

LEZIONE 0

INFO

► Lezioni:

- Lun 16/09: 9:30 – 13:30 aula DELTA
- Mer 18/09: 9:30 – 13:30 aula OMEGA
- Ven 20/09: 9:30 – 13:30 aula OMEGA
- Mar 24/09: 13:30 – 17:30 aula GAMMA**
- Gio 26/09: 9:30 – 13:30 aula DELTA

Lezioni frontali / seminari + esercitazioni facoltative

Mail: alessandro.petrini@unimi.it

SW: Visual Studio (Windows) oppure g++ ed editor di testo (linux)

Slide ed esercizi su: http://github.com/topopiccione/cpp_2019

INFO

- ▶ Materiale on-line:
- ▶ <http://www.cplusplus.com/>
- ▶ <http://en.cppreference.com/w/>
- ▶ <https://stackoverflow.com/questions/388242/the-definitive-c-book-guide-and-list>
- ▶ **cplusplus.com tutorial:** <http://www.cplusplus.com/files/tutorial.pdf>
- ▶ Lippman, Lajoie, Moo – C++ Primer
- ▶ Scott Meyers – Effective C++ (serie)
- ▶ Herb Sutter – Exceptional C++ (serie)
- ▶ Bjorne Stroustrup – The C++ Programming Language

INTRODUZIONE

INTRODUZIONE

- Da internet...




INTRODUZIONE

- ▶ Da Kernighan – Ritchie «The C Programming Language, 2° Ed.»
- ▶ - C è un linguaggio di programmazione general-purpose
- ▶ - «Not a very high-level language» e «not a big one» e «not specialized»
- ▶ - Progettato e sviluppato per Unix, ma esistono compilatori C per mille piattaforme
- ▶ - Linguaggio non legato ad un hw specifico (alta portabilità)
- ▶ - Tipizzato e tipi fondamentali sono char, integer e float di varie dimensioni
- ▶ - Supporto per gerarchie di dati (array, strutture, unioni)
- ▶ - Espressioni formate da operatori e operandi
- ▶ - Puntatori forniscono sistema di aritmetica della memoria indipend. dalla macchina
- ▶ - Strutture di control-flow: if-then, while-do, do-while, for, switch-case
- ▶ - Programmazione ricorsiva

INTRODUZIONE

- ▶ Cosa manca?
 - ▶ - operazioni per trattare direttamente oggetti composti (stringhe! liste!)
 - ▶ - gestione della memoria (ad eccezione di var dichiarate su stack): niente heap o garbage collector
 - ▶ - I/O: niente funzioni per leggere e scrivere su file
 - ▶ - multi-thread
- ▶ C popolare perché: grande flessibilità, poca sintassi inutile, programmi girano molto velocemente, codice abbastanza portabile
- ▶ Critiche: niente oggetti, classi o modi per creare gerarchie; a volte, scrivere funzioni per eseguire task relativamente semplici può essere difficile

INTRODUZIONE

- ▶ C++ è:
 - ▶ - linguaggio general purpose
 - ▶ - compilato
 - ▶ - statically-typed
 - ▶ - multiparadigmatico (impertivo, obj-oriented, generic [codice riutilizzabile in diversi contesti], high-order [funzioni che manipolano funzioni])
 - ▶ - di medio livello
- 

INTRODUZIONE

- ▶ C++ è linguaggio piuttosto vecchio (introdotto fine anni '80). E' così longevo per la filosofia con cui viene sviluppato ed aggiornato
- ▶ Primo standard ufficiale nel 1998: C++98
- ▶ C++03
- ▶ C++11
- ▶ C++14
- ▶ C++17
- ▶ C++20 (tra poco)
- ▶ Obiettivo: caratteristiche di un linguaggio ad alto livello, senza sacrificare velocità di esecuzione, portabilità e semplicità d'uso

INTRODUZIONE

- ▶ Filosofia:
 - ▶ - evoluzione del C++ è dettata da real-world problem
 - ▶ - non forzare uno stile rigido di programmazione
 - ▶ - aggiungere o utilizzare una feature solo quando serve
 - ▶ - performance non sono quasi mai sacrificate
 - ▶ - dare la possibilità agli sviluppatori di ottenere ogni tipo di controllo, se lo desiderano (da grandi poteri derivano grandi responsabilità)
 - ▶ - quando possibile, rafforzare controlli di correttezza e sicurezza durante la compilazione
- ▶ Lo slogan che più di tutti riassume il C++ può essere questo:

