a `low`, ogni volta che l'iterazione è completata `i` aumenta di 1 e l'iterazione continua finché `i` è minore o uguale a `high`, a ogni iterazione si stampa il valore di `i`.

Vediamo adesso un esempio più complesso: il riempimento di una regione della console con asterischi, l'utente sceglie la posizione della regione e le sue dimensioni:

```
#include <iostream>

int main()
{
    // input dimensioni
    unsigned short X, Y, lenght, width;
    std::cout << "inserire la posizione del primo vertice\n";
    std::cin >> X >> Y;
    std::cout << "inserire la dimensione della regionw\n";
    std::cin >> lenght >> width;

    // riduzione delle dimensioni per evitare l'overflow
    X %= 50;
    Y %= 50;
    lenght %= 50;
    width %= 50;

    // spostamento in giù
    for (int i = 0; i < Y; ++i) std::cout << '\n';

    // output di ogni riga
    auto limitX(lenght + X);
    for (int i = 0; i < width; ++i) {

        // spostamento di lato
        int j{};
        for (; j < X; ++j) std::cout << ' ';

        // output asterischi
        for (; j < limitX; ++j) std::cout << '* ';

        // termine riga
        std::cout << '\n';
    }

    return 0;
}
```

Possiamo vedere i due for all'interno del for più grande, che usano la stessa variabile `j` (è la stessa perché dichiarata fuori dal for), come per dimostrare che non è necessario fornire tutte e 3 le informazioni del for.

Le parole chiave `break` e `continue`

In C++ l'utilizzo dei cicli è reso più semplice dalle parole chiave `break` e `continue`: un'istruzione `break` esce dal ciclo mentre `continue` salta il giro, vediamo come si possono utilizzare con questo esempio:

```
#include <iostream>

int main()
{
    for (double i = -1.0;; i += 1.35)
    {
        if (i > 10) i /= 10;
        if (int(i) == 0) continue;

        double number;
        std::cout << "inserisci un numero\n";
        std::cin >> number;

        if (number == i) continue;
        i += number;

        if (i > 100) {
            std::cout << "hai inserito un numero troppo grande\n";
            break;
        }
    }

    std::cout << "fine\n";
    return 0;
}
```

In questo programma l'utente deve indovinare il valore della variabile i per terminare il programma, e il valore cambia a ogni iterazione, un istruzione `continue` viene utilizzata per saltare ogni iterazione dove i è vicino a 0, e due istruzioni **break** sono utilizzate per uscire dal ciclo se l'utente indovina il numero o se il numero suggerito è troppo grande.

L'istruzione SWITCH

In c++ esiste un istruzione detta `switch` che è un'alternativa più veloce delle istruzioni if-else, tuttavia se le ottimizzazioni del compilatore sono attivate, il tempo di esecuzione rimane lo stesso, è utile quando ci sono molti if-else.

switch(<*espressione*>) {
case <*caso_1*>:
<istruzioni>
break;
...
case <*caso_n*>:
<istruzioni>
break;

default:
<istruzioni>
}

L'espressione deve essere di tipo integrale (intero o carattere).
Ecco due codici equivalenti, uno con gli if-else, l'altro con uno switch:

```
#include <iostream>

int main()
{
    int month;
    std::cout << "inserisci un numero di mese\n";
    std::cin >> month;

    if (month == 1) std::cout << "e' gennaio\n";
    else if (month == 2) std::cout << "e' febbraio\n";
    else if (month == 3) std::cout << "e' marzo\n";
    else if (month == 4) std::cout << "e' aprile\n";
    else if (month == 5) std::cout << "e' maggio\n";
    else if (month == 6) std::cout << "e' giugno\n";
    else if (month == 7) std::cout << "e' luglio\n";
    else if (month == 8) std::cout << "e' agosto\n";
    else if (month == 9) std::cout << "e' settembre\n";
    else if (month == 10) std::cout << "e' ottobre\n";
    else if (month == 11) std::cout << "e' novembre\n";
    else if (month == 12) std::cout << "e' dicembre\n";
    else std::cout << "quello non e' un mese\n";

    return 0;
}
```

```
#include <iostream>

int main()
{
    int month;
    std::cout << "inserisci un numero di mese\n";
    std::cin >> month;

    switch (month) {
    case 1: std::cout << "e' gennaio\n";
        break;
    case 2: std::cout << "e' febbraio\n";
        break;
    case 3: std::cout << "e' marzo\n";
        break;
    case 4: std::cout << "e' aprile\n";
        break;
    case 5: std::cout << "e' maggio\n";
```

```
        break;
    case 6: std::cout << "e' giugno\n";
        break;
    case 7: std::cout << "e' luglio\n";
        break;
    case 8: std::cout << "e' agosto\n";
        break;
    case 9: std::cout << "e' settembre\n";
        break;
    case 10: std::cout << "e' ottobre\n";
        break;
    case 11: std::cout << "e' novembre\n";
        break;
    case 12: std::cout << "e' dicembre\n";
        break;
    default: std::cout << "quello non e' un mese\n";
    }

    return 0;
}
```

E' molto importante l'utilizzo di `break` perché tutte le etichette `case` dello switch sono come dei goto, quindi il programma trasferirà il controllo all'etichetta giusta, ma per evitare di svolgere tutte le altre istruzioni sotto, bisogna uscire dallo switch.

Alcune funzioni esterne utili

Per gli interi

- `std::swap` scambia il valore di due variabili, necessario includere `<algorithm>`:

```
#include <algorithm>
#include <iostream>

int main()
{
    int A = 1, B = 2;
    std::swap(A, B);
    //  A = 2, B = 1

    return 0;
}
```

- `std::min` e `std::max` calcolano il minimo e il massimo tra due variabili:

```
#include <iostream>
```

```
int main()
{
    int A, B;
    std::cin >> A >> B;

    std::cout << "il minimo e' "  << std::min(A, B) << '\n';
    std::cout << "il massimo e' " << std::max(A, B) << '\n';
    return 0;
}
```

Per i caratteri

- `isalpha` permette di controllare se un carattere è alfabetico,
- `isdigit` controlla se un carattere è numerico,
- `isalnum` controlla se un carattere è alfanumerico
- `tolower` e `toupper` trasformano un carattere nelle sue versioni in minuscolo o maiuscolo rispettivamente

```
#include <iostream>

int main()
{
    char character;
    do {
        std::cout << "inserisci una lettera\n";
        std::cin >> character;
    } while (!isalpha(character));

    std::cout << "la lettera in minuscolo e' ";
    std::cout << tolower(character) << '\n';

    std::cout << "la lettera in maiuscolo e' ";
    std::cout << toupper(character) << '\n';

    return 0;
}
```

Le funzioni matematiche

In C++ esistono delle funzioni matematiche che per usare è necessario includere `<cmath>`, eccone alcuni esempi:

Arrotondamento

- `std::floor` arrotonda per difetto

- `std::ceil` arrotonda per eccesso
- `std::round` arrotonda per difetto o per eccesso

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

int main()
{
    double number = 13.9;

    std::cout << std::floor(number) << '\n'; // output = 13
    std::cout << std::ceil(number)  << '\n'; // output = 14
    std::cout << std::round(number) << '\n'; // output = 14

    return 0;
}
```

Ecco un altro esempio dell'uso di queste funzioni: capire se un numero in virgola mobile (un `float` \ `double` \ `long double`) è intero

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

int main()
{
    double number;
    std::cin >> number;

    if (std::floor(number) == std::ceil(number))
        std::cout  << "il numero e' intero";
    else std::cout << "il numero non e' intero";

    return 0;
}
```

Valore assoluto

- `std::fabs` calcola il valore assoluto di un numero:

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

int main()
{
    std::cout << std::fabs(-10.5) << '\n'; // output = 10.5
    std::cout << std::fabs(+71)   << '\n'; // output = 71
```

```
    return 0;
}
```

Esponenti

- `std::pow` calcola l'elevamento a potenza
  (base e esponente possono anche essere negativi o decimali):

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

int main()
{
    std::cout << std::pow(3, 3) << '\n'; // output = 3^3 = 27
    std::cout << std::pow(2, 5) << '\n'; // output = 2^5 = 32

    return 0;
}
```

- `std::sqrt` calcola la radice quadrata e `std::cbrt` la radice cubica:

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

int main()
{
    // la radice quadrata di 25 è 5
    std::cout << std::sqrt(25) << '\n';

    // la radice cubica di 64 è 4
    std::cout << std::cbrt(64) << '\n';

    return 0;
}
```

- `std::hypot` calcola l'ipotenusa di un triangolo rettangolo dati i cateti:

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

int main()
{
    std::cout << std::hypot(3, 4)  << '\n'; // 3, 4  e 5
    std::cout << std::hypot(5, 12) << '\n'; // 5, 12 e 13

    return 0;
}
```

Logaritmi e trigonometria

- Per calcolare il logaritmo si possono usare `std::log2`, `std::log10` e `std::log`.
- Tra le funzioni trigonometriche ci sono `std::sin`, `std::sinh`, `std::cos`, `std::cosh`, `std::tan`, `std::tanh`, `std::asin`, `std::acos`, `std::atan`, `std::asinh`, `std::acosh` e `std::atanh`.

La complessità temporale di un'operazione

La notazione **O grande** è uno strumento per descrivere il comportamento asintotico di una funzione.

$$ f(n) \leq k \cdot g(n) \quad \forall n \geq n_0 \qquad \Rightarrow \qquad f(n) = O(g(n)) $$

Esempi di ordini:

- $O(1)$: **tempo costante**, cioè non dipende dalla dimensione dell'input
- $O(log(n))$: **tempo logaritmico** come la ricerca binaria
- $O(n)$: **tempo lineare**, il tempo di esecuzione è direttamente proporzionale alla dimensione dell'input, come la ricerca lineare
- $O(n \cdot log(n))$: **tempo semilogaritmico** come il merge sort
- $O(n^2)$: **tempo quadratico** come due loop for annidati
- $O(b^n)$: **tempo esponenziale**.

Livello Intermedio

Le funzioni

Una funzione è un blocco di codice che si può riutilizzare, che elabora degli input detti **parametri** e restituisce un output tramite la parola chiave `return`.

Le basi

Vediamo come si scrivono le funzioni **void** (una funzione void non restituisce nessun valore):

```cpp
#include <iostream>

void Function()
{
    std::cout << "questa e' una funzione\n";
}

int main()
{
    Function(); // chiamata di funzione
    return 0;
}
```

Quando viene chaiamata una funzione, il controllo viene trasferito a quella funzione, e viene eseguito tutto il codice che c'è all'interno di questa.

I parametri

Vediamo adesso come passare dei parametri a una funzione:

```
#include <iostream>

void PrintNumbers(double param1, double param2)
{
    std::cout << "paramtero 1: " << param1 << '\n';
    std::cout << "paramtero 2: " << param2 << '\n';
}

int main()
{
    double A, B;
    std::cin >> A >> B;

    PrintNumbers(A, B);
    return 0;
}
```

In questo caso la funzione `PrintNumbers` accetta come parametri due numeri decimali e li scrive.

return

Si usa la parola chiave `return` per restituire il valore calcolato dalla funzione:

```
#include <iostream>

double Add(double param1, double param2)
{
    return param1 + param2;
}

int main()
{
    double A, B;
    std::cin >> A >> B;

    auto res = Add(A, B);
    std::cout << res;
    return 0;
}
```

In questo esempio viene chiamata la funzione Add per sommare i due numeri decimali, che restituisce con `return` la loro somma, e il valore restituito viene assegnato a un risultato per essere scritto.

Passare parametri per riferimento

Per restituire più di un valore NON si può usare due volte `return`, questo perché in un istruzione `return` il controllo esce dalla funzione, e ciò che viene dopo è ignorato, il modo corretto di fare ciò sarebbe passare i parametri per riferimento.

Perché quando viene chiamata una funzione, viene eseguita una copia dei due input per non modificarli, ma ciò non succede se si fornisce l'**indirizzo**, esempio:

```
#include <iostream>

void Divide(
    int dividend,
    int divisor,
    int& quotient,
    int& rest
)
{
    if (divisor == 0) return;
    quotient = dividend / divisor;
    rest = dividend - divisor * quotient;
}

int main()
{
    int A, B;
    std::cout << "inserisci due numeri\n";
    std::cin >> A >> B;

    int Quotient, Rest;
    Divide(A, B, Quotient, Rest);

    std::cout << "quoziente = " << Quotient << '\n';
    std::cout << "resto = "     << Rest     << '\n';
    return 0;
}
```

Poiché i parametri `dividend` e `divisor` non sono preceduti dall'**operatore di indirizzo** `&`, ne viene eseguita una copia, ma i parametri `quotient` e `rest` vengono modificati.
in questo modo non c'è bisogno di utilizzare il `return`.

Funzione parametro

Ogni funzione ha una sua **firma**, che contiene le informazioni riguardo il nome, il tipo restituito e i datatype dei parametri, quando due funzioni hanno lo stesso nome ma firme diverse, si dice che la funzione è **sovraccaricata**, non si può sovraccaricare la funzione `main`.

