# Il linguaggio C++

## "A better C"

Il linguaggio C++ nasce presso i laboratori di ricerca della AT&T Bell (Murray Hill, NJ) a partire dal 1979, ad opera di Bjarne Stroustrup. Rappresenta una estensione del linguaggio di programmazione C. Per questo motivo, un compilatore C++ è in grado di compilare anche programmi C.

Le principali estensioni sono:

- ✓ alcuni miglioramenti rispetto al linguaggio C;
- ✓ un supporto per la gestione dei tipi di dato astratto;
- ✓ un supporto per la programmazione orientata agli oggetti.

## Evoluzione del C++

- 1979: inizio delle ricerche di Bjarne Stroustrup presso i laboratori di ricerca della AT&T Bell (Murray Hill, NJ)
- 1983: Bjarne Stroustrup, *"Adding classes to C: an excercise in language evolution"*, Software Practice and Experience 13
- 1983: il linguaggio denominato "C with classes" assume il nome di C++, coniato da Rick Mascitti (collaboratore di Stroustrup).
- 1986: Bjarne Stroustrup, *"The C++ Programming Language"*, Addison-Wesley
- 1990: Margaret Ellis, Bjarne Stroustrup, *"The Annotated C++ Reference Manual"*, Addison-Wesley
  → al linguaggo vengono aggiunti i template
- 1994: nasce la Standard Template Library (STL), ad opera di Alexander Stepanov et al.
- 1998: standardizzazione internazionale ANSI/ISO. Questa versione del linguaggio assume la denominazione *Standard C++*.

# Il primo programma C++

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

int main() {
    // L'esecuzione ha inizio dalla funzione main
    cout << "Il mio primo programma in C++";
    return 0;
}
```

⇒ La direttiva `#include <iostream>` include la libreria delle funzioni predefinite di input/output tipiche del C++. Fra queste, la funzione **cout** permette di inviare del testo allo standard output ("console output").
Analogamente esiste anche la funzione **cin** che permette di leggere dallo standard input ("console input").

⇒ L'istruzione `using namespace std` indica al compilatore di utilizzare il namespace std.
Il **namespace** è un costrutto C++ che permette di localizzare i nomi

degli identificatori al fine di evitare conflitti sui nomi di variabili, funzioni e classi. Attraverso questa indicazione, indichiamo al compilatore che tutti i nomi vanno ricercati all'interno del namespace std. Senza questa istruzione, per specificare che il nome cout appartiene al namespace std avremmo dovuto scrivere:

```
std::cout << "Il mio primo programma in C++";
```

Nota: l'intera Standard Library del C++ è definita all'interno del namespace std.

# Un semplice programma C++

```
// Programma per il calcolo del fattoriale
#include <iostream>
using namespace std;

int Fatt(int n);  // prototipo della funzione Fatt

int Fatt(int n) {
    /* funzione che calcola il fattoriale del numero in ingresso */
    int fat;
    fat = 1;
    while (n >= 1) {
        fat = fat * n;
        n = n - 1;
    }
    return fat; // valore restituito
}

main() {
    int i;
    cout << "Calcolo del fattoriale di :";
    cin >> i;
    cout << "Risultato = ";
    cout << Fatt(i);
}
```

## Alcune caratteristiche del linguaggio

Un programma C++ ha diverse componenti:

- **direttive** → iniziano con il simbolo #
- **definizioni** → di variabile o altro
- **funzioni** → con parametri formali (notare la funzione speciale **main**)
- **classi** (le vedremo più avanti)

Le parentesi graffe delimitano un blocco; ciò che è dichiarato in un blocco è visibile solo all'interno del blocco.

Esistono due modi di specificare i commenti:

- sono racchiusi tra /* e */
- oppure un commento comincia con // e si estende fino al termine della riga

## Espressioni e istruzioni in C++

Una espressione in C++ può essere una costante, una variabile una espressione composta
Una espressione composta è formata da un operatore e da uno o due operandi che sono a loro volta espressioni.

Alcuni operandi di C++:

- Aritmetici: `* / + -`
- Logici: `&& || ! < <= == != >= >`
- Assegnamento: `= *= += -= /=`
- Incremento/decremento: `++ --`
- If aritmetico: `? :`
- Una istruzione semplice è una espressione seguita da ";"
- Una istruzione composta è una sequenza di istruzioni racchiuse dai simboli "{" e "}"

## Istruzioni di controllo in C++

Sono praticamente le stesse del C. Ricordiamo alcune istruzioni tra le più usate:

*Istruzioni condizionali:*