INTRODUZIONE

- ▶ C++ makes programming **more enjoyable** for **serious programmers**

INTRODUZIONE

- ▶ C++ makes programming **more enjoyable** for **serious programmers**

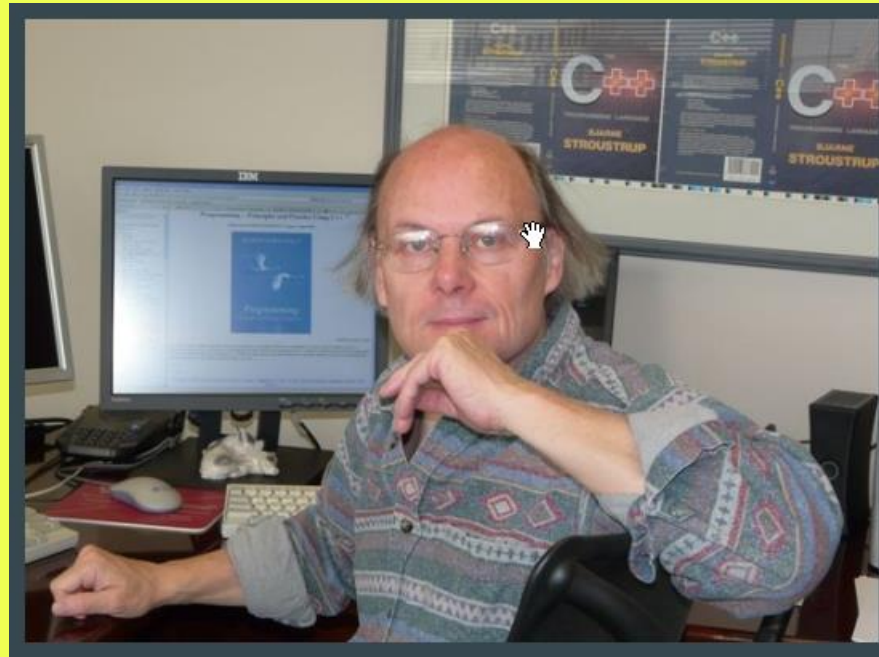
▶ *(Megan Fox, 2008)*



INTRODUZIONE

- ▶ C++ makes programming **more enjoyable** for **serious programmers**

▶ *(Bjorne Stroustrup)*



INTRODUZIONE

- ▶ E' tanto diverso dal C? No: mantiene stessa semantica di base, ma aggiunge nuove strutture e funzionalità.

```
▶ #include <stdio.h>
#include <stdlib.h>
int main( int argc, char** argv ) {
    printf( "Buondì\n" );
    return 0;
}
```

INTRODUZIONE

- ▶ E' tanto diverso dal C? No: mantiene stessa semantica di base, ma aggiunge nuove strutture e funzionalità.

- ▶

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv) {
    printf("Buondì\n");
    return 0;
}
```

- ▶