E' possibile passare una funzione come parametro di un'altra funzione, in questo caso bisogna scrivere la firma della funzione parametro nell'elenco dei parametri e alla chiamata della funzione non si mettono le parentesi alla funzione parametro.

Esempio:

```
#include <iostream>

void func1(int x)
{
    std::cout << "chiamata funzione 1 con parametro " << x << '\n';
}

void func2(int x)
{
    std::cout << "chiamata funzione 2 con parametro " << x << '\n';
}

void Printer(void function(int param))
{
    std::cout << "queste sono le funzioni: \n";
    function(1);
    function(2);
    function(3);
}

int main()
{
    Printer(func1);
    Printer(func2);
    return 0;
}
```

La ricorsione

Una funzione può chiamare se stessa, quando ciò accade si dice che la funzione è **ricorsiva**, esempio:

```
#include <iostream>

size_t Factorial(size_t number)
{
    if (number <= 2) return number;
    return number * Factorial(number - 1);
}
```

Bisogna fare molta attenzione alla ricorsione, questo perché

- Utilizzare due ricorsioni nella stessa espressione è molto lento
- Quando avvengono troppe ricorsioni si verifica l'errore di **stack overflow**

Ecco un esempio di codice che genera l'errore di stack overflow:

```
int main() { main(); }
```

Qui avviene una ricorsione infinita, che eventualmente riempie la memoria disponibile e il programma viene interrotto.

Dichiarazione e definizione

E' possibile staccare la dichiarazione della funzione dalla definizione, ad esempio la funzione di prima

```
void Divide(
    int dividend,
    int divisor,
    int& quotient,
    int& rest
)
{
    if (divisor == 0) return;
    quotient = dividend / divisor;
    rest = dividend - divisor * quotient;
}
```

Si può riscrivere così:

```
void Divide(
    int dividend,
    int divisor,
    int& quotient,
    int& rest
);

/*
il resto del codice...
......
*/

void Divide(
    int dividend,
    int divisor,
    int& quotient,
    int& rest
)
{
    if (divisor == 0) return;
    quotient = dividend / divisor;
    rest = dividend - divisor * quotient;
}
```

Questa cosa è molto comoda, infatti non si può chiamare una funzione in una riga di codice prima della dichiarazione, ma si possono spostare tutte le dichiarazioni all'inizio del codice e definire la funzione in seguito

E' possibile creare dei parametri facoltativi, che però devono essere messi per ultimi nell'elenco dei parametri di una funzione, riscriviamo l'esempio di prima supponendo che `dividend` sia predefinito a 1 così come `divisor`:

```
void Divide(
    int& quotient,
    int& rest,
    int dividend = 1,
    int divisor = 1
);

void Divide(
    int& quotient,
    int& rest,
    int dividend,
    int divisor
)
{
    if (divisor == 0) return;
    quotient = dividend / divisor;
    rest = dividend - divisor * quotient;
}
```

In questo modo si può fare una chiamata di funzione senza dover per forza specificare quali siano `dividend` e `divisor`, ricordare che non si può fare così con quotient e rest perché sono assegnati per indirizzo e devono essere **lvalue modificabili** quindi non costanti (e quindi neanche numeri dato che sono costanti).

Gli specificatori

Una funzione può essere contrassegnata da alcune parole chiave dette **specificatori**, tra le quali troviamo `inline`, `constexpr` e `static`.

Se una funzione è inline, vuol dire che il compilatore sostituirà con il corpo della funzione tutte le chiamate (se possibile), ad esempio

```
#include <iostream>

inline void PrintNumbers(double param1, double param2)
{
    std::cout << "paramtero 1: " << param1 << '\n';
    std::cout << "paramtero 2: " << param2 << '\n';
}
```

```
int main()
{
    double A, B;
    std::cin >> A >> B;

    PrintNumbers(A, B);
    return 0;
}
```

Viene considerato come

```
#include <iostream>

int main()
{
    double A, B;
    std::cin >> A >> B;

    std::cout << "paramtero 1: " << A << '\n';
    std::cout << "paramtero 2: " << B << '\n';
    return 0;
}
```

Se una funzione è constexpr, vuol dire che il compilatore la eseguirà prima di far partire il programma (a tempo di compilazione), questo si fa di solito con funzioni piccole:

```
#include <iostream>

constexpr double Square(double number)
{
    return number * number;
}

int main()
{
    const double number = 5;

    // questo è calcolato a tempo di compilazione (25)
    std::cout << Square(number);

    return 0;
}
```

Una funzione statica è visibile solo dal file in cui è dichiarata (cioè non può essere messa in un file header); una variabile statica conserva il suo valore tra diverse chiamate di funzione.

Le direttive

In C++ le direttive sono istruzioni che vengono valutate dal **preprocessore**, che modifica il codice secondo le istruzioni e lo fornisce al compilatore.

Elenco di direttive:

- `#define`
- `#elif`
- `#else`
- `#endif`
- `#error`
- `#if`
- `#ifdef`
- `#ifndef`
- `#include`
- `#line`
- `#pragma`
- `#undef`

Abbiamo già visto `#include`, quindi descriveremo le altre diirettive.

La direttiva `#define` permette di definire una **macro** mentre `#undef` la annulla.
Esempio:

```
#define SIZE
```

`SIZE` è una macro, in generale si utilizzano le lettere maiuscole.

La direttiva `#ifdef` controlla se una macro è definita o meno, e ci deve essere sempre una direttiva `#endif` alla fine, il codice presente tra `#ifdef` e `#endif` viene eseguito se e solo se la macro in questione è definita a quella riga:

```
#ifdef SIZE

// questo codice viene eseguito solo se SIZE è definito

#endif
```

La direttiva `#ifndef` controlla se una macro NON è definita (contrario di `#ifdef`), come in questo esempio:

```
#ifndef __cplusplus
#error error STL1003: Unexpected compiler, expected C++ compiler.
#endif
```

Qui `__cplusplus` è una macro definita dal compilatore C++, quindi se il compilatore è sbagliato, si usa la direttiva `#error` per generare un errore

La direttiva `#if` attiva il codice non se la macro è definita ma se è veritiera (cioè si espande in un valore non nullo),

```
#if _HAS_CXX23
#include <optional>
#endif
```

`_HAS_CXX23` è una macro definita come 1 solo se il programma va dalla versione C++23 in poi, e solo in questo caso si include **`<optional>`**

E' possibile combinare `#if` e `#ifdef` con `#elif` ed `#else`.

La direttiva `#line` serve per cambiare la linea del programma, è utile quando si vuole creare dei warning a una certa riga di codice di un'altro file:

```
#line 100 "example.cpp"
// questa riga sarà vista come riga 100 in example.cpp
```

La direttiva `#pragma` è la più complessa, un esempio potrebbe essere questo:

```
#pragma once
```

In questo modo non è possibile includere il file più di una volta.

Le macro

In C++ le macro vengono dichiarata con la direttiva `#define`, e servono a definire simboli che vengono sostituiti dal preprocessore.
#define <*nome*> <*espressione*>

Macro come costanti

Esempio di macro:

```
#define M_PI 3.14159265358979323
```

Ogni volta che il preprocessore trova `M_PI` nel codice lo sostituirà con 3.14159265358979323.

Macro come funzioni

Le macro possono anche accettare degli argomenti:

```
#define SQUARE(x) (x * x)     // sbagliato
#define SQUARE(x) ((x) * (x)) // giusto
```

Le macro non sono delle funzioni, servono solo a sostituire del testo, per questo è sbagliato il primo esempio di macro:

```
#define SQUARE(x) (x * x)
SQUARE(1 + 2); // si espande in 1 + 2 * 1 + 2 = 5 non 9
```

```
#define SQUARE(x) ((x) * (x))
SQUARE(1 + 2); // si espande in (1 + 2) * (1 + 2)
```

Operatori

Con le macro è possibile utilizzare l'operatore `##` per concatenare due simboli:

```
#define DECLARE_VARIABLE(x) int __##x{}
int Var{};
DECLARE_VARIABLE(Var); // si espande in 'int __Var{}';
```

Se una macro è molto lunga la si può mandare a capo con un backslash:

```
#define DECLARE_VARIABLE(x) \
int __##x{}
```

Esempi

Esistono alcune predefinite del preprocessore:

- `__FILE__`: il nome del file corrente
- `__LINE__`: il nome della linea corrente
- `__DATE__`: data di compilazione
- `__TIME__`: ora di compilazione

Altre macro sono definite esternamente, un esempio è `_STD` che si espande in `::std::`

Le enumerazioni

In C++ le enumerazioni sono un modo per rendere il codice più leggibile, dove invece di dichiarare tante macro in questo modo:

```
#define MONDAY    1
#define TUESDAY   2
#define WEDNESDAY 3
#define THURSDAY  4
#define FRIDAY    5
#define SATURDAY  6
#define SUNDAY    7
```

enum

Si può dichiarare un enum:

```
enum Day
{
    monday   ,
    tuesday  ,
    wednesday,
    thursday ,
    friday   ,
    saturday ,
    sunday
};
```

In un enum ogni parola viene associata implicitamente a un valore da 0 a n, per usare l'enum si può fare così:

```
enum Day
{
    monday   ,
    tuesday  ,
    wednesday,
    thursday ,
    friday   ,
    saturday ,
    sunday
};
int main()
{
    int Day = monday;

    // oppure
    ::Day day = Day::monday;

    // l'operatore '::' è usato solo perché c'è una variabile che
    // ha lo stesso nome dell'enum
}
```

enum class

Due enum non possono avere nessun elemento in comune, tuttavia questo si può risolvere rendendoli classi:

```
enum class Day
{
    monday   ,
    tuesday  ,
    wednesday,
    thursday ,
    friday   ,
    saturday ,
    sunday
};
enum class DAY
{
    sunday   ,
    monday   ,
    tuesday  ,
    wednesday,
    thursday ,
    friday   ,
    saturday ,
};
int main()
{
    Day lastDay  = Day::sunday;
    DAY firstDay = DAY::sunday;
}
```

Gli enum e gli enum class sono due modi di definire dei datatype secondari.

Le strutture

In C++ esiste sono un modo per raggruppare più variabili di datatype differenti sotto lo stesso nome, il nome della struttura, che diventa un datatype secondario, un esempio è la struttura `_COORD` di `<Windows.h>` che si definisce così:

```
struct _COORD {
    short X;
    short Y;
};
```

Questo è un tipico esempio di struttura, contiene due variabili `short`, chiamate `X` e `Y` che sono le coordinate di un punto, (questa struttura è usata anche per indicare la posizione del cursore).

Tuttavia la definizione completa della struttura è in realtà questa:

```
typedef struct _COORD {
    SHORT X;
    SHORT Y;
} COORD, *PCOORD;
```

`SHORT` è un typedef per `short`.

Per accedere agli elementi di una struttura si usa l'**operatore di accesso ai membri**:

```
#include <iostream>
#include <Windows.h> // necessario per usare COORD

int main()
{
    COORD point{ 5, 2 };

    std::cout << point.X << '\n'; // output = 5
    std::cout << point.Y << '\n'; // output = 2

    return 0;
}
```

Gli array

In C++ gli array sono degli insiemi di lunghezza costante che contengono valori dello stesso datatype, possono essere statici (allocati nella **stack**) oppure dinamici (allocati nell'**heap**), Per ora vedremo solo come creare array statici.
*<datatype> <nome> [<dimensione>];**

Inizializzazione di un array

Esempio: array di 1024 elementi

```
int Array[1024]{};
```

Per riempire un array con dei valori si usa `std::fill`, necessario includere `<algorithm>`:

```
#include <algorithm>
int main()
{
    int Array[1024]{};
    std::fill(Array, Array + 1024, 1);
```

```
    return 0;
}
```

Accesso agli elementi

Per accedere a un elemento di un array si usa l'**operatore** `[]` (Il primo elemento si trova alla posizione 0):

```
#include <algorithm>
#include <iostream>

int main()
{
    int Array[1024]{};
    std::fill(Array, Array + 1024, 1);

    for (int i = 0; i < 1024; ++i) std::cout << Array[i] << '\n';

    return 0;
}
```

Dimensione

Per calcolare la dimensione di un array si usa `sizeof`, che calcola la dimensione in byte di una certa variabile o di un certo datatype:

```
#include <algorithm>
#include <iostream>

int main()
{
    int Array[1024]{};
    std::fill(Array, Array + sizeof(Array) / sizeof(Array[0]), 1);

    for (int i = 0; i < sizeof(Array) / sizeof(Array[0]); ++i)
        std::cout << Array[i] << '\n';

    return 0;
}
```

Infatti la dimensione in byte di un array è pari al numero degli elementi moltiplicato per la dimensione in byte di uno qualsiasi degli elementi.

Passare un array a una funzione

Per passare un array a una funzione bisogna aggiungere `[]` davanti al nome del parametro nella funzione, e bisogna fornire anche la dimensione, infatti quando passando un'array a una funzione viene passato in realtà

un **puntatore** al primo elemento dell'array, cioè una variabile che contiene l'indirizzo del primo elemento dell'array.