```
if ( <espressione> ) <istruzioni1> else <istruzioni2>

switch ( <espressione> ) {
    case <costante1> : <istruzioni1>
    case <costante2> : <istruzioni2>
    ....
    default : <istruzioni>
}
```

*Cicli:*

```
while ( <espressione> ) <istruzioni>

do <istruzioni> while ( <espressione> )

for ( <espr1> ; <espr2> ; <espr3> ) <istruzioni>
```

(è possibile dichiarare una variabile in <espr1>)

# Un'altra versione del programma per il calcolo del fattoriale

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

int Fatt(int n);

int Fatt(int n) {
    int fat = 1; // inizializzazione all'atto della definizione
    while (n >= 1) {
        fat *= n;
        n--;
    }
    return fat;
}

main() {
    int i;
    cout << "Calcolo del fattoriale di :";
    cin >> i;
    cout << "Risultato = " << Fatt(i) << endl;
    /* occorrenze successive degli operatori di input/output si
    possono concatenare
    endl = new line */
}
```

## Tipi di dato base in C++

Sono gli stessi del C con alcune varianti.

Tipi primitivi principali: `int short long float double long double char bool`

Esempio dichiarazione variabili:

```
int x;
float a, b=1.34;
bool trovato = false;
char carattere = 'b';
```

Tipi enumerati: **enum**

```
enum mese {gennaio, febbraio, marzo, ... }
mese m = marzo;
```

## Regole di scope

Lo **scope** (visibilità) di una variabile è la porzione di codice in cui la variabile dichiarata è riferibile (utilizzabile)
Regola: lo scope di una variabile o di un qualsiasi altro identificatore si estende dal punto immediatamente successivo la dichiarazione fino alla fine del blocco di istruzioni (delimitato dalle parentesi graffe) in cui è inserita.

```
// Qui x non è visibile
...
{
    ...
    // qui x non è visibile
    int x = 5; // Da ora in poi esiste una variabile x
    ... // qui x è visibile
    { // x e` visibile anche in questo blocco
        ...
    }
    ...
} // da questo punto x ora non è più visibile
  // (si dice che x è out of scope)
```

## Altre regole di visibilità

All'interno di uno stesso blocco non è possibile dichiarare più volte lo stesso identificatore, ma è possibile ridichiararlo in un blocco annidato; in tal caso la nuova dichiarazione nasconde quella più esterna che ritorna visibile non appena si esce dal blocco ove l'identificatore viene ridichiarato

```
{ // blocco A
    int x = 5, y = 0;       // variabili locali al blocco A
                            //(e ai blocchi nidificati in A)
    cout << x << endl;      // stampa 5
    while (y++ < 3) {       // blocco B
        cout << x << ' ';   // stampa 5
        char x = '#';
        cout << x << ' ';   // ora stampa #
    }
    cout << x << endl;      // stampa di nuovo 5
}
```

## Variabili globali

Le variabili globali sono note in tutto il programma, possono essere utilizzate in ogni parte del codice e mantengono il loro valore per tutta l'esecuzione del programma: il loro scope si estende all'intero programma.
Le variabili globali vengono create con una dichiarazione che è posta al di fuori di tutte le funzioni del programma.
Se all'interno di un programma una variabile locale ed una globale hanno lo stesso nome, tutti i riferimenti sono intesi alla variabile locale.