```
#include <iostream>
using namespace std;
int main(int argc, char** argv) {
    cout << "Ehilà!" << endl;
    return 0;
}
```

INTRODUZIONE

- ▶ Obiettivi (dal punto di vista del programmatore)
- ▶ C++ permette di:
 - ▶ - aggiungere una funzionalità solo quando serve
 - usare lo stile di programmazione più adatto al tipo di applicazione
 - prendere in mano il controllo totale del OS o del sistema (hw), nel caso sia necessario
 - non sacrificare le performance
- ▶ Perché imparare C++?
- ▶ - Uno dei linguaggi più popolari ed amati (<https://www.tiobe.com/tiobe-index/>)
- Usato da tutte le più grandi compagnie del mondo per produrre sw di alta qualità

INTRODUZIONE

Why C++: Users (companies)



Why C++: Users (web browsers)



Why C++: Users (software)



Why C++: Users (games)



INTRODUZIONE

- ▶ Perché tra gli OS manca Linux?
- ▶ Nel 2007, Linus Torvalds: "C++ is a horrible language. It's made more horrible by the fact that a lot of substandard programmers use it, to the point where it's much much easier to generate total and utter crap with it."
- ▶ "Trust me - writing kernel code in C++ is a BLOODY STUPID IDEA. [...] any compiler or language that likes to hide things like memory allocations behind your back just isn't a good choice for a kernel."
- ▶ (<http://harmful.cat-v.org/software/c++/linus>)

INTRODUZIONE

- ▶ Perché tra gli OS manca Linux?
- ▶ Nel 2007, Linus Torvalds: "C++ is a horrible language. It's made more horrible by the fact that a lot of substandard programmers use it, to the point where it's much much easier to generate total and utter crap with it."
- ▶ "Trust me - writing kernel code in C++ is a BLOODY STUPID IDEA. [...] any compiler or language that likes to hide things like memory allocations behind your back just isn't a good choice for a kernel."
- ▶ (<http://harmful.cat-v.org/software/c++/linus>)

↑ [-] **crocodile7** 32 points 2 years ago*

↓ Bjarne is a amazing -- he's the *only* programmer in the world who *fully* understands *all* of C++.

INTRODUZIONE

- ▶ Perché tra gli OS manca Linux?
- ▶ Nel 2007, Linus Torvalds: "C++ is a horrible language. It's made more horrible by the fact that a lot of substandard programmers use it, to the point where it's much much easier to generate total and utter crap with it."
- ▶ "Trust me - writing kernel code in C++ is a BLOODY STUPID IDEA. [...] any compiler or language that likes to hide things like memory allocations behind your back just isn't a good choice for a kernel."
- ▶ (<http://harmful.cat-v.org/software/c++/linus>)

↑ [-] **crocodile7** 32 points 2 years ago*

↓ Bjarne is a amazing -- he's the *only* programmer in the world who *fully* understands *all* of C++.

Edit: seems like he said he does not. The situation is worse than I thought.

- ▶ (https://www.reddit.com/r/programming/comments/1szncc/how_much_of_this_is_true_c_vs_c_by_linus_torvald/)

INTRODUZIONE

- ▶ Perché tra gli C++
- ▶ Nel 2007, Linus Torvalds called C++ a "horrible language. It's made more horrible by the fact that a lot of programmers use it, to the point where it's much much easier to get things done than to wrap with it."
- ▶ "Trust me - while C++ is a BLOODY STUPID IDEA. [...] any compiler or language that does things like memory allocations behind your back just isn't a good choice for..."
- ▶ (<http://harmfulerror.com/c++/linus>)



↑ [-] crocod... 2 years ago
↓ Bjarne is a amazing -- he is the *only* programmer in the world who *fully* understands *all* of C++.
Edit: seems like he said he does not. The situation is worse than I thought.

- ▶ (https://www.reddit.com/r/programming/comments/1szncc/how_much_of_this_is_true_c_vs_c_by_linus_torvald/)

INTRODUZIONE

- ▶ Perché tra gli C
- ▶ Nel 2007, Linus Torvalds, il creatore di Linux, si lamentò per il fatto che a lui, che è un programmatore, era così difficile imparare a programmare in C++ che era più facile per lui imparare a programmare in C. (http://harmful.virginia.edu/c++/linus)
- ▶ "Trust me - with C++ is a BLC (Bjarne's Little C++) language that is like memory, your back just isn't a good choice for
- ▶ the horrible language by the fact that it's much much



[-] croco 2 years ago
Bjarne is a amazing -- he is the *only* programmer in the world who *fully* understands *all* of C++.
Edit: seems like he said he does not. The situation is worse than I thought.

- ▶ (https://www.reddit.com/r/programming/comments/1szncc/how_much_of_this_is_true_c_vs_c_by_linus_torvald/)

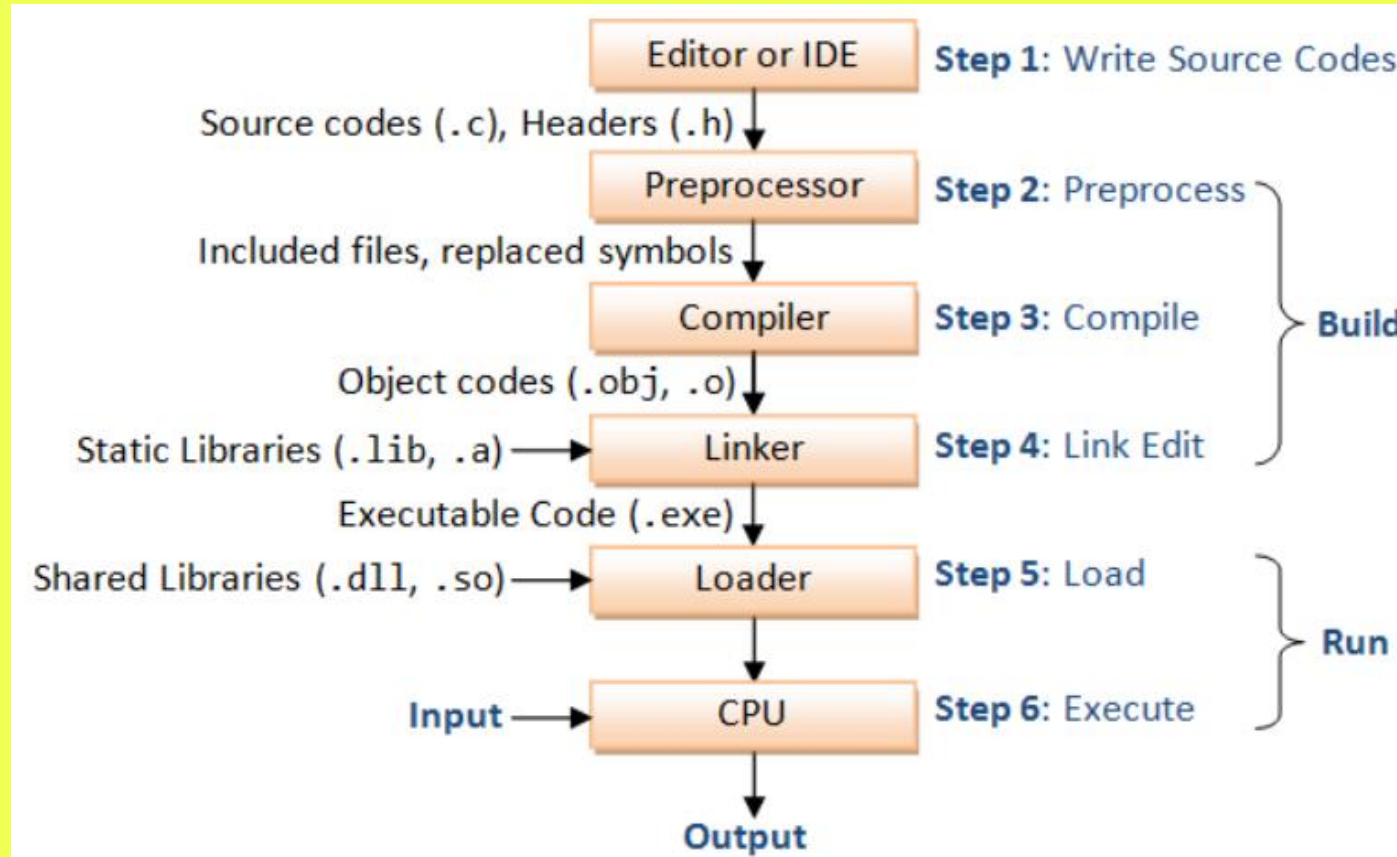
BASI

BASI

- ▶ Diamo per scontato che sappiate un linguaggio di programmazione di alto livello e che abbiate visto:
 - ▶ - concetto di variabile
 - ▶ - array e struct
 - ▶ - operatori booleani e aritmetici
 - ▶ - strutture di iterazione for, while, do
 - ▶ - strutture di branch: if..then
- ▶ Consultare Kernighan – Ritchie per altre info
Concentriamoci su peculiarità che ci serviranno in C++
 - ▶ - dare la possibilità di ottenere ogni tipo di controllo
 - rafforzare controlli di correttezza e sicurezza durante la compilazione

COMPILAZIONE

- C++ è linguaggio compilato



COMPILAZIONE

- ▶ Compiliamo ed eseguiamo il primo programma:

- ▶ `#include <iostream>`

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

- ▶ `g++ helloW.cpp -o hello`

`./hello`

TIPI BASE

- ▶ C++ prevede lo stesso insieme di tipi base di C:
- ▶ Booleane: `bool` boolean literal: `true false`
- ▶ Caratteri: `char` char literals: `'a', 'c', '9', '\n'`
- ▶ Interi: `int, short, long...` int literals: `0, 123, -34, 0xff`
- ▶ Floating point: `double, float` floating point literals: `1.2, -32.99, .333, 1.4e-23`
- ▶ Standard Template Library definisce un mucchio di nuovi tipi!
- ▶ Stringhe: `std::string` string literal: `"Ciao"`
- ▶ Numeri complessi
- ▶ Ecc...

TIPI BASE

- ▶ Caratteri: `char` `char` literals: `'a', 'c', '9', '\n'`
- ▶ Interi: `int`, `short`, `long...` `Int` literals: `0, 123, -34, 0xff`
- ▶ Storicamente: `char` variabili a 8 bit, possono essere usati per contenere valori numerici da -128 a 127, o da 0 a 255
- ▶ `int`, `long`, `short` hanno delle implementazioni che sono dipendenti dal sistema operativo su cui il programma viene compilato (da standard library utilizzata). Idem per lunghezza (dimensione in bit) dei puntatori.
- ▶ E' più corretto usare tipi numerici che indicano esplicitamente da quanti bit sono formati (includendo `<cstdint>`):

▶ <code>char</code>	<code>int8_t</code>	<code>unsigned char</code>	<code>uint8_t</code>
▶ <code>int</code>	<code>int32_t</code>	<code>unsigned int</code>	<code>uint32_t</code>
- ▶ Tutto parte da 0

NOMI IN C++

- ▶ I nomi in C++ sono sequenze di caratteri che iniziano con una lettera e contengono lettere, numeri e underscore. Es:

```
x  
nome_della_variabile  
trasformata21
```

- ▶ Nomi non validi:

```
12elementi  
nota$musicale  
funzione principale
```

- ▶ C++ è case sensitive
- ▶ Non usare parole riservate dal linguaggio: if, else, while, int...
- ▶ Nomi che cominciano con underscore sono di norma riservati a funzioni o macro di sistema

NOMI IN C++

- ▶ Questione di stile!
- ▶ Acronimi e abbreviazioni creano confusione: xfv, ccls, nouv, ccptrs...
- ▶ Nomi troppo corti possono essere usati, ma solo in contesti riconoscibili
 - i: variabile di un ciclo for
 - x: variabile temporale locale
- ▶ Nomi a classi e variabili devono essere descrittivi, ma non troppo:
 - getIntersecPoint()
 - setNewTime()
 - ...
 - remainingFreeSlotsInAVeryLongComputationTableOfDouble
 - arrayDiElementiDiInteriContenteSommeParziali[]

STRINGHE ED INPUT

- ▶ Introduciamo variabili e i/o a console:

- ▶ `#include <iostream>`

```
int main() {  
    std::cout << "Immetti il tuo nome" <<  
        " seguito da invio" << std::endl;  
  
    std::string nome;  
    std::cin >> nome;  
  
    std::cout << "Grazie " + nome + "!";  
  
    return 0;  
}
```

BASE TYPES

- ▶ Introduciamo variabili e i/o a console:

- ▶ `#include <iostream>`