Esempio:

```
#include <algorithm>
#include <iostream>

static void print(int arr[], int size)
{
    std::cout << "L'array e' {";
    for (int i = 0; i < size - 1; ++i)
    std::cout << arr[i] << ' ';
    std::cout << arr[size - 1] << '}';
}

int main()
{
    int Array[16]{}, size = sizeof(Array) / sizeof(Array[0]);
    std::fill(Array, Array + size, 1);

    print(Array, size);
    return 0;
}
```

Il ciclo FOREACH

Un ciclo **foreach** è un for che iterà su ogni elemento di un array in ordine crescente.

for (*<datatype>* *<nome_elemento>* : *<nome_array>*)
{
<istruzioni da ripetere>
}

Esempio:

```
    int Array[128], i{};

    for (auto& el : Array) {
        el = i;
        ++i;
    }
```

Questo ciclo riempie l'array di numeri interi consecutivi a partire da 0, viene usata la parola chiave `auto` come datatype, ed è presente l'**operatore di indirizzo** `&` a indicare che la modificando la variabile, viene modificato anche l'array, in un foreach si può anche usare `const`, che indica che la variabile del ciclo non può essere modificata.

Array multidimensionali

Un array multidimensionale è un array che ha come elementi degli altri array, lo si può dichiarare così:

```
int Array2d[10][3];     // due dimensioni
int Array3d[16][20][2]; // tre dimensioni
// ecc...
```

Supponendo di avere un array 2D, Arr:

```
int Arr[12][12];
```

E due indici i e j, Arr[i] è un array 1D, quindi `(Arr[i])[j]` o più semplicemente `Arr[i][j]` è un elemento dell'Array.

Algoritmi array

Algoritmi di ordinamento

- **Bubble Sort**
  Il Bubble Sort confonta ogni coppia di elementi adiacenti e li scambia se sono nell'ordine sbagliato:

```
#include <algorithm>
static void BubbleSort(int arr[], int size)
{
    for (int i = 0; i < size - 1; ++i)
        for (int j = 0; j < size - i - 1; ++j)
            if (arr[j] > arr[j + 1])
                std::swap(arr[j], arr[j + 1]);
}
```

- **Selection Sort**
  Il Selection Sort trova l'elemento minimo e lo riposiziona all'inizio e poi ripete il processo

```
static void SelectionSort(int arr[], int size)
{
    for (int i = 0; i < size - 1; ++i)
    {
        int min{ i };
        for (int j = i + 1; j < size; ++j)
            if (arr[j] < arr[min]) min = j;
        std::swap(arr[i], arr[min]);
    }
}
```

- **Insertion Sort**
  L'Insertion Sort inserisce gli elementi già ordinati uno alla volta nella posizione corretta

```
static void insertionSort(int arr[], int size)
{
    for (int i = 1; i < size; ++i)
    {
        int key = arr[i], j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

- **Quick Sort**
  Il Quick Sort seleziona un elemento come pivot e ordina l'array in modo che tutti gli elementi minori siano a sinistra del pivot e tutti gli elementi maggiori a destra.

```
static int partition(int arr[], int low, int high)
{
    int pivot = arr[high], i = low - 1;
    for (int j = low; j < high; ++j) if (arr[j] <= pivot)
    {
        i++;
        std::swap(arr[i], arr[j]);
    }
    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}

static void QuickSort(int arr[], int low, int high)
{
    if (low < high) {
        auto pi = partition(arr, low, high);
        QuickSort(arr, low, pi - 1);
        QuickSort(arr, pi + 1, high);
    }
}
```

Per chiamare `QuickSort` si fa così:

```
QuickSort(arr, 0, sizeof(arr) / sizeof(arr[0]) - 1);
```

- Heap Sort
  L'Heap Sort costruisce una struttura dati a forma di albero (heap) dall'array ed estrae l'elemento massimo uno alla volta

```
static void Heapify(int arr[], int n, int i)
{
    int largest = i, left = 2 * i + 1, right = 2 * i + 2;

    if (left  < n && arr[left]  > arr[largest]) largest =  left;
    if (right < n && arr[right] > arr[largest]) largest = right;

    if (largest != i) {
        std::swap(arr[i], arr[largest]);
        Heapify(arr, n, largest);
    }
}

static void HeapSort(int arr[], int size)
{
    for (int i = size / 2 - 1; i >= 0; --i)
        heapify(arr, size, i);

    for (int i = size - 1; i > 0; --i)
    {
        std::swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

Algoritmi di ricerca

- Ricerca lineare Si controlla per ogni indice dell'array se l'elemento corrisponde, funziona sempre

```
static int LinearSearch(const int arr[], int size, int element)
{
    for (int i = 0; i < size; ++i) if (arr[i] == element)
        return i;
    return -1;
}
```

- Ricerca binaria Funziona solo su array ordinati, si parte dalla metà dell'array e dimezzando l'incremento a ogni iterazione si aumenta o si diminuisce l'indice se l'elemento a quell'indice è maggiore o minore dell'elemento da trovare

```
static int binarySearch(const int arr[], int size, int element)
{
    int left{}, right{ size - 1 };
```

```
    while (left <= right)
    {
        int mid = left + (right - left) / 2;

        if (arr[mid] == element) return mid;
        else if (arr[mid] < element) left = mid + 1;
        else right = mid - 1;
    }

    return -1;
}
```

I puntatori

In C++ un **puntatore** è una variabile il cui valore è l'indirizzo di un'altra variabile.

Sintassi di base

***<datatype>* <nome>* = nullptr;**

E' possibile dichiarare un puntatore senza inizializzarlo ma per sicurezza si inizializza a `nullptr`.

Per assegnare un indirizzo a un puntatore si può utilizzare l'**operatore di indirizzo** `&`:

```
    int Variable{};
    int *pointer = &Variable;
```

Per ottenere il valore della variabile a cui punta il puntatore si usa l'**operatore 'di dereferenziazione'** `*`:

```
    int Variable{};
    int *pointer = &Variable;
    int NewVariable = *pointer;
```

Puntatori che puntano a un array \ struttura

Se un puntatore punta a un array, sta in realtà puntando al primo elemento di quell'array, e incrementando il puntatore si può accedere agli altri elementi dell'array:

```
    int arr[5]{1, 2, 3, 4, 5};
    int* pArr = &arr;         // pArr     punta a arr[0] = 1
    std::cout << *(pArr + 1); // pArr + 1 punta a arr[1] = 2
```

Se un puntatore punta a una struttura, si può accedere ai suoi elementi tramite l'**operatore 'di accesso tramite puntatore a membro'** `->`:

```
#include <iostream>
struct point {
    double x;
    double y;
    double z;
};
int main()
{
    point P1{1, -1, 0};
    point* ptr1 = &P1;
    ptr1->z = 3; // modifica di P1.z

    // output = {1, -1, 3}
    std::cout << '{'  << P1.x;
    std::cout << ", " << P1.y;
    std::cout << ", " << P1.z << "}\n";

    return 0;
}
```

Puntatori nell'heap

Per dichiarare un puntatore nell'**heap** si usa l'operatore `new`:

```
    int* ptr = new int; // alloca memoria per un singolo intero
    *ptr = 10; // assegnazione del valore
```

Se il puntatore punta a un array la sintassi cambia leggermente

```
    int* Ptr = new int[7]; // alloca memoria per 7 interi
```

Per accedere a un elemento dell'array si fa come con gli array statici (operatore `[]`).

Quando un puntatore non serve più deve essere deallocato:

```
    delete ptr;   // operatore delete per deallocare puntatori
    ptr = nullptr;

    delete[] Ptr; // operatore delete[] per deallocare array
    Ptr = nullptr
```

Le eccezioni

E' buona pratica, ogni volta che si crea un puntatore nell'heap, di controllare se il puntatore è nullo, in questo caso si può sollevare un'eccezione, che ferma in automatico il programma quando c'è un errore che non va bene:

```
    int* ptr = new int;
    if (!ptr) throw std::bad_alloc();
```

Esistono altri tipi di eccezioni come ad esempio:

- `std::invalid_argument`
- `std::out_of_range`
- `std::overflow_error`

Puntatori intelligenti

In C++ l'utilizzo dei puntatori è facilitato dai puntatori intelligenti (necessario includere `<memory>`): `std::unique_ptr`, `std::shared_ptr` e `std::weak_ptr`:

`std::unique_ptr` è un puntatore che garantisce l'univocità del possesso di un oggetto, ciò significa che solo un unique_ptr può possedere un determinato oggetto alla volta, non si usa `delete` perché quando un unique_ptr esce dal suo ambito, l'oggetto a cui punta viene deallocato automaticamente.

Non si può copiare un unique_ptr ma lo si può spostare con `std::move`

Esempio:

```
#include <iostream>
#include <memory>
#include <utility> // per std::move

int main()
{
    std::unique_ptr<int> ptr1 = std::make_unique<int>(10);
    std::cout << *ptr1 << '\n';

    std::unique_ptr<int> ptr2 = std::move(ptr1);

    // ptr1 è nullo e l'oggetto è spostato a ptr2

    return 0;
}
```

`std::shared_ptr` è un altro puntatore intelligente, e permette la condivisione delle proprietà dell'oggetto, non si usa delete perché l'oggetto a cui uno shared_ptr punta viene deallocato ogni volta che lo shared_ptr viene distrutto o diventa nullo.

shared_ptr ha un contatore implementato per tenere il conto di quanti puntatori punta allo stesso oggetto alla volta.

Esempio:

```cpp
#include <iostream>
#include <memory>

int main()
{
    std::shared_ptr<int> ptr1 = std::make_shared<int>(11);

    {
        auto ptr2 = ptr1;
        std::cout << ptr1.use_count() << '\n'; // output = 2
        std::cout << *ptr2 << '\n';            // output = 11

    } // ptr2 viene distrutto

    std::cout << ptr1.use_count() << '\n';     // output = 1

    // ptr1 viene distrutto, l'oggetto è deallocato
    return 0;
}
```

Tuttavia se ci sono due oggetti che puntano l'uno all'altro (riferimento circolare), il contatore non arriverà mai a zero, ma questo problema si può risolvere sostituendo uno shared_ptr con uno `std::weak_ptr`.

`std::weak_ptr` non incrementa il contatore di riferimenti degli altri shared_ptr, ma può osservare se l'oggetto esiste ancora e può essere convertito in shared_ptr.

Esempio:

```cpp
#include <iostream>
#include <memory>

struct StructA;
struct StructB;
struct StructA {
    std::shared_ptr<StructB> b_ptr;
};
struct StructB {
    std::shared_ptr<StructA> a_ptr;
};

int main()
{
    std::shared_ptr<StructA> a = std::make_shared<StructA>();
    std::shared_ptr<StructB> b = std::make_shared<StructB>();
```

```
    // ogni oggetto punta all'altro
    a->b_ptr = b;
    b->a_ptr = a;

    return 0; // gli oggetti non verranno deallocati
}
```

Ecco un esempio dell'uso del metodo lock di weak_ptr:

```
#include <iostream>
#include <memory>

int main()
{
    std::shared_ptr<int> sptr = std::make_shared<int>(1);
    std::weak_ptr wptr = sptr;

    // codice...
    // sptr.reset() per deallocare l'oggetto

    if (auto checkptr = wptr.lock()) // operatore = non ==
        std::cout << "l'oggetto è ancora vivo\n";
    else std::cout << "l'oggetto è stato deallocato\n";

    return 0;
}
```

Windows.h

`<Windows.h>` è un file header utilizzato per accedere alle API (application programming interface) di Windows, per utilizzarle serve un **handle** (puntatore void) alla console, in questo modo:

```
#include <Windows.h>
HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
```

Cambiare l'attributo della console

Per cambiare il colore del testo e dello sfondo della console si utilizza la funzione `SetConsoleTextAttribute`.
Esempio:

```
    SetConsoleTextAttribute(hConsole, BACKGROUND_RED | FOREGROUND_BLUE);
```

Le macro utilizzate sono queste:

- Per il colore del testo
    - `FOREGROUND_RED`
    - `FOREGROUND_BLUE`
    - `FOREGROUND_GREEN`
    - `FOREGROUND_INTENSITY`
- Per il colore dello sfondo
    - `BACKGROUND_RED`
    - `BACKGROUND_BLUE`
    - `BACKGROUND_GREEN`
    - `BACKGROUND_INTENSITY`

Queste macro vengono combinate utilizzando l'**operatore bitwise or**

Riposizionare il cursore

Per cambiare la posizione del cursore si utilizza la funzione `SetConsoleCursorPosition`.
Esempio:

```
    SetConsoleCursorPosition(hConsole, COORD{ 0, 10 });
```

Qui il cursore viene riposizionato alla coordinata (0, 10) sulla console, il `COORD` prima delle parentesi graffe è un **type cast**.