La memorizzazione delle variabili globali avviene in una regione fissa della memoria appositamente allocata. Le variabili globali sono di aiuto quando parecchie funzioni del programma fanno riferimento agli stessi dati. Ma il loro utilizzo è sconsigliato per tre motivi:

1. Occupano la memoria per tutta la durata dell'esecuzione del programma, non solo quando servono.
2. L'uso di molte variabili globali rende il programma incline agli errori e difficile da debuggare, oltre che poco leggibile.

3. L’uso di una variabile globale al posto di una locale rende una funzione meno generale, in quanto si affida a qualcosa che deve essere definito al suo esterno.

→ le variabili globali sono l’opposto della information hiding!

## Cast - Conversione del tipo dei dati

Il tipo di un dato può essere convertito implicitamente (con una assegnazione) o esplicitamente (con l'operazione di **cast**).
Esempio:

```
int x=3;
float y = 5.3;
x = int(y); // oppure
x = (int)y;
y = x;
```

## Costanti in C++

Si definiscono facendo precedere la parola chiave **const** dalla definizione di una variabile.  La costante deve essere inizializzata.
Esempio:

```
const float PIGRECO = 3.14;
const int N = 100;
```

## Alias o riferimenti

Sono riferimenti (*references*) ad una una variabile già definita.
Vengono definiti facendo uso del carattere speciale **&** nella dichiarazione.

Esempio:

```
int x;
int& y = x;
x = 3;
y = y++;
cout << x; // stampa 4
```

## Dati strutturati in C++

E' possibile definire dati strutturati con il tipo **struct**
Per un dato strutturato è possibile definire anche operazioni da effettuare su di esso.

```
struct data {
    int giorno;
    int mese;
    int anno;
    void stampa() {
        cout << giorno << "/" << mese << "/" << anno
    }
};

...
data oggi;
oggi.giorno = 6; oggi.mese = 3; oggi.anno= 2002;
oggi.stampa();
```

## Puntatori

Un *puntatore* è una variabile che contiene un riferimento ad un'altra variabile. Da un punto di vista fisico un puntatore è l'indirizzo di una locazione di memoria.
Ad esempio, se il puntatore x contiene l'indirizzo di y, allora si dice che x *punta* a y. L'utilizzo di dati che puntano ad altri dati è detto *indirezione*.

Definizione di un puntatore
Analogamente al C, in C++ un puntatore si dichiara anteponendo al nome di una variabile il simbolo * (*star*).

Esempi di definizione di puntatori:

```
int *i;
char *c;
float *n;
float *n1, *n2;    // definizione multipla
int* j;            // diverso stile
int* a, b;         // attenzione!
```

## Operazioni sui puntatori

L'operatore **&** applicato ad una variabile restituisce il puntatore ad essa.
L'operatore ***** applicato a un puntatore restituisce la variabile puntata.

Esempi:

```
int i, j;
int *p,*q;
p = &i;     // p punta alla variabile i
j = p;      /* nella variabile j va il contenuto della variabile
puntata da p */
*q = j;     // nella variabile puntata da q va il contenuto di j
*p = *q;    /* nella variabile puntata da p va il contenuto della
variabile puntata da q */
p = q;      // p e q puntano alla stessa variabile
```

## Array in C++

Un *array* è una collezione di variabili dello stesso tipo, a cui si fa riferimento con un nome comune. E' possibile accedere ad uno specifico elemento di un array mediante un *indice*.
Gli array possono avere più dimensioni.