```
int main() {  
    std::cout << "Immetti il tuo nome ed età separati da spazio" <<  
        " e seguiti da invio" << std::endl;  
  
    std::string nome;  
    int eta;  
    std::cin >> nome >> eta;  
  
    std::cout << "Grazie " + nome + "! Sei vecchio, hai "  
        << eta << " anni. Riguardati" << std::endl;  
  
    return 0;  
}
```


UHM...

- ▶ Operatore "+" ha funzioni diverse su stringhe ed interi... Non solo:

■ Strings	■ Integers and floating-point numbers
■ <u>cin</u> >> reads a word	■ <u>cin</u> >> reads a number
■ <u>cout</u> << writes	■ <u>cout</u> << writes
■ + concatenates	■ + adds
■ += s adds the string s at end	■ += n increments by the int n
■ ++ is an error	■ ++ increments by 1
■ - is an error	■ - subtracts
■ ...	■ ...

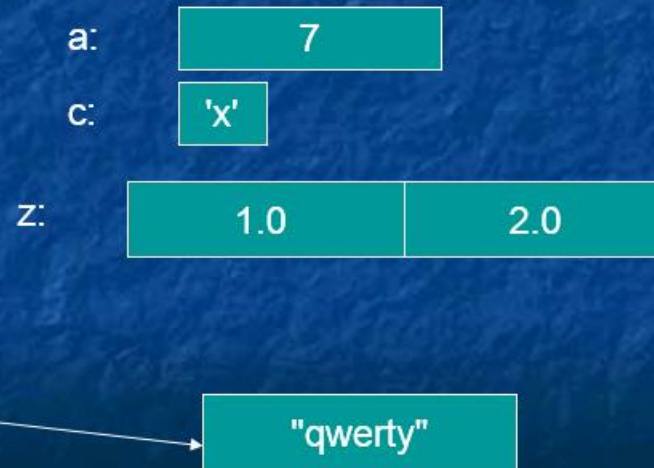
- ▶ Il tipo di variabile determina quali operazioni siano valide e quale sia il significato per quel determinato tipo (overload degli operatori)

OGGETTI, DICHIARAZIONE, DEFINIZIONE

- Dalle presentazioni di Bjarne Stoustrup:

- An object is some memory that can hold a value of a given type
- A variable is a named object
- A declaration names an object

```
int a = 7;  
char c = 'x';  
complex<double> z(1.0,2.0);  
string s = "qwerty";
```



- C'è differenza tra dichiarazione e definizione!

OGGETTI, DICHIARAZIONE, DEFINIZIONE

► Dichiarazione:

- introduce un nome in uno scope (parte di codice delimitato da parentesi graffe)
- ne dichiara anche il tipo
- può includere una inizializzazione
- un nome, prima di essere usato, deve essere dichiarato
- spesso nomi introdotti negli header file, tramite direttiva `#include <file.h>`

► Es:

```
int a;  
int b, c;  
double inverso(int x);
```

► Definizione:

- una dichiarazione che definisce completamente l'entità dichiarata

► Es:

```
int a = 7;  
int b = 0, c = 3;  
double inverso(int x) { return 1.0 / (double) x; }
```

OGGETTI, DICHIARAZIONE, DEFINIZIONE

- ▶ Type safety: ogni oggetto dichiarato, verrà utilizzato in accordo con il suo tipo:
 - una variabile verrà utilizzata solo dopo essere stata definita (inizializzata)
 - le sole operazioni che possono essere eseguite sono quelle definite per il tipo
 - ogni operazione definita per quel tipo lascia la variabile in uno stato valido
- ▶ Idealmente: static type safety
 - Un programma che viola la type safety non compila
- ▶ Idealmente: dynamic type safety
 - un programma che viola la type safety compila, ma la violazione viene gestita a runtime

OGGETTI, DICHIARAZIONE, DEFINIZIONE

- ▶ Type safety: ogni oggetto dichiarato, verrà utilizzato in accordo con il suo tipo:
 - una variabile verrà utilizzata solo dopo essere stata definita (inizializzata)
 - le sole operazioni che possono essere eseguite sono quelle definite per il tipo
 - ogni operazione definita per quel tipo lascia la variabile in uno stato valido
- ▶ Idealmente: static type safety
 - Un programma che viola la type safety non compila
- ▶ Idealmente: dynamic type safety
 - un programma che viola la type safety compila, ma la violazione viene gestita a runtime
- ▶ C++ non è totalmente "statically type safe"
 - sarebbe un grosso vincolo per l'espressione di idee
- ▶ C++ non è totalmente "dinamically type safe"
 - potrebbe interferire con le prestazioni
- ▶ Nota: RTTI non è considerata una buona pratica in C++

OGGETTI, DICHIARAZIONE, DEFINIZIONE

- ▶ Type qualifier: forniscono ulteriori informazioni o vincoli alla variabile nel momento della sua dichiarazione
- ▶ `const`: dichiara e definisce una variabile costante
- ▶ `int x = 3;`
- ▶ `const int x = 3;`
- ▶ `volatile`: il valore della variabile può cambiare in un modo non esplicitamente definito dal programma

OGGETTI, DICHIARAZIONE, DEFINIZIONE

```
▶ int main() {  
    int a = 20000;  
    char c = a;  
    int b = c;  
  
    if (a != b)  
        std::cout << "Houston, abbiamo un problema" << std::endl;  
    else  
        std::cout << "Raga, tutto rego" << std::endl;  
  
    double d;  
    double dd = d;  
    std::cout << "d = " << d << "; dd = " << dd << std::endl;  
}
```

OGGETTI, DICHIARAZIONE, DEFINIZIONE

- Es: scriviamo un programma per convertire cm a pollici

```
▶ int main() {
    const double cmPerPollice = 2.54;
    int inputVal;
    char inputUnit;

    while (std::cin >> inputVal >> inputUnit) {
```

▶ $\{ \}$

OGGETTI, DICHIARAZIONE, DEFINIZIONE

- ▶ Es: scriviamo un programma per convertire cm a pollici