Ottenere i dati della console

Per ottenere dati come la posizione del cursore o la dimensione del buffer della console c'è un unica funzione che si chiama `GetConsoleScreenBufferInfo`:

```
#include <Windows.h>
HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
CONSOLE_SCREEN_BUFFER_INFO csbi;

int main()
{
    if (!GetConsoleScreenBufferInfo(hConsole, &csbi)) return 1;

    csbi.dwSize;              // dimensioni del buffer
    csbi.dwCursorPosition;    // posizione del cursore
    csbi.wAttributes;         // attributo della console

    // SMALL_RECT con le posizioni dei vertici della finestra
    // nel buffer dell console
    csbi.srWindow;

    csbi.dwMaximumWindowSize; //dimensioni massime della finestra
```

```
    return 0;
}
```

L'if serve per controllare se l'operazione di ottenere i dati dalla console ha avuto successo, e il `CONSOLE_SCREEN_BUFFER_INFO csbi` viene passato per riferimento.

Nascondere il cursore

Si può cambiare le informazioni del cursore con la funzione `SetConsoleCursorInfo`:

```
#include <Windows.h>
HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
CONSOLE_CURSOR_INFO HideCursor{ 10, FALSE };
CONSOLE_CURSOR_INFO ShowCursor{ 10, TRUE };

int main()
{
    SetConsoleCursorInfo(hConsole, &HideCursor); // nasconde il cursore

    // altro codice...

    SetConsoleCursorInfo(hConsole, &ShowCursor); // mostra il cursore
    return 0;
}
```

Pulire un area dello schermo

Per riempire una linea dello schermo esistono queste funzioni: `FillConsoleOutputCharacterA`, `FillConsoleOutputCharacterW` e `FillConsoleOutputAttribute`.

Ecco le definizioni di queste funzioni:

```
__declspec(dllimport) int __stdcall
FillConsoleOutputCharacterA(
    _In_  HANDLE hConsoleOutput,
    _In_  CHAR cCharacter,
    _In_  DWORD nLength,
    _In_  COORD dwWriteCoord,
    _Out_ LPDWORD lpNumberOfCharsWritten
);

__declspec(dllimport) int __stdcall
FillConsoleOutputCharacterW(
    _In_  HANDLE hConsoleOutput,
    _In_  WCHAR cCharacter,
    _In_  DWORD nLength,
    _In_  COORD dwWriteCoord,
    _Out_ LPDWORD lpNumberOfCharsWritten
```

```
);

__declspec(dllimport) int __stdcall
FillConsoleOutputAttribute(
    _In_  HANDLE hConsoleOutput,
    _In_  WORD wAttribute,
    _In_  DWORD nLength,
    _In_  COORD dwWriteCoord,
    _Out_ LPDWORD lpNumberOfAttrsWritten
);
```

`_In_` è una macro che indica che il parametro è di input, mentre `_Out_` indica che il parametro va passato per riferimento per ottenere un'informazione.

Bisogna fornire (in ordine) l'Handle, il carattere \ attributo da scrivere, quanti caratteri \ attributi scrivere, la coordinata di partenza e si ottiene il numero di caratteri \ attributi cambiati.

La differenza tra `FillConsoleOutputCharacterA` e `FillConsoleOutputCharacterW` è il fatto che la seconda funziona anche con i caratteri unicode (estensione della tabella ASCII), Tuttavia esiste la macro `FillConsoleOutputCharacter`, che sceglie in automatico qual è la funzione da utilizzare.

Cancellare tutto

Si può utilizzare `system("cls")` per cancellare qualsiasi cosa scritta sulla console.

Le stringhe

In C++ la classe `std::string` e la classe `std::wstring` rappresentano un **array di caratteri**, e ci sono molti metodi utili per gestire queste stringhe.

Ottenere la dimensione di una stringa

Si usa il **metodo** `size` oppure la **funzione** `std::size`:

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

int main()
{
    setlocale(0, "");

    std::wstring Str;
    std::getline(std::wcin, Str);

    std::wcout << L"la dimensione della stringa è ";
    std::wcout << std::size(Str);

    return 0;
}
```

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

int main()
{
    setlocale(0, "");

    std::wstring Str;
    std::getline(std::wcin, Str);

    std::wcout << L"la dimensione della stringa è ";
    std::wcout << Str.size();

    return 0;
}
```

Capire se una stringa è vuota

Si può controllare se la dimensione è 0 ma esiste anche il metodo `empty`:

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

int main()
{
    setlocale(0, "");

    std::wstring Str;
    std::getline(std::wcin, Str);

    if (Str.empty()) std::wcout << L"la stringa è vuota";
    else std::wcout << L"la stringa non è vuota";

    return 0;
}
```

Accesso agli elementi di una stringa

Si usa il metodo `at` oppure l'operatore `[]`:

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

int main()
{
    setlocale(0, "");
```

```
    std::wstring Str;

    do std::getline(std::wcin, Str);
    while (Str.empty());

    std::wcout << L"il primo carattere è " << Str.at(0) << L'\n';
    std::wcout << L"l'ultimo carattere è " << Str[Str.size() - 1];

    return 0;
}
```

Convertire una stringa in un numero

Se la stringa non è convertibile, la funzione di conversione solleverà un'eccezione `invalid_argument`, ci sono parecchie funzioni:

- `std::stoi` converte in `int`
- `std::stof` converte in `float`
- `std::stod` converte in `double`
- `std::stold` converte in `long double`
- `std::stol` converte in `long`
- `std::stoul` converte in `unsigned long`
- `std::stoll` converte in `long long`
- `std::stoull` converte in `unsigned long long`

Esempio d'uso:

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

int main()
{
    setlocale(0, "");

    wstring Num = L"123";
    std::wcout << stoi(Num); // output = 123
    return 0;
}
```

Convertire un numero in una stringa

Si utilizza `std::to_string` per le stringhe normali e `std::to_wstring` per quelle wide.
Questo è molto utile, per esempio, per calcolare la somma delle cifre di un numero:

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

```
int main()
{
    setlocale(0, "");

    size_t Num, sum;
    std::cin >> Num;
    std::wstring Str = std::to_wstring(Num);

    // per convertire da carattere a numero si sottrae L'0' = 49
    for (const auto& ch : Num) sum += ch - L'0';

    std::wcout << L"la somma delle cifre è " << sum << L'\n';
    return 0;
}
```

Trovare un carattere in una stringa

Si usa il metodo `find`, che trova la prima occorrenza:

```
#include <conio.h>
#include <iostream>
#include <string>

int main()
{
    setlocale(0, "");

    std::wstring string;
    std::wcout << L"inserisci una stringa\n";
    std::getline(std::wcin, string);
    std::wcout << L"inserisci il carattere da cercare\t";
    char c = _getche();
    std::wcout << '\n';

    int pos = string.find(c);
    if (pos == wstring::npos) {
        std::wcout << L"il carattere non è presente in quella stringa";
        return 0;
    }

    std::wcout << L"la prima occorrenza del carattere " << c;
    std::wcout << L"è alla posizione " << pos << L'\n';
    return 0;
}
```

Se il carattere non è presente, il metodo `find` restituisce `string::npos` o `wstring::npos`.

Inserire un carattere in una stringa

Si usa il metodo `insert`:

```
#include <conio.h>
#include <iostream>
#include <string>

int main()
{
    setlocale(0, "");

    std::wstring string;
    std::wcout << L"inserisci una stringa\n";
    std::getline(std::wcin, string);

    int pos;
    do {
        std::wcout << L"inserisci la posizione del nuovo carattere\n";
        std::wcin >> pos;

        if (pos < 0 || pos >= string.size()) {
            std::wcout << L"la posizione non è valida";
            return 0;
        }
    } while (pos < 0 || pos >= string.size());

    std::wcout << L"inserisci il carattere da inserire\t";
    char c = _getche();
    std::wcout << '\n';

    string.insert(string.begin() + pos, c);
    std::wcout << L"la nuova stringa è " << string << L'\n';
    return 0;
}
```

Tagliare una stringa

Si usa il metodo `erase`, che ha 4 varianti:

- Cancellare tutto

```
    str.erase(); // va bene anche str.clear()
```

- Cancellare un singolo carattere nella stringa

```
    str.erase(str.begin() + position);
```

- Cancellare tutti i caratteri della stringa a partire da una posizione

```
    str.erase(position);
```

- Cancellare un determinato numero di caratteri a partire da una posizione

```
    str.erase(position, amount);
```

Per cancellare l'ultimo carattere di una stringa si può usare il metodo `pop_back`:

```
    str.pop_back();
```

Aggiungere un carattere a una stringa

Ci sono due modi per aggiungere un carattere a una stringa: con il metodo `push_back` e con l'**operatore** `+`:

```
#include <conio.h>
#include <iostream>
#include <string>

int main()
{
    setlocale(0, "");

    std::wstring string;
    std::wcout << L"inserisci la stringa\n";
    std::getline(std::wcin, A);

    std::wcout << L"inserisci il carattere da aggiungere\n";
    char ch = _getche();

    std::wstring result = string;  // result = string + ch
    result.push_back(ch);          // va bene lo stesso

    std::wcout << L"nuova stringa " << result;
    return 0;
}
```

Qui si può anche usare l'**operatore** `+=`.

Concatenare due stringhe

Ci sono due modi per concatenare due stringhe: con il metodo `append` e con l'**operatore** `+`:

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

int main()
{
    setlocale(0, "");

    std::wstring A, B;
    std::wcout << L"inserisci due stringhe\n";
    std::getline(std::wcin, A);
    std::getline(std::wcin, B);

    std::wstring result = A;   // result = A + B
    result.append(B);          // va bene lo stesso

    std::wcout << L"stringhe concatenate: " << result;
    return 0;
}
```

Anche qui si poteva usare l'**operatore** `+=`.

Creare una sottostringa

Si usa il metodo `substr`:

```
    std::wstring part = str.substr(FirstIndex, LastIndex);
```

La stessa cosa che si fa così con `erase`:

```
    std::wstring part = str;
    part.erase(LastIndex + 1);
    part.erase(0, FirstIndex);
```

Altro

Esistono sono gli operatori `=` per assegnare una stringa, e `==` per eguagliare due stringhe

---

Gli stringstream

In C++ gli `stringstream` (necessario includere `<sstream>`) sono degli oggetti che ricreano il funzionamento di `cout` e `cin` ma senza avere accesso alla console, ne esistono molte varianti,

- `stringstream` esegue operazioni di input e output
- `ostringstream` esegue operazioni di output
- `istringstream` esegue operazioni di input

- `wstringstream` esegue operazioni di input wide e output wide
- `wostringstream` esegue operazioni di output wide
- `wistringstream` esegue operazioni di input wide

Ottenere la stringa di uno stringstream

Si usa il metodo `str`:

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

int main()
{
    setlocale(0, "");

    std::wstring str;
    std::wcout << L"inserisci una stringa\n";
    std::wcin >> str;

    std::wstringstream stream(str);
    std::wcout << L"la stringa è " << stream.str();
    return 0;
}
```

Gli stringstream di input

Con i `wistringstream` si possono eseguire solo operazioni di input dallo stringstream.
Esempio:

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

int main()
{
    setlocale(0, "");

    std::wstring str = L"123 45.67 read";
    std::wistringstream iss(str);

    int i;
    double d;
    std::wstring s;

    iss >> i >> d >> s; // lettura dati dal wistringstream

    std::wcout << L"intero: "  << i << L'\n'; // output = 123
    std::wcout << L"double: "  << d << L'\n'; // output = 45.67
```

```
    std::wcout << L"stringa: " << s << L'\n'; // output = read

    return 0;
}
```

Si può usare `std::ws` per forzare il salto degli spazi bianchi (spazi, tabulazioni o nuove righe) durante la lettura, esempio:

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

int main()
{
    setlocale(0, "");

    std::wstring str = L"    123    45.67    read";
    std::wistringstream iss(str);

    int i;
    double d;
    std::wstring s;

    iss >> std::ws >> i; // ignora gli spazi bianchi prima della stringa
    iss >> d;            // gli spazi bianchi in mezzo alla stringa non hanno
effetto
    iss >> std::ws >> s; // ignora gli spazi bianchi prima della stringa da
leggere

    std::wcout << L"intero: "  << i << L'\n';
    std::wcout << L"double: "  << d << L'\n';
    std::wcout << L"stringa: " << s << L'\n';

    return 0;
}
```

Gli stringstream di output

Con i `wostringstream` si possono eseguire solo operazioni di output dallo stringstream.
Esempio:

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

int main()
{
    setlocale(0, "");
```

```
    std::wostringstream oss;

    int i = 123;
    double d = 45.67;
    std::wstring s = L"read";

    oss << i << L' ' << d << L' ' << s; // scrittura dati sul wostringstream

    std::wcout << "stringstream: " << oss.str(); // output = 123 45.67 read
    return 0;
}
```

La manipolazione dell'input

I `wostringstream` sono utili quando bisogna usare le funzioni del file header `<iomanip>` (input output manipulator), con dei dati, vediamo i principali:

- **`std::boolalpha`** e **`std::noboolalpha`**
  Si usa `boolalpha` per visualizzare le variabili booleane come il loro valore di verità invece di un numero, `noboolalpha` disattiva `boolalpha`:

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

int main()
{
    setlocale(0, "");

    std::wostringstream stream;
    bool alpha = true;

    stream << boolalpha   << alpha;  // stream.str() = true
    stream << noboolalpha << !alpha; // stream.str() = true0

    std::wcout << L"lo stringstream è " << stream.str();
    return 0;
}
```

- **`std::oct`**, **`std::dec`**, **`std::hex`** e **`std::setbase`**
  `oct` converte i numeri in base 8, `dec` in base 10 e `hex` in base 16, si può anche utilizzare `setbase` al loro posto, ma non con altre basi; `dec` è predefinito:

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

int main()
```

```
{
    setlocale(0, "");

    std::wostringstream stream;
    int number = 40;

    stream << number << L' ';
    stream << std::oct << number << L' '; // 40 in base 8  è 50
    stream << std::hex << number << L' '; // 40 in base 16 è 28
    stream << std::setbase(10);           // reset della base

    // output = 40 50 28
    std::wcout << L"lo stringstream è " << stream.str();
    return 0;
}
```

- **`std::uppercase` e `std::nouppercase`**
  Se un numero convertito in esadecimale ha delle cifre alfabetiche, queste saranno minuscole, per scriverle maiuscole si usa `uppercase`, per annullare `uppercase` si usa `nouppercase`:

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

int main()
{
    setlocale(0, "");

    std::wostringstream stream;

    stream << std::uppercase;
    stream << std::hex << 59 << L' '; // 59 in base 16 è 3b
    stream << std::nouppercase;

    // output = 3B
    std::wcout << L"lo stringstream è " << stream.str();
    return 0;
}
```

- **`std::fixed`, `std::scientific` e `std::defaultfloat` con `std::setprecision`**
  `setprecision` stabilisce quante cifre decimali dopo la virgola ci devono essere, `fixed` indica che il numero di cifre decimali è fisso, `scientific` forza l'output in notazione scientifica e defaultfloat annulla `fixed` o `scientific`:

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

int main()
```

```
{
    setlocale(0, "");

    std::wostringstream stream;
    double num = 9876.54321;

    stream << L"   " << std::fixed        << std::setprecision(2) << num;
    stream << L"   " << std::scientific   << std::setprecision(2) << num;
    stream << L"   " << std::defaultfloat << std::setprecision(2) << num;

    // output =   9876.54   9.88e+03   9.9e+03
    std::wcout << L"lo stringstream è" << stream.str();
    return 0;
}
```

Le espressioni regolari

In C++ le classi `std::regex` e `std::wregex` (necessario includere `<regex>`) sono un modo per controllare se una stringa rispetta una qualche condizione.
Ecco un esempio di regex

```
#include <regex>
std::wregex rgx(L"[1-3]");
```

Le funzioni di ricerca

Esistono tre funzioni importanti riguardanti la ricerca con le regex:

- `regex_match` controlla se l'intera stringa corrisponde alla regex:

```
#include <iostream>
#include <regex>
#include <string>

int main()
{
    setlocale(0, "");

    std::wstring text;
    std::wcout << L"inserisci una stringa\n";
    std::getline(std::wcin, text);

    // controlla se la stringa corrisponde a L"reg"
    if (std::regex_match(text, std::wregex(L"reg")))
        std::wcout << L"hai indovinato la parola\n";
```

```
    return 0;
}
```

- `regex_search` controlla se almeno una porzione della stringa corrisponde alla regex:

```
#include <iostream>
#include <regex>
#include <string>

int main()
{
    setlocale(0, "");

    std::wstring text;
    std::wcout << L"inserisci una stringa\n";
    std::getline(std::wcin, text);

    // controlla se la stringa contiene L"reg"
    if (std::regex_search(text, std::wregex(L"reg")))
        std::wcout << L"hai inserito una stringa corretta\n";

    return 0;
}
```

E' possibile dichiarare uno `std::smatch`, e utilizzare `regex_search` per riempirlo con le occorrenze trovate; uno `std::smatch` è un array non modificabile, con i metodi `size` e `empty`; per convertire l'occorrenza in stringa si utilizza il metodo `str`:

```
#include <iostream>
#include <regex>
#include <string>

int main()
{
    setlocale(0, "");

    std::wstring text;
    std::wcout << L"inserisci una stringa\n";
    std::getline(std::wcin, text);

    std::wsmatch matches;

    if (std::regex_search(text, matches, std::wregex(L"reg")))
    {
        std::wcout << L"prima occorrenza:\n" << L"posizione : " ;
        std::wcout << matches.position(0)    << L"\ntesto : ";
        std::wcout << matches[0].str();
    }
```

```
    return 0;
}
```

- `regex_replace` sostituisce tutte le occorrenze di una regex con una stringa:

```
#include <iostream>
#include <regex>
#include <string>

int main()
{
    setlocale(0, "");

    std::wstring text;
    std::wcout << L"inserisci una stringa\n";
    std::getline(std::wcin, text);

    std::wsmatch matches;

    // sostituisce tutte le occorrenze con L"not reg"
    if (std::regex_replace(text, std::wregex(L"reg"), L"not reg"))
        std::wcout << L"la nuova stringa è " << text << L'\n';

    return 0;
}
```

Le regole delle regex

Per indicare la presenza di un carattere bisogna aggiungere un backslash (`\\`) solo se il carattere è tra questi (metacarattere):

- `.`: corrisponde a qualsiasi carattere singolo eccetto `\n`
- `^`: inizio stringa
- `$`: fine stringa
- `*`: 0+ occorrenze del carattere precedente
- `+`: 1+ occorrenze del carattere precedente
- `?`: 0 o 1 occorrenza del carattere precedente
- `[]`: un carattere tra quelli elencati nelle parentesi
- `()`: un gruppo
- `\\`: backslash
- parentesi graffe

Esempio:

```
std::wregex(L"^12"); // corrisponde a 12 (il 12 è all'inizio)
std::wregex(L"\\^12"); // corrisponde a ^12,
```

Non è eseguita l'escape del carattere backslash: si sta passando una stringa al costruttore della wregex, la stringa contiene solo un carattere backslash.

Le parentesi graffe vengono utilizzate per stabilire quante occorrenze del carattere precedente:

- `{x}` esattamente `x` occorrenze
- `{x,y}` tra `x` e `y` occorrenze
- `{x,}` almeno `x` occorrenze

Esempio:

```
std::wregex(L"n{3}"); // corrisponde a L"nnn"
std::wregex(L"n{3,}"); // corrisponde a L"nnn", L"nnnn", ...
```

Esistono delle **escape sequence** per le regex (escape con `\\`):

- `\d`: qualsiasi cifra
- `\D`: tutto tranne una cifra
- `\w`: un carattere che può comparire nel nome di una variabile
- `\W`: un carattere che non può comparire nel nome di una variabile
- `\s`: uno spazio bianco
- `\S`: qualsiasi carattere tranne uno spazio bianco
- `\b`: un bordo di parola (prima o dopo un carattere che fa parte di una parola)
- `\B`: una posizione che non è un bordo di parola

Le parentesi quadre si usano con queste regole:

- `[...]`: quasiasi carattere tra quelli nelle parentesi quadre (i puntini di sospensione sono indicativi)
- `[^...]`: qualsiasi carattere che non si trova nelle parentesi quadre
- `[A-z]`: qualsiasi carattere compreso tra A e z nella tabella ASCII

Esempio dell'uso delle parentesi quadre:

```
// corrisponde a una sequenza di 2 lettere maiuscole o minuscole
std::wregex(L"[A-Za-z]{2}");
```

Esempio dell'uso delle parentesi tonde:

```
std::wregex(L"(ab)+"); // corrisponde a L"ab", L"abab", ...
```

E' possibile utilizzare l'**operatore di alternanza** `|` in questo modo:

```
std::wregex(L"(abc)|(123)"); // corrisponde a L"123" o L"abc"
```

I vettori

In C++ la classe `std::vector` (necessario includere `<vector>`) viene utilizzata per creare degli array che possono essere estesi o contratti.

Inizializzare un vettore

- Inizializzazione con **lista di inizializzazione**, esempio completo:

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

int main()
{
    setlocale(0, "");

    std::vector<int> vect{ 5, 4, 7 };
    for (int i = 0; i < vect.size(); ++i)
    {
        std::wcout << L"elemento " << i + 1 << L" del vettore: ";
        std::wcout << vect[i] << L'\n';
    }

    return 0;
}
```

- Inizializzazione con un altro vettore:

```
std::vector<int> vect{ 1, 2, 3, 4 };
std::vector<int> NewVect{ vect }; // NewVect = { 1, 2, 3, 4 }
```

- Inizializzazione con l'intervallo di un altro vettore:

```
std::vector<int> vect{ 1, 2, 3, 4, 5 };
std::vector<int> part{ vect.begin() + 1, vect.end() - 1 };
// part = { 2, 3, 4 }
```

- Inizializzazione con dimensione fissa:

```
std::vector<int> vect(5); // vect ha 5 elementi non inizializzati
```

- Inizializzazione con dimensione fissa e valore:

```
std::vector<int> vect(5, 0) // vect ha 5 elementi di valore 0
```

Metodi

Tra i metodi in comune con le stringhe ci sono:

- `size`: restituisce la dimensione (anche la funzione `size` va bene)
- `empty`: restituisce un valore vero se e solo se il vettore è vuoto
- `at`: restituisce un'elemento data la posizione (va bene anche l'operatore `[]`)
- Gli operatori `=` (assegnazione) e `==` (uguaglianza)
- `insert`: inserisce un elemento nel vettore
- `erase`: solo il metodo che elimina un elemento dal vettore
- `push_back`: aggiunge un elemento alla **fine** del vettore
- `pop_back`: elimina l'ultimo elemento dal vettore
- `clear`: rende il vettore nullo

Esempio completo:

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

int main()
{
    setlocale(0, "");
    std::vector<int> OddNumbers(100);
    for (int i = 0; i < size(OddNumbers); ++i)     // funzione size
        OddNumbers.at(i) = 2 * i + 1;              // metodo at

    // elimina il settimo elemento, cioè 13
    OddNumbers.erase(OddNumbers.begin() + 6);      // metodo erase
    // inserisce -1 alla settima posizione
    OddNumbers.insert(OddNumbers.begin() + 6, -1); // metodo insert

    // elimina l'ultimo elemento di OddNumbers
    OddNumbers.pop_back();                         // metodo pop_back

    // size restituisce un size_t
    size_t size = OddNumbers.size();               // metodo size

    // estensione di OddNumbers
    for (int i = size; i < 2 * size; ++i)
        OddNumbers.push_back(2 * i + 1);           // metodo push_back

    // output
    for (int i = 0; i < OddNumbers.size(); ++i)
    {
        if (i == 6) continue;

        std::wcout << L"numero dispari #" << i + 1;
```

```
        std::wcout << L" = " << OddNumbers[i];     // operatore []
        std::wcout << L'\n';
    }

    OddNumbers.clear();                            // metodo clear
    OddNumbers.empty() ?                           // metodo empty
        std::wcout << L"il vettore è stato reso nullo\n" :
        std::wcout << L"qualcosa è andato storto\n";

    // se il vettore è nullo 'return 0' altrimenti 'return 1'
    return !OddNumbers.empty();
}
```

queue e deque

Oltre a `std::vector` ci sono altre due classi simili in C++: **`std::queue`** (includi **`<queue>`**) e **`std::deque`** (includi **`<deque>`**).

`std::queue`

`std::queue` ha tre metodi nuovi oltre a `size` ed `empty`:

- **`push`**: aggiunge un elemento alla fine
- **`pop`**: rimuove l'elemento all'**inizio**
- **`front`**: accede all'elemento all'inizio (senza modificarlo)

Non ha metodi per accedere a elementi al centro

`std::vector` e `std::deque`

`std::deque` aggiunge questi due metodi a `std::vector`:

- **`push_front`**: aggiunge un elemento all'inizio
- **`pop_front`**: rimuove l'elemento all'inizio

Tuttavia è preferibile usare `std::vector` quando possibile perché `std::deque` utilizza gli **indici circolari** per rendere efficienti le operazioni all'inizio e alla fine dell'array, ma questo significa utilizzare l'operatore modulo per accedere agli elementi, rallentando l'accesso.

Le mappe

In C++ le classi **`std::map`** e **`std::unordered_map`** servono per creare oggetti che associano a ogni elemento una chiave unica, le chiavi vengono messe in ordine crescente o alfabetico solo in `std::map`.