### Definizione di array

```
int vettore[10];       // vettore di interi (10 elementi)
int matrice[2][3];     // matrice 2X3 di interi
for (int i=0; i<10; i++) vettore[i] = 0; // gli indici vanno da 0
                                         // a N-1
matrice[1][2] = 3;
float x[3] = {1.0, 2.3, 3.4 } // Un array può essere inizializzato
                              // all'atto della sua creazione
float y[] = {1.0, 3.3, 5.0, 6.3 } // In questo caso la sua
                                  // dimensione viene calcolata
                                  // automaticamente
```

## Array e puntatori

Esiste uno stretto legame tra array e puntatori.
Definendo:

```
float x[4];
```

x denota un puntatore al primo elemento dell'array; incrementando x di un intero k si accede alla posizione k-esima dell'array (ovvero all'elemento di indice k+1).

Quindi:

`*x` e `x[0]` sono la stessa cosa,
`x` e `&x[0]` sono la stessa cosa,
`*(x + 3)` e `x[3]` sono la stessa cosa,
`(x + 2)` e `&x[2]` sono la stessa cosa

## Gestione della memoria a runtime

Durante l'esecuzione di un programma C++ (*runtime*) la memoria viene gestita dal sistema in due maniere:

**gestione statica**: viene allocata dal sistema operativo un'area di memoria di dimensione fissa per tutta l'esecuzione del programma (variabili globali)

**gestione dinamica**: vengono allocate due aree di memoria di dimensioni variabili che vengono usate in maniera diversa:

- lo *stack*: quando una funzione viene invocata vengono allocate automaticamente tutte le variabili locali e i parametri attuali sullo stack in un *record di attivazione*; successivamente, quando l'esecuzione della funzione termina, il record di attivazione viene cancellato e lo stack viene riportato nella situazione in cui era prima dell'invocazione;
- lo *heap* o *free store*: la gestione viene lasciata al programmatore mediante creazione e distruzione dinamica di variabili (tramite puntatori)

## Dichiarazione di funzioni

In C++ la dichiarazione di funzioni è obbligatoria: l'utilizzo di una funzione deve essere preceduto dal suo *prototipo*.
Si tratta di una differenza importante rispetto al C, nel quale è possibile invocare una funzione senza averla prima dichiarata: il compilatore C assume che la funzione chiamata ritorni un valore di tipo int ed abbia parametri tutti di tipo int (i migliori compilatori C emettono un warning).

I prototipi di funzione specificano tre aspetti di una funzione:

- Il tipo restituito
- Il numero dei suoi parametri
- Il tipo dei suoi parametri

## Funzioni: passaggio parametri

Quando viene richiamata una funzione, si riserva spazio di memoria per i suoi *parametri formali*, ciascuno dei quali viene inizializzato secondo il corrispondente *parametro attuale*.
In C++ i parametri ad una funzione possono essere passati:

- ✓ per valore
- ✓ per puntatore
- ✓ per riferimento (*reference*)
- ✓ per riferimento costante (*const reference*)

Per default il C++ adotta il passaggio di parametri per valore. Con questa modalità il parametro passato non viene modificato dalla funzione chiamata: si crea una copia del parametro e la funzione opera su tale copia.
Nel caso in cui sia necessario cambiare il valore del parametro passato, analogamente al C è possibile utilizzare parametri di tipo puntatore.
Il C++ mette a disposizione anche il passaggio di parametri per riferimento (specificato facendo precedere il parametro formale dal simbolo **&**): anche in questo caso il parametro passato può essere

modificato dalla funzione. In altri termini possiamo dire che la funzione opera sul parametro stesso, non su una copia del parametro.