```
▶ int main() {  
    const double cmPerPollice = 2.54;  
    int inputVal;  
    char inputUnit;  
  
    while (std::cin >> inputVal >> inputUnit) {  
        if (inputUnit == 'i')  
            std::cout << "val: " << inputVal << " in -> " <<  
                << inputVal * cmPerPollice << " cm." <<std::endl;  
        else if (inputUnit == "c")  
            std::cout << "val: " << inputVal << " cm -> " <<  
                << inputVal / cmPerPollice << " in." <<std::endl;  
        else  
            return 0;  
    }  
}
```

ANTICIPAZIONE SU C++14

- ▶ `auto`

il compilatore desume il tipo di dato specificato nella definizione

- ▶

```
auto x = 1;      // 1 è intero, allora x è int
auto z = 'f';    // z è char
auto d = 1.2;    // d double
```
- ▶

```
#include <cmath>
auto k = std::sqrt(2)    // tipo viene intuito dal return type di sqrt
```
- ▶

```
auto s = "Sono una stringa!"    // Ehm...
```

ANTICIPAZIONE SU C++14

- ▶ `auto`

il compilatore desume il tipo di dato specificato nella definizione

- ▶

```
auto x = 1;      // 1 è intero, allora x è int
auto z = 'f';    // z è char
auto d = 1.2;    // d double
```

- ▶

```
#include <cmath>
auto k = std::sqrt(2)    // tipo viene intuito dal return type di sqrt
```

- ▶

```
auto s = "Sono una stringa?"    // const char[]
```

ANTICIPAZIONE SU C++14

- ▶ `auto`

il compilatore desume il tipo di dato dal tipo della definizione

- ▶

```
auto x = 1;      // 1 è intero, allora x è int
auto z = 'f';    // z è char
auto d = 1.2;    // d double
```
- ▶

```
#include <cmath>
auto k = std::sqrt(2)    // tipo viene intuito dal return type di sqrt
```
- ▶

```
auto s = "Sono una stringa!"    // const char[]

auto f;    // No. Smettila.
```
- ▶ Nota: il tipo è stabilito in sede di compilazione, non a runtime!

ESPRESSIONI E ISTRUZIONI

- ▶ Espressioni sono composte da operatori e operandi

```
▶ int base = 12;  
  int altezza = 24;  
  int area = base * altezza;      // Espressione che calcola prodotto
```

Regole per stabilire la precedenza degli operatori; nel dubbio, usare le parentesi

- ▶ Non mi pare sia il caso di fare altri esempi!

OPERATORI

- Lista operatori e precedenza. Se in dubbio, usare le parentesi!!!

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	a++ a-- type() type{} a() a[] . ->	Suffix/postfix increment and decrement Functional cast Function call Subscript Member access	
3	++a --a +a -a ! ~ (type) *a &a sizeof new new[] delete delete[]	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT C-style cast Indirection (dereference) Address-of Size-of ^[note 1] Dynamic memory allocation Dynamic memory deallocation	Right-to-left
4	.* ->*	Pointer-to-member	Left-to-right
5	a*b a/b a%b	Multiplication, division, and remainder	
6	a+b a-b	Addition and subtraction	

OPERATORI

- Lista operatori e precedenza. Se in dubbio, usare le parentesi!!!

7	<< >>	Bitwise left shift and right shift	
8	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
9	== !=	For relational operators = and ≠ respectively	
10	a&b	Bitwise AND	
11	^	Bitwise XOR (exclusive or)	
12		Bitwise OR (inclusive or)	
13	&&	Logical AND	
14		Logical OR	Right-to-left
15	a?b:c	Ternary conditional ^[note 2]	
	throw	throw operator	
	=	Direct assignment (provided by default for C++ classes)	
	+= -=	Compound assignment by sum and difference	
	*= /= %=	Compound assignment by product, quotient, and remainder	
	<<= >>=	Compound assignment by bitwise left shift and right shift	
	&= ^= =	Compound assignment by bitwise AND, XOR, and OR	
16	,	Comma	Left-to-right

ESPRESSIONI E ISTRUZIONI

► Istruzioni:

- espressione seguita da ";"
- una dichiarazione
- "control statement" che definisce il flusso di esecuzione

►
`A = b;`
`double pi = 3.14;`
`if (x == 2) y = 4;`
`while (cin >> numero) vettoreDiNumeri.push_back(numero);`
`float media = (valore1 + valore2) / 2.0f;`
`return -1;`

FLOW CONTROL: SELEZIONE

- Sometimes we must select between alternatives
- For example, suppose we want to identify the larger of two values. We can do this with an **if** statement

```
    if (a<b)           // Note: No semicolon here
        max = b;
    else               // Note: No semicolon here
        max = a;
```

- The syntax is

```
    if (condition)
        statement-1    // if the condition is true, do statement-1
    else
        statement-2    // if not, do statement-2
```

FLOW CONTROL: SELEZIONE

Da standard, la condizione di un if..else può essere:

- una espressione
- una singola dichiarazione di variabile opportunamente inizializzata

Es:

```
if (true) ....  
if (0) ....  
int x = 3;  
if (x > 2) ....  
if (int y = 3) ....
```

FLOW CONTROL: SELEZIONE

Da standard, la condizione di un if..else può essere:

- una espressione
- una **singola** dichiarazione di variabile opportunamente inizializzata

Es:

```
if (true) ....  
if (0) ....  
int x = 3;  
if (x > 2) ....  
if (int y = 3) ....
```

Questi casi sono validi?