Dichiarazione

Esempio di dichiarazione di una mappa:

```cpp
#include <map>

// mappa con una chiave stringa
std::map<std::wstring, int> Map;
```

Esempio di dichiarazione di una mappa non ordinata:

```cpp
#include <unordered_map>
std::unordered_map<std::wstring, int> Map;
```

Inserimento

Si usa il metodo `insert` o l'operatore `=`

```cpp
#include <map>
#include <string>

int main()
{

    // le chiavi vengono ordinate in ordine alfabetico
    std::map<std::wstring, int> ordinals
    {
        { L"one"  , 1 },
        { L"two"  , 2 },
        { L"three", 3 },
        { L"four" , 4 },
        { L"five" , 5 },
        { L"six"  , 6 },
        { L"seven", 7 },
        { L"eight", 8 },
        { L"nine" , 9 }
    };

    // primo modo di inserire un elemento
    map.insert({ L"ten", 10 });

    // secondo modo di inserire un elemento
    map[L"eleven"] = 11;

    return 0;
}
```

Eliminazione

Si usa il metodo `erase`:

```cpp
#include <map>
#include <string>

int main()
{
    std::map<std::wstring, int> ordinals
    {
        { L"one"  , 1 },
        { L"two"  , 2 },
        { L"three", 3 },
    };

    map.erase(L"one");

    return 0;
}
```

Capire se la mappa contiene l'elemento

```cpp
#include <iostream>
#include <map>
#include <string>

int main()
{
    setlocale(0, "");

    std::map<std::wstring, int> ordinals
    {
        { L"one"  , 1 },
        { L"two"  , 2 },
        { L"three", 3 },
        { L"four" , 4 },
        { L"five" , 5 },
        { L"six"  , 6 },
        { L"seven", 7 },
        { L"eight", 8 },
        { L"nine" , 9 }
    };

    std::wstring ordinal;
    std::wcout << L"Inserisci una stringa\n"
    std::wcin >> ordinal;

    if (ordinals.find(L"ordinal") == ordinals.end())
    {
        std::wcout << L"la stringa non corrisponde a un";
        std::wcout << L" numero da 1 a 9\n";
        return 0;
    }
```

```
    std::wcout << L"la stringa corrisponde a " << ordinals[ordinal];
    return 0;
}
```

Iterare su ogni elemento

Ogni elemento di una mappa accede alla chiave con `first` e al valore con `second`.

```
#include <iostream>
#include <map>
#include <string>

int main()
{
    setlocale(0, "");

    std::map<std::wstring, int> ordinals
    {
        { L"one"  , 1 },
        { L"two"  , 2 },
        { L"three", 3 },
        { L"four" , 4 },
        { L"five" , 5 },
        { L"six"  , 6 },
        { L"seven", 7 },
        { L"eight", 8 },
        { L"nine" , 9 }
    };

    std::wcout << L"numeri da 1 a 9 in ordine alfabetico:\n";
    for (const auto& ord : ordinals)
        std::wcout << ord.first << L'(' << ord.second << L")\n";

    return 0;
}
```

Differenze

`std::unordered_map` differisce da `std::map` perché non ordina le chiavi e usa una **tabella hash** per accedere agli elementi (tempo di $O(1)$)

Misurare il tempo

Utilizzo di `std::chrono`

Ci sono tre tipi di clock:

- `std::chrono::system_clock`: orologio utilizzato per l'ora corrente, può essere influenzato da cambiamenti manuali dell'ora
- `std::chrono::steady_clock`: orologio utilizzato per misurare intervalli di tempo
- `std::chrono::high_resolution_clock`: è l'orologio più preciso tra i tre

Esempio d'uso:

```
#include <iostream>
#include <chrono>

int main()
{
    setlocale(0, "");

    auto start = std::chrono::high_resolution_clock::now();

    // codice...
    // ...

    auto end = std::chrono::high_resolution_clock::now();
    std::wcout << L"tempo trascorso: ";
    std::wcout << std::chrono::duration_cast
        <std::chrono::nanoseconds>(end - start).count();
    std::wcout << L" microsecondi\n";

    return 0;
}
```

Utilizzo delle API di Windows

```
#include <iostream>
#include <Windows.h>

int main()
{
    setlocale(0, "");

    LARGE_INTEGER frequency, start, end;
    QueryPerformanceFrequency(&frequency);
    QueryPerformanceCounter(&start);

    // codice...
    // ...

    QueryPerformanceCounter(&end);
    double duration =
        double(end.QuadPart - start.QuadPart) / frequency.QuadPart;
    std::wcout << L"tempo trascorso: " << duration << L" secondi\n";
```

```
    return 0;
}
```

Utilizzo di `GetTickCount`

Queste funzioni contano i millisecondi dall'accensione del sistema, tuttavia la precisione è limitata.

```
#include <iostream>
#include <Windows.h>

int main()
{
    setlocale(0, "");

    DWORD start = GetTickCount64();

    // codice...
    // ...

    DWORD end = GetTickCount64();
    std::wcout << L"tempo trascorso: " << (end - start);
    std::wcout << L" millisecondi\n";

    return 0;
}
```

Viene utilizzato `GetTickCount64` al posto di `GetTickCount` perché `GetTickCount` va in overflow dopo circa 50 giorni.

Utilizzo di `clock`

Questa è una funzione definita nell'header `<ctime>`, che su Windows ha una precisione pressoché uguale a `GetTickCount`, definita dalla macro `CLOCKS_PER_SEC`. Non è un approccio adatto alla misurazione del tempo reale, perché misura il numero di clock ticks della CPU del computer

Esempio:

```
#include <ctime>
#include <iostream>

int main()
{
    setlocale(0, "");

    clock_t start = clock();

    // codice...
    // ...
```

```
    clock_t end = clock();

    std::wcout << L"tempo trascorso: ";
    std::wcout << double(end - start) / CLOCKS_PER_SEC;
    std::wcout << L" secondi\n";

    return 0;
}
```

Generare dei numeri casuali

Con la funzione `rand`

L'utilizzo di `rand` è limitato perché il massimo è di solito impostato a 32767.

```
#include <iostream>
#include <ctime>

int main()
{
    setlocale(0, "");

    std::srand(std::time(NULL)); // impostazione del seme
    std::wcout << std::rand();

    return 0;
}
```

Con i `random_device`

Questo è un metodo che genera numeri casuali di alta qualità, tuttavia è lento, esempio:

```
#include <iostream>
#include <random>

int main()
{
    setlocale(0, "");

    std::random_device rnd;
    std::wcout << rnd();

    return 0;
}
```

Con il mersenne twister

Questo metodo è più veloce rispetto ai `random_device` perché il numero casuale viene calcolato una sola volta per impostare il seme, e a partire da esso si usa un generatore pseudo-casuale per generare degli altri numeri casuali.

Esempio:

```
#include <iostream>
#include <random>

int main()
{
    setlocale(0, "");

    std::random_device rnd;
    std::mt19937 generator(rnd());
    std::uniform_int_distribution<> distr(1, 100);

    std::wcout << distr(generator);

    return 0;
}
```

Se l'intervallo è molto grande è sufficiente cambiare il template di `uniform_int_distribution`:

```
#include <iostream>
#include <random>

int main()
{
    setlocale(0, "");

    std::random_device rnd;
    std::mt19937 generator(rnd());
    std::uniform_int_distribution<long long> distr(0, 1e14);

    std::wcout << distr(generator);

    return 0;
}
```

Livello Avanzato

Le classi

In C++ una classe è una struttura che può contenere dei **metodi**, cioè delle funzioni, oltre alle variabili.

In una classe, si possono definire dei **costruttori** cioè delle funzioni che servono a inizializzare la classe, è anche possibile definire il **distruttore**, una funzione che viene chiamata quando la classe deve essere distrutta.

Variabili (**campi**), metodi, costruttori e distruttore si dicono **membri** della classe, un membro è **pubblico**, se ovunque nel codice si può accedere a esso, o **privato** se ci si può accedere solo dalla classe.

In realtà anche una struttura può contenere dei metodi, la differenza tra una struttura e una classe è il fatto che in una struttura i membri sono pubblici per impostazione predefinita, ed esistono anche dei **costruttori** predefiniti, mentre in una classe i membri sono privati per opzione predefinita.

Per dichiarare una classe si fa così:

```
class Class1
{
    // membri
};
```

Esempio completo di una classe:

```
#define M_PI 3.14159265358979323
#include <iostream>

class Circle
{
public:
    int radius;

    int Diameter()
    {
        return 2 * radius;
    }
    double Circumference()
    {
        return M_PI * radius;
    }
    long double Area()
    {
        return M_PI * radius * radius;
    }
};

int main()
{
    setlocale(0, "");

    Circle c1;
    std::wcout << L"inserisci il raggio del cerchio: ";
    std::wcin >> c1.radius;
```

```
    std::wcout << L"il diametro è: "        << c1.Diameter();
    std::wcout << L"\nla circonferenza è: " << c1.Circumference();
    std::wcout << L"\nl'area è: "           << c1.Area();

    return 0;
}
```

In questo caso, la classe `Circle` è usata per creare un **oggetto** `c1`.

La parola chiave **public** denota quali membri sono pubblici, per rendere dei membri privati si usa la parola chiave **private**.

I costruttori e il distruttore

In una classe in C++ un **costruttore** si distingue da un metodo perché ha lo stesso nome della classe, possono esistere più costruttori, ma devono avere firme diverse, anche il **distruttore** ha lo stesso nome della classe, ma deve essere preceduto da un carattere `~` (premere ALT + 126 dal tastierino numerico, num lock deve essere disattivato).

Sintassi dei costruttori

Esempio:

```
    Circle(int R)
    {
        radius = std::abs(R);
    }
```

Che si può riscrivere così:

```
    Circle(int R) : radius(std::abs(R)) {}
```

Gli oggetti di una determinata classe devono sempre essere inizializzati con i paramteri di un costruttore, quindi di solito si aggiunge un costruttore di default:

```
    Circle()      : radius(0)           {}
    Circle(int R) : radius(std::abs(R)) {}
```

Grazie a questi costruttori un oggetto può essere inizializzato in questi modi:

```
int main()
{
```

```
    Circle c1, c2(1), c3{ 5 };

    // il resto del codice

    return 0;
}
```

Sintassi del distruttore

Ecco un esempio di come si può implementare il distruttore con la classe `Circle`:

```
#define M_PI 3.14159265358979323
#include <iostream>

class Circle
{
public:
    int radius;

    // costruttore di default
    Circle()      : radius(0)          {}

    // costruttore con parametro
    Circle(int R) : radius(std::abs(R)) {}

    // distruttore
    ~Circle()
    {
        setlocale(0, "");
        std::wcout << L"\nl'oggetto è stato distrutto\n";
    }

    int Diameter()
    {
        return 2 * radius;
    }
    double Circumference()
    {
        return M_PI * radius;
    }
    long double Area()
    {
        return M_PI * radius * radius;
    }
};

int main()
{
    setlocale(0, "");

    Circle c1;
```

```
    std::wcout << L"inserisci il raggio del cerchio: ";
    std::wcin >> c1.radius;

    std::wcout << L"il diametro è: " << c1.Diameter();
    std::wcout << L"\nla circonferenza è: " << c1.Circumference();
    std::wcout << L"\nl'area è: " << c1.Area();

    return 0; // qui viene chiamato il distruttore
}
```

Il costruttore di copia

Quando un oggetto viene passato a una funzione per valore, una funzione ritorna un oggetto o un oggetto viene inizializzato con un altro oggetto della stessa classe, viene chiamato il **costruttore di copia**:

```
class Circle
{
public:
    int radius;

    // costruttore di default
    Circle()              : radius(0)           {}

    // costruttore con parametro
    Circle(int R)         : radius(std::abs(R))  {}

    // costruttore di copia
    Circle(Circle& other) : radius(other.radius) {}

    // distruttore
    ~Circle()
    {
        setlocale(0, "");
        std::wcout << L"\nl'oggetto è stato distrutto\n";
    }

    int         Diameter()      { return 2 * radius;             }
    double      Circumference() { return M_PI * radius;          }
    long double Area()          { return M_PI * radius * radius; }
};
```

Esempio in viene chiamato il costruttore di copia:

```
int main()
{
    Circle first = 10;

    // chiamata al costruttore di spostamento
    // c1 è inizializzato come una copia di first
```

```
    Circle c1 = first;

    return 0;
}
```

Il costruttore di spostamento

Quando un oggetto viene spostato (con `std::move`) si chiama il **costruttore di spostamento**:

```
class Circle
{
public:
    int radius;

    // costruttore di default
    Circle()                        : radius(0)           {}

    // costruttore con parametro
    Circle(int R)                   : radius(std::abs(R))  {}

    // costruttore di copia
    Circle(Circle& other)           : radius(other.radius) {}

    // costruttore di spostamento
    Circle(Circle&& other) noexcept : radius(other.radius) {}

    // distruttore
    ~Circle()
    {
        setlocale(0, "");
        std::wcout << L"\nl'oggetto è stato distrutto\n";
    }

    int         Diameter()      { return 2 * radius;             }
    double      Circumference() { return M_PI * radius;          }
    long double Area()          { return M_PI * radius * radius; }
};
```

In un costruttore di spostamento di utilizza `noexcept` per non solleverà mai un'eccezione.

Esempio in viene chiamato il costruttore di spostamento:

```
int main()
{
    Circle first = 10;

    // chiamata al costruttore di spostamento
    // c1 è inizializzato a first e first è distrutto
    Circle c1 = std::move(first);
```

```
    return 0;
}
```

Costruttore con `std::initializer_list`

`std::initializer_list` è una classe utilizzata nei costruttori di classi più complesse (come `std::vector`), ha il metodo `size`.