```
if ((int x = 3)) ....  
if (!(1 == 0) && (int z = 4))) ....
```

Domani esamineremo differenza tra espressioni e istruzioni

Nota C++17: `if (int a = 12, b = 17; a && b) ...`

FLOW CONTROL: SELEZIONE

Switch..case permette di confrontare un valore con un insieme di espressioni costanti e modificare il flow di esecuzione. E' simile a catena di if..else

```
switch (expression)
{
    case constant1:
        group-of-statements-1;
        break;
    case constant2:
        group-of-statements-2;
        break;
    .
    .
    .
    default:
        default-group-of-statements
}
```

switch example	if-else equivalent
<pre>switch (x) { case 1: cout << "x is 1"; break; case 2: cout << "x is 2"; break; default: cout << "value of x unknown"; }</pre>	<pre>if (x == 1) { cout << "x is 1"; } else if (x == 2) { cout << "x is 2"; } else { cout << "value of x unknown"; }</pre>

Break interrompe esecuzione di case corrente ed esecuzione esce da scope definito da switch.

```
1 switch (x) {
2     case 1:
3     case 2:
4     case 3:
5         cout << "x is 1, 2 or 3";
6         break;
7     default:
8         cout << "x is not 1, 2 nor 3";
9 }
```

FLOW CONTROL: ITERAZIONE WHILE E FOR

```
int main() {  
    int i = 0;  
    while (i < 100) {  
        std::cout << i << " " << square( i ) << std::endl;  
        i++;  
    }  
    return 0;  
}
```

```
int main() {  
    for (int i = 0; i < 100; i++) {  
        std::cout << i << " " << square( i ) << std::endl;  
    }  
    return 0;  
}
```

FLOW CONTROL: ITERAZIONE WHILE E FOR

Sintassi for:

```
for (inizializzazione; condizione; incremento) statement
```

Ciascuna delle tre fasi è opzionale. Le istruzioni:

```
for ( ; ; )
```

```
for ( ; n < 10; )
```

sono valide. Possono essere anche composte:

```
for ( n = 0, i = 100; n != i; n++, i-- )
```

break: termina il ciclo for

continue: termina prematuramente l'iterazione corrente

FUNZIONI

```
int main() {  
    for (int i = 0; i < 100; i++) {  
        std::cout << i << "  " << square( i ) << std::endl;  
    }  
    return 0;  
}
```

Cosa è `square(i)`? Chiamata alla funzione `square`

```
int square( int x ) {  
    return x * x;  
}
```

Definiamo una funzione quando vogliamo separare due blocchi di codice, per:

- descrivere concetti separati
- rendere il programma più leggibile
- riutilizzare codice in più parti del programma (e non solo)
- facilitare test e manutenzione
- ...

FUNZIONI

Formalmente:

```
return_type nome_della_funzione( lista_degli_argomenti ) {  
  
    corpo_della_funzione;  
  
    return qualche_valore;    // di tipo return_type  
  
}
```

Gli argomenti della funzione possono essere dichiarati con opportuni qualificatori:

```
float inverso(int const x) {  
    return 1.0f / (float) x;  
}
```


FUNZIONI

Es:

```
int max(int x, int y) {  
    if (x > y)  
        return x;  
    else  
        return y;  
}
```

[...]

```
int a = max(10, 33);           // a = 33  
int b = max(20, -2);          // b = 20
```

FUNZIONI

Es:

```
void incrementoDiDue( int x ) {  
    x = x + 2;  
}
```

[...]

```
int c = 0;  
for (int i = 0; i < 10; i++) {  
    incrementoDiDue( c );  
    std::cout << c << std::endl;  
}
```

FUNZIONI

Es:

```
void incrementoDiDue( int x ) {  
    x = x + 2;  
}
```

[...]

```
int c = 0;  
for (int i = 0; i < 10; i++)  
    std::cout << incrementoDiDue( c ) << std::endl;
```

Il valore di c non cambia mai. Paradigma "call-by-value"

FUNZIONI: VALUE AND REFERENCE

// call-by-value (send the function a copy of the argument's value)

```
int f(int a) { a = a+1; return a; }
```

```
int main()
```

```
{
```

```
    int xx = 0;
```

```
    cout << f(xx) << '\n'; // writes 1
```

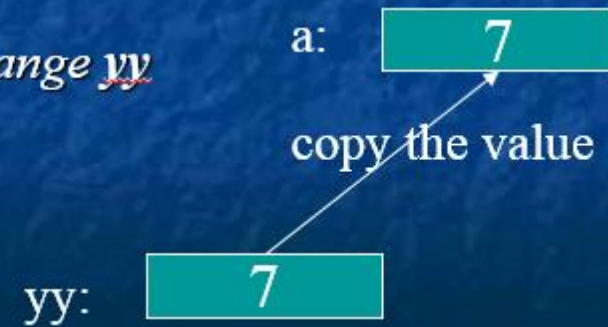
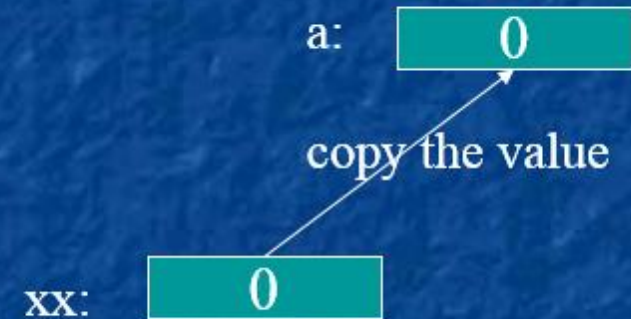
```
    cout << xx << '\n';    // writes 0; f() doesn't change xx
```

```
    int yy = 7;
```

```
    cout << f(yy) << '\n'; // writes 8; f() doesn't change yy
```

```
    cout << yy << '\n';    // writes 7
```

```
}
```



FUNZIONI: VALUE AND REFERENCE

// call-by-reference (pass a reference to the argument)

```
int f(int& a) { a = a+1; return a; }
```

```
int main()
```

```
{
```

```
  int xx = 0;
```

```
  cout << f(xx) << '\n'; // writes 1
```

// f() changed the value of xx

```
  cout << xx << '\n'; // writes 1
```

```
  int yy = 7;
```

```
  cout << f(yy) << '\n'; // writes 8
```

// f() changes the value of yy

```
  cout << yy << '\n'; // writes 8
```

```
}
```

a:

1st call (refer to xx)

xx:

0

2nd call (refer to yy)

yy:

7

CALL-BY-REFERENCE

E' una "novità" rispetto a linguaggio C:

```
int f( int& a ) {  
    a = a + 1;  
    return a;  
}
```

```
int f( int *a ) {  
    *a = *a + 1;  
    return *a;  
}
```

Per chi è abituato ad usare i puntatori, call-by-reference appare strana, perché:

- sintassi da passaggio per valore
- semantica da passaggio per puntatore

CALL-BY-REFERENCE

Call-by-reference e call-by-value sono entrambe legali ed ampiamente accettate, ma cosa è meglio fare?

Andiamo a chiedere a chi è più bravo di noi...

Google C++ Style Guide (<https://google.github.io/styleguide/cppguide.html>)

CALL-BY-REFERENCE

C++ Style Guide (<https://google.github.io/styleguide/cppguide.html>)

All parameters passed by reference must be labeled `const`.

Definition:

In C, if a function needs to modify a variable, the parameter must use a pointer, eg `int foo(int *pval)`. In C++, the function can alternatively declare a reference parameter: `int foo(int &val)`.

Pros:

Defining a parameter as reference avoids ugly code like `(*pval)++`. Necessary for some applications like copy constructors. Makes it clear, unlike with pointers, that a null pointer is not a possible value.

Cons:

References can be confusing, as they have value syntax but pointer semantics.

Decision:

Within function parameter lists all references must be `const`:

```
void Foo(const string &in, string *out);
```

In fact it is a very strong convention in Google code that input arguments are values or `const` references while output arguments are pointers. Input parameters may be `const` pointers, but we never allow non-`const` reference parameters except when required by convention, e.g., `swap()`.

However, there are some instances where using `const T*` is preferable to `const T&` for input parameters. For example:

- You want to pass in a null pointer.
- The function saves a pointer or reference to the input.

Remember that most of the time input parameters are going to be specified as `const T&`. Using `const T*` instead communicates to the reader that the input is somehow treated differently. So if you choose `const T*` rather than `const T&`, do so for a concrete reason; otherwise it will likely confuse readers by making them look for an explanation that doesn't exist.

CALL-BY-REFERENCE

```
void f( int a, int& b, const int& c ) {  
    ++a; ++b; ++c;  
}
```

CALL-BY-REFERENCE

```
void f( int a, int& b, const int& c ) {  
    ++a; ++b; ++c;           // Errore: c é const  
}
```

```
void g( int a, int& b, const int& c ) {  
    ++a; ++b; int cc = c; ++cc;  
}
```

CALL-BY-REFERENCE

```
void f( int a, int& b, const int& c ) {  
    ++a; ++b; ++c;  
}
```

```
void g( int a, int& b, const int& c ) {  
    ++a; ++b; int cc = c; ++cc;  
}
```

```
int main() {  
    int x = 0, y = 0, z = 0;  
    g(x, y, z);  
    g(1, 2, 3);  
    g(1, y, 3);  
}
```

CALL-BY-REFERENCE

```
void f( int a, int& b, const int& c ) {  
    ++a; ++b; ++c;  
}  
  
void g( int a, int& b, const int& c ) {  
    ++a; ++b; int cc = c; ++cc;  
}  
  
int main() {  
    int x = 0, y = 0, z = 0;  
    g(x, y, z);      // x == 0, y == 1, z == 0  
    g(1, 2, 3);      // errore: b vuole una reference  
    g(1, y, 3);      // ok, c è const, posso passare un "temporary"  
}
```