Vediamo una classe con un costruttore che accetta una lista di inizializzazione come argomento e inizializza delle variabili con somma e prodotto:

```
#include <initializer_list>
#include <iostream>

class DataProcesser
{
public:
    int sum;
    int prod;

    DataProcesser()                                 : sum(0), prod(1) {}
    DataProcesser(std::initializer_list<int> data) : sum(0), prod(1)
    {
        for (const auto& n : data) {
            sum  += n;
            prod *= n;
        }
    }

    void print() const
    {
        setlocale(0, "");

        std::wcout << L"somma:    " << sum  << L'\n';
        std::wcout << L"prodotto: " << prod << L'\n';
    }
};

int main()
{
    setlocale(0, "");

    DataProcesser Data{ 2, -3, -1, 9, 4 };
    Data.print();

    return 0;
}
```

Qui si utilizza `const` dopo `print()` per indicare che il metodo non può modificare i campi della classe, se invece si mette `const` prima del nome di un parametro, si indica che è il parametro a non poter essere modificato.

L'ereditarietà

In C++ una classe può **ereditare** da un'altra classe, questo significa che avrà tutti i membri protetti e pubblici della classe base.

Derivare una classe

Un membro si dice **protetto** (parola chiave `protected`) se è accessibile solo alla classe base e a eventuali classi **derivate** (cioè classi che ereditano dalla classe base).

Ecco un esempio di classe derivata da `std::vector`:

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

class vector_t : std::vector<int>
{
public:

    // costruttori
    vector_t() {}
    vector_t(std::initializer_list<int> list)
    {
        for (const auto& n : list) push_back(n);
    }

    std::wstring string()
    {
        std::wstring output = L"{";
        for (int i = 0; i < size() - 1; ++i)
            output += L' ' + at(i);

        output += at(size() - 1);
        return output;
    }

    void print()
    {
        std::wcout << string();
    }

    void println()
    {
        std::wcout << string() << L'\n';
```

```
    }
};
```

Questa classe aggiunge dei metodi a `std::vector`, ma adesso il codice esterno non riesce ad accedere ai membri di `std::vector` da `vector_t`, questo perché quando una classe eredita da un'altra, tutti i membri ereditati sono **privati**, per risolvere questo problema bisogna aggiungere `public` prima del nome della classe base:

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

class vector_t : public std::vector<int>
{
    // il resto della classe
};
```

Sovrascrivere un metodo

Una classe derivata può **sovrascrivere** un metodo della classe base con la parola chiave `override` solo se la classe base ha dichiarato il metodo come `virtual`, in questo modo la classe derivata eseguirà sempre la sua versione sovrascritta del metodo invece di quella originaria:

```
#include <iostream>

class Base
{
public:

    virtual void show() const
    {
        setlocale(0, "");
        std::wcout << L"questa è la classe base\n";
    }
};

class Derived : public Base
{
    void show() const override
    {
        setlocale(0, "");
        std::wcout << L"questa è la classe derivata\n";
    }
};
```

Il puntatore `this`

Quando un paramtero in una classe ha lo stesso nome di un campo, è possibile usare il **puntatore** `this` per accedere al campo, esso è un puntatore che viene passato implicitamente a tutti i metodi non statici di una classe, si può dereferenziare `this` per ottenere l'oggetto su cui è stato chiamato il metodo.

Esempio d'uso:

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

class MyClass
{
    // campi privati
    int x;
    int y;

    // metodi pubblici
public:

    MyClass(int x, int y)
    {
        setlocale(0, "");

        this->x = x;
        this->y = y;

        // stessa cosa usare this->str()
        std::wcout << L"oggetto corrente: " << (*this).str() << L'\n';
    }

    std::wstring str()
    {
        return L'{' + std::to_wstring(x) + L", " + std::to_wstring(y) + L'}';
    }
};
```

Costruttori e distruttori

Il costruttore della classe base viene chiamato prima di quello della classe derivata, mentre il distruttore della classe derivata viene chiamato prima di quello della classe base.

E' possibile chiamare il costruttore della classe base nel costruttore della classe derivata:

```
#include <iostream>

class Base
{
public:
    Base() {}
    Base(const Base&)
    {
```

```
        setlocale(0, "");
        std::wcout << L"costruttore di copia della classe base\n";
    }
};

class Derived : public Base
{
public:
    Derived() {}
    Derived(const Derived&) : Base(*this)
    {
        setlocale(0, "");
        std::wcout << L"costruttore di copia della classe derivata\n";
    }
};
```

In questo caso non è stato scritto il nome del parametro del costruttore perché esso non è utilizzato.
Se i costruttori presentano dei parametri si può fare così:

```
#include <iostream>

class Base
{
public:
    Base() {}
    Base(int x)
    {
        setlocale(0, "");
        std::wcout << L"costruttore Base con parametro ";
        std::wcout << x << L'\n';
    }
};

class Derived : public Base
{
public:
    Derived() {}
    Derived(int x, int y) : Base(x)
    {
        setlocale(0, "");
        std::wcout << L"costruttore Derived con parametro ";
        std::wcout << y << L'\n';
    }
};
```

Ereditare i costruttori

Quando si deriva una classe, essa non eredita i costruttori, tuttavia ciò si risolve facilmente con `using`, esempio:

```
class vector_t : public std::vector<int>
{
    using std::vector<int>::vector;
};
```

Distruttore virtuale

Quando si distrugge con `delete` un oggetto di una classe derivata tramite un puntatore alla classe base, se il distruttore non è virtuale verrà chiamato solo quello della classe base, causando potenziali perdite di memoria:

```
#include <iostream>

class Base {
public:
    Base ()    { std::cout << "costruttore Base\n"; }
    ~Base()    { std::cout << "distruttore Base\n"; }
};

class Derived : public Base
{
public:
    Derived () { std::cout << "costruttore Derived\n"; }
    ~Derived() { std::cout << "distruttore Derived\n"; }
};

int main()
{
    Base* obj = new Derived();
    delete obj; // il distruttore Derived non verrà chiamato
    return 0;
}
```

In questi casi si rende il distruttore virtuale:

```
#include <iostream>

class Base {
public:
            Base () { std::cout << "costruttore Base\n"; }
    virtual ~Base() { std::cout << "distruttore Base\n"; }
};

class Derived : public Base
{
public:
    Derived ()      { std::cout << "costruttore Derived\n"; }
```

```
    ~Derived()      { std::cout << "distruttore Derived\n"; }
};
```

Il sovraccarico degli operatori

In una classe in C++ si possono definire degli operatori con il **sovraccarico**, si scrive una funzione il cui nome è `operator` seguito dall'operatore, qui ne vedremo alcuni.

sovraccarico dell'operatore `=`

Vediamo come si può sovraccaricare l'operatore di assegnazione di questa classe `coord`:

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

class coord
{
public:
    int X;
    int Y;

    coord()                              : X(0)      , Y(0)       {}
    coord(int x, int y)                  : X(x)      , Y(y)       {}
    coord(const coord& other)            : X(other.X), Y(other.Y) {}
    coord(const coord&& other) noexcept : X(other.X), Y(other.Y) {}

    coord& operator=(const coord other)
    {
        X = other.X;
        Y = other.Y;
        return *this;
    }

    std::wstring str() const
    {
        return L'{' + std::to_wstring(X) + L", " + std::to_wstring(Y) + L'}';
    }
};
```

Con questo operatore `=` si può assegnare un `coord` a un altro `coord`, la funzione restituisce un `coord&`, dove l'operatore di indirizzo indica che viene restituito un riferimento al risultato, questo serve per poter fare delle assegnazioni multiple con una istruzione:

```
int main()
{
    setlocale(0, "");
```

```
    coord cone{ 1, 2 }, ctwo, cthree, cfour, cfive;
    cfive = cfour = cthree = ctwo = cone;

    std::wcout << one.str()   << L'\n';
    std::wcout << two.str()   << L'\n';
    std::wcout << three.str() << L'\n';
    std::wcout << four.str()  << L'\n';
    std::wcout << five.str()  << L'\n';

    return 0;
}
```

Spiegazione: quando si esegue `ctwo = cone`, a `ctwo` viene assegnato il valore di `cone`, ma l'operatore restituisce `*this` cioè il risultato, di conseguenza a `cthree` viene assegnato il valore del risultato, e così via.

sovraccarico degli operatori logici e di confronto

Esempi:

- Operatori `&&` e `||`

```
bool operator&&(const coord other) const
{
    return (X != 0 && Y != 0) && (other.X != 0 and other.Y != 0);
}
bool operator||(const coord other) const
{
    return (X != 0 && Y != 0) || (other.X != 0 and other.Y != 0);
}
```

Questi due operatori consentono di comparare i valori booleani delle due coordinate, in questo caso, una coordinata è falsa se e solo se è nulla.

- Operatori `<`, `>`, `<=`, `>=`, `==` e `!=`

```
bool operator==(const coord other) const
{
    return X == other.X && Y == other.Y;
}
bool operator!=(const coord other) const
{
    return X != other.X || Y != other.Y;
}
bool operator<(const coord other) const
{
    return std::hypot(X, Y) <  std::hypot(other.X, other.Y);
}
bool operator>(const coord other) const
{
```

```
    return std::hypot(X, Y) >  std::hypot(other.X, other.Y);
}
bool operator<=(const coord other) const
{
    return std::hypot(X, Y) <= std::hypot(other.X, other.Y);
}
bool operator>=(const coord other) const
{
    return std::hypot(X, Y) >= std::hypot(other.X, other.Y);
}
```

In questo caso confrontiamo i moduli delle due coordinate (le distanze da {0, 0}), che si calcolano con il teorema di pitagora (`std::hypot`).

sovraccarico degli operatori aritmetici

```
coord operator+(const coord other) const
{
    coord result = *this;
    result.X += other.X;
    result.Y += other.Y;
    return result;
}
coord& operator+=(const coord other)
{
    *this = *this + other;
    return  *this;
}
coord& operator++()    // ++ pre-fisso  (++a)
{
    *this = *this + coord{1, 0};
    return  *this;
}
coord& operator++(int) // ++ post-fisso (a++)
{
    *this = *this + coord{0, 1};
    return  *this;
}
```

Il riferimento (`&`) si trova solo sul tipo restituito dagli operatori `+=` e `++` perché quel risultato viene assegnato solo in caso di una seconda operazione (`c = a += b`) invece di una sola operazione come con l'operatore `+` (`c = a + b`).

L'operatore `++` post-fisso si distingue da quello pre-fisso perché accetta un parametro di tipo `int`, ma esso non viene mai usato.

sovraccarico degli operatori speciali

- Operatore `->`

```
coord* operator->()
{
    return this;
}
```

Si eseguisce `return this` invece di `return *this` perché si sta restituendo un puntatore.

- Operatore `()`

E' possibile utilizzare l'operatore `()` per eseguire una chiamata di funzione con un oggetto (**funtore**):

```
coord operator()(int scalar) const
{
    coord result = *this;
    result.X *= scalar;
    result.Y *= scalar;
    return result;
}
```

In questo caso viene utilizzato l'operatore `()` per eseguire il prodotto per uno scalare, adesso questo operatore si può utilizzare così:

```
int main()
{
    setlocale(0, "");
    coord Coord{2, -1};

    // output = {6, -3}
    std::wcout << L"triplo della coordinata: " << Coord(3).str();
    std::wcout << L'\n';

    return 0;
}
```

I template

In C++ i **template** permettono di creare codice generico, utilizzabile con diversi datatype allo stesso modo.

Funzioni template

Vediamo alcuni esempi di funzioni template:

```
template<typename T> static void swap(T& A, T& B)
{
```

```
    auto temp = B;
    A = temp;
    B = A;
}
template<class T> T add(T A, T B)      { return A + B; }
template<class T> T subtract(T A, T B) { return A - B; }
```

In questo modo si evita di scrivere funzioni diverse per datatype diversi, evitando la duplicazione del codice.

L'utilizzo di `class` al posto di `typename` non cambia nulla, sono solo due diversi modi di scrivere un template, `T` è il nome di un datatype che vale solo all'interno della funzione, il suo reale datatype viene determinato dai parametri nella chiamata di funzione:

```
int main()
{
    setlocale(0, "");

    std::wcout << L"1 + 2 è "     << add(1, 2)          << L'\n';
    std::wcout << L"1.5 - 2.3 è " << subtract(1.5, 2.3) << L'\n';

    return 0;
}
```

Template doppi

E' possibile scrivere una funzione con più di un template come in questo esempio:

```
template<class T, class U> T add(T A, U B)      { return A + B; }
template<class T, class U> T subtract(T A, U B) { return A - B; }
```

Per usare funzioni di questo genere è buona pratica specificare i template nella chiamata di funzione (per prevenire errori di compilazione):

```
int main()
{
    setlocale(0, "");

    std::wcout << L"1 + 2.21 è ";
    std::wcout << add<int, double>(1, 2.21)     << L'\n';

    std::wcout << L"1.5 - 2 è ";
    std::wcout << subtract<double, int>(1.5, 2) << L'\n';

    return 0;
}
```

Classi Template

Anche le strutture possono essere template, ma il ragionamento è analogo; le classi template si distinguono dalle funzioni template perché bisogna sempre specificare il template (anche se è uno solo).

Esempio:

```
template<class Ty> class MyClass
{
public:
    Ty data;
    MyClass(Ty val) : data(val) {}
};

// oggetto di tipo int
MyClass<int>    intObj(4);

// oggetto di tipo double
MyClass<double> doubleObj(7.5);
```

La sintassi infatti è molto simile a classi come `std::vector` o `std::queue`, questo perché sono **classi template** anche loro, le mappe invece hanno un **template doppio**.

Tecniche avanzate dei template

In C++ i template hanno un sacco di utilizzi, qui vedremo quelli avanzati.

Template facoltativi e alias

Per definire un alias di una classe template la sintassi cambia leggermente rispetto a un alias normale.

Esempio di alias normale:

```
using vector_int = vector<int>;
```

Esempio di alias di template:

```
template<typename T> using vec = vector<T>;
```

Nel primo esempio `vector_int` è un vettore di interi, mentre nel secondo esempio `vec` è un vettore di qualsiasi template.
Si può impostare un valore predefinito al template, per esempio int:

```
#include <vector>
template<typename T = int> using vec = std::vector<T>;

int main()
{
    vec<>       intvect{ 1, 2, 3 };
    vec<double> dblvect{ 1.0, 2.0, 3.0 };
    return 0;
}
```

Template con valori costanti

I template possono accettare parametri che non sono tipi (ma devono essere integrali, puntatori o riferimenti), esempio:

```
template<typename T, int size> static void BubbleSort(T arr[])
{
    for (int i = 0; i < size - 1; ++i)
        for (int j = 0; j < size - i - 1; ++j)
            if (arr[j] > arr[j + 1])
                std::swap(arr[j], arr[j + 1]);
}
```

Viene passato un parametro integrale `size` al template, non alla funzione (ma cambia solo la chiamata di funzione, per il resto la logica è la stessa).

La specializzazione di un template

Si utilizza questa tecnica quando si vuole creare un'altra versione di un blocco di codice solo per un template particolare.

```
#include <iostream>

template<typename T> class MyClass
{
public:
    void show()
    {
        std::cout << "versione generica\n";
    }
};

template<> class MyClass<bool>
{
public:
    void show()
    {
```

```
        std::cout << "versione specializzata con template bool\n";
    }
};

int main()
{
    MyClass<int>  obj1;
    obj1.show();  // output = versione generica

    MyClass<bool> obj2;
    // output = versione specializzata con template bool
    obj2.show();
}
```

Questa era una **specializzazione totale**, si parla di **specializzazione parziale** quando la versione specializzata ha un template che dipende da quello della versione base, esempio:

```
#include <iostream>

template<typename T> class MyClass
{
public:
    void show()
    {
        std::cout << "template valore\n";
    }
};

template<typename T> class MyClass<T*>
{
public:
    void show()
    {
        std::cout << "template puntatore\n";
    }
};

int main()
{
    MyClass<int>  obj1;
    obj1.show();  // output = template valore

    MyClass<int*> obj2;
    obj2.show();  // output = template puntatore
}
```

Template variadici

I template variadici consentono di accettare un numero variabile di parametri di tipo:

```
#include <iostream>

template<typename... Args> static void write(Args... args)
{
    setlocale(0, "");
    (std::wcout << ... << args) << L'\n';
}

int main()
{
    write(23, L' ', 1.05, L"written"); // output = 23 1.05written
    return 0;
}
```

I puntini di sospensione vengono espansi dal compilatore in questo modo: `std::wcout << 23 << L' ' << 1.05 << L"written"`.

Oppure la stessa cosa si può fare anche in questo modo:

```
#include <iostream>

template<typename T> static void print_single(T value)
{
    std::cout << value << '\n';
}

template<typename... Args> static void print(Args... args)
{
    (print_single(args), ...);
}

int main()
{
    write(23, L' ', 1.05, L"written");
    return 0;
}
```

std::is_same, std::is_integral e std::enable_if

Utilizzo di `std::is_same`:

```
if constexpr (std::is_same<T, int>::value)
{
    // codice da eseguire se T è un intero
}
```

E' molto importante l'utilizzo di `constexpr`, perché `std::is_same` agisce a tempo di compilazione, lo stesso codice si può riscrivere così:

```
if constexpr (std::is_same_v<T, int>)
{
    // codice da eseguire se T è un intero
}
```

Utilizzo di `std::is_integral`:

```
if constexpr (std::is_integral<T, int>::value)
{
    // codice da eseguire se T è integrale
}
```

Che si può riscrivere così:

```
if constexpr (std::is_integral_v<T, int>)
{
    // codice da eseguire se T è integrale
}
```

Con i template, `std::enable_if` permermette di abilitare una porzione di codice solo se una condizione è vera, in questo caso, se T è integrale.

```
template<typename T>
typename std::enable_if<std::is_integral<T>::value, T>::type
add(T A, T B) { return A + B; }
```

Questo codice può essere riscritto in questo modo con i **concetti**:

```
template<typename T> concept Integral = std::is_integral_v<T>;

template<Integral T> T add(T a, T b) { return a + b; }
```

La differenza è che nel secondo codice avviene un errore di compilazione se la condizione non è rispettata invece di disattivare il codice.

Metaprogrammazione

La **metaprogrammazione** consiste nel fare calcoli a tempo di compilazione, e ciò si può fare con i template, come in questo esempio:

```
template<int N> struct Factorial {
    static const int value = N * Factorial<N - 1>::value;
};
template<> struct Factorial<1> {
    static const int value = 1;
};
template<> struct Factorial<0> {
    static const int value = 1;
};

int main()
{
    setlocale(0, "");

    // output = 720
    std::wcout << L"il fattoriale di 6 è " << Factorial<6>::value << L'\n';

    return 0;
}
```

Poiché la variabile `value` è statica, essa matiene il suo valore tra le diverse chiamate, il compilatore calcola `value` con la prima versione della struttura, fino a quando `N` non arriva a 0 o 1, in quei casi si utilizza la versione specializzata, che imposta `value` a 1.

La parola chiave `friend`

In C++ si dice che una funzione o una classe è amica di un'altra classe se ha accesso alle sue variabili private e protette pur trovandosi all'esterno di quest'ultima.

Esempio:

```
#include <iostream>

class MyClass
{
private: int SecretValue;
public:
    MyClass(int value) : SecretValue(value) {}

    friend void  ShowSecret(const MyClass& obj);
    friend class FriendClass;
};

class FriendClass
{
public:
    void IncrSecret(MyClass& obj) const { obj.SecretValue++; }
    void DecrSecret(MyClass& obj) const { obj.SecretValue--; }
```

```
};

void ShowSecret(const MyClass& obj)
{
    setlocale(0, "");
    std::wcout << L"valore segreto: " << obj.SecretValue << L'\n';
}

int main()
{
    MyClass Object(40);
    FriendClass Friend;

    ShowSecret(Object);
    Friend.IncrSecret(Object);
    ShowSecret(Object);

    return 0;
}
```

In questo esempio la classe `FriendClass` e la funzione `ShowSecret` riescono ad accedere alla variabile privata `SecretValue` di `MyClass` perché vengono dichiarate **`friend`** nella classe `MyClass`.

Si può usare la parola chiave `friend` per sovraccaricare il comportamento degli operatori di classi esterne con una classe.

Esempio: sovraccaricare l'operatore `<<` di `wcout` perché possa funzionare con una classe derivata da `std::vector`.

```
template<class Ty = int>class new_vector : public std::vector<Ty>
{
public: using std::vector<Ty>::vector;

    friend std::wostream& operator<<(
        std::wostream& os,
        const new_vector& vect
        )
    {
        os << L"{";
        for (size_t i = 0; i < vect.size(); ++i)
        {
            os << vect[i];
            if (i < vect.size() - 1) os << L", ";
        }
        os << L"}";
        return os;
    }
};

int main()
{
```

```
    setlocale(0, "");
    new_vector<>  vec{ 1, 2, 3 };
    std::wcout << vec << L'\n';  // output: {1, 2, 3}
    return 0;
}
```

Ecco un altro esempio: sovraccaricare l'operatore `+=` di `std::wstring` perché funzioni con interi.

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

class new_wstring : public std::wstring
{
public: using std::wstring::wstring;

    friend std::wstring& operator+=(
        std::wstring& str,
        const int&  param
        )
    {
        str += std::to_wstring(param);
        return str;
    }
};

int main()
{
    setlocale(0, "");
    new_wstring str = L"123";
    str += 4;
    std::wcout << str << L'\n';  // Output: 1234
    return 0;
}
```

I thread

In C++ un **thread** è un processo, si possono quindi creare più processi per eseguire diversi compiti allo stesso momento.

Si può utilizzare `std::this_thread::sleep_for` per fermare l'esecuzione di un thread per una certa durata di tempo, ad esempio:

```
#include <chrono>
#include <thread>

int main()
{
```

```
    // aspetta 2 secondi
    std::this_thread::sleep_for(std::chrono::seconds(2));
    return 0;
}
```

Thread con una funzione

Per utilizzare un thread bisogna assegnargli una funzione:

```
#include <iostream>
#include <thread>

void Function()
{
    // il codice della funzione
}

int main()
{
    std::thread Process(Function); // attiva il thread
    Process.join();                // aspetta il termine del thread
    return 0;
}
```

Il thread chiamante (quello che esegue l'intero programma) attiva il thread secondario dichiarato con `std::thread`, quindi aspetta il suo termine prima di continuare la sua esecuzione.

Thread con una lambda

Un thread può anche essere attivato senza una funzione, per fare ciò si utilizza una **funzione lambda**:

```
#include <iostream>
#include <thread>

int main()
{
    std::thread Process([]() {

        // codice della funzione lambda

        }
    );
    Process.join();

    return 0;
}
```

Passare parametri a un thread di una lambda

Le lambda possono catturare delle variabili dall'ambiente, se per valore o per riferimento si descrive fra le parentesi quadre:

```
#include <iostream>
#include <thread>

int main()
{
    std::thread Process1([=]() {

        // tutte le variabili sono catturate per valore

        }
    );
    Process1.join();

    std::thread Process1([&]() {

        // tutte le variabili sono catturate per riferimento

        }
    );
    Process2.join();

    return 0;
}
```

```
#include <iostream>
#include <thread>

int main()
{
    setlocale(0, "");
    int param1 = 3, param2 = 5;

    std::wcout << L"l'indirizzo del primo parametro è ";
    std::wcout << &param1 << L'\n';

    std::wcout << L"l'indirizzo del secondo parametro è ";
    std::wcout << &param2 << L'\n';

    std::thread Process1([param1, &param2]() {

        std::wcout << L"parametro 1: "           << param1 << L'\n';
        std::wcout << L"parametro 2: "           << param2 << L'\n';

        std::wcout << L"indirizzo parametro 1: " << &param1 << L'\n';
        std::wcout << L"indirizzo parametro 2: " << &param2 << L'\n';
```

```
        // l'indirizzo del primo parametro cambia:
        // ne viene eseguita una copia
        // l'indirizzo del secondo parametro rimane lo stesso

        }
    );
    Process1.join();

    return 0;
}
```

Thread di funzione membro

Per eseguire un thread di una funzione membro di una classe bisogna passare anche il puntatore a `this`, in questo modo:

```
#include <iostream>
#include <thread>

class MyClass
{
public:
    void show() { std::wcout << L"thread in esecuzione\n"; }
    void operator->() { show(); }
};

int main()
{
    setlocale(0, "");
    MyClass obj;

    // passaggio di operatore e puntatore all'oggetto
    std::thread Process1(&MyClass::operator->, &obj);
    Process1.join();

    // passaggio di operatore e puntatore all'oggetto
    std::thread Process2(&MyClass::show, &obj);
    Process2.join();

    return 0;
}
```

Passare parametri a un thread di una funzione

Se la funzione di un thread ha degli argomenti, è possibile passare questi argomenti direttamente al thread:

```cpp
#include <iostream>
#include <thread>

void funct(int param1, int& param2)
{
    std::wcout << L"parametro 1: " << param1 << L'\n';
    std::wcout << L"parametro 2: " << param2 << L'\n';

    std::wcout << L"indirizzo parametro 1: " << &param1 << L'\n';
    std::wcout << L"indirizzo parametro 2: " << &param2 << L'\n';
}

int main()
{
    setlocale(0, "");
    int var1 = 3, var2 = 5;

    std::wcout << L"l'indirizzo del primo parametro è ";
    std::wcout << &var1 << L'\n';

    std::wcout << L"l'indirizzo del secondo parametro è ";
    std::wcout << &var2 << L'\n';

    std::thread Process1(funct, var1, std::ref(var2));
    Process1.join();

    return 0;
}
```

Si utilizza `std::ref` quando bisogna passare un parametro per riferimento a un thread.

Classi importanti con i thread

FINE