## Esempi di passaggio parametri a funzioni

```
int passaPerValore();
int passaPerPuntatore();
int passaPerReference();
int passaPerConstReference();
void f1(int r);              // per valore
void f2(int *r);             // per puntatore
void f3(int& r);             // per reference
void f4(const int& r);       // per const reference

int main() {
    // confrontiamo i diversi tipi di passaggio parametri
    passaPerValore();
    passaPerPuntatore();
    passaPerReference();
    passaPerConstReference();
    return 0;
}
```

```
int passaPerValore() {
    int x = 47;
    cout << "x = " << x << endl;
    cout << "&x = " << &x << endl;
    f1(x);     // E' pass-by-value
    cout << "x = " << x << endl;   // quanto vale x?
    return 0;
}

void f1(int r) {
  cout << "r = " << r << endl;
  cout << "&r = " << &r << endl;
  r = 5;       // cambiamo il valore del parametro!
  cout << "r = " << r << endl;
}
```

```
int passaPerPuntatore()
{
    int x = 47;
    cout << "x = " << x << endl;
    cout << "&x = " << &x << endl;
    f2(&x);    // passiamo l'indirizzo di x
    cout << "x = " << x << endl;   // quanto vale x?
    return 0;
}

void f2(int *r) {
  cout << "r = " << r << endl;
  cout << "*r = " << *r << endl;
  *r = 5;      // cambiamo il valore del parametro!
               // (il puntatore viene dereferenziato)
  cout << "r = " << r << endl;
}
```

```
int passaPerReference()
{
    int x = 47;
    cout << "x = " << x << endl;
    cout << "&x = " << &x << endl;
    f3(x);      // Sembra un passaggio per valore,
                // in realtà è per reference
    cout << "x = " << x << endl;     // quanto vale x?

    return 0;
}

void f3(int& r) {
  cout << "r = " << r << endl;
  cout << "&r = " << &r << endl;
  r = 5;        // cambiamo il valore del parametro!!
  cout << "r = " << r << endl;
}
```

```
int passaPerConstReference()
{
    int x = 47;
    cout << "x = " << x << endl;
    cout << "&x = " << &x << endl;
    f4(x);     // Passaggio per const reference
    cout << "x = " << x << endl;
    return 0;
}

void f4(const int& r) {
  cout << "r = " << r << endl;
  cout << "&r = " << &r << endl;
  // r = 5;   <-- non compila perchè cambia il valore di r,
                // dichiarato const
  cout << "r = " << r << endl;
}
```

Alcune considerazioni

Sia il passaggio per puntatore che per riferimento consentono di modificare il valore del parametro. Tuttavia usando i puntatori:

- all'interno della funzione chiamata si è costretti ad eseguire tutte le operazioni attraverso puntatori (eventualmente dereferenziandoli)
- nelle funzioni chiamanti è necessario passare gli indirizzi degli argomenti, invece dei valori

→ è piuttosto scomodo e poco leggibile!

Utilizzando il passaggio per reference il compito di passare l'indirizzo dell'argomento invece che il suo valore è lasciato al compilatore, a tutto vantaggio della leggibilità.

Se si vuole essere certi che la funzione chiamata non modifichi il valore del parametro, è possibile utilizzare sia il passaggio per valore che per const reference.
Usando il passaggio per valore, viene creato spazio in memoria per una copia del parametro, e viene effettuata una operazione di copia del

parametro.  Se il parametro è un array di grandi dimensioni questa operazione è largamente inefficiente!
→ è preferibile utilizzare il passaggio per const reference: in questo modo il parametro viene passato per indirizzo e non viene effettuata nessuna copia, e il controllo sulla non modificabilità del parametro è ad opera del compilatore.

## Valori di default per una funzione

Nella dichiarazione di una funzione è possibile specificare il valore di default di uno più parametri.

```
void swapVett (int* v1, int* v2, int size, int skip = 0);

void swapVett (int* v1, int* v2, int size, int skip /* = 0 */) {
    int a;
    for (int i = skip; i < size; i++) {
        a = v1[i];
        v1[i] = v2[i];
        v2[i] = a;
    }
}

...

int v1[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int v2[10] = {50, 21, 34, 11, 58, 99, 101, 33, 85, 12};
swapVett(v1, v2, 10, 4);      // skip primi 4 elementi
swapVett(v1, v2, 10);         // swap degli interi vettori
```

## Ricorsione (o Recursione)

In C++ è possibile definire funzioni che richiamano se stesse, dette funzioni *ricorsive* (o *recursive*).

```
int Fatt(int n) {
    if (!n) return 1;
    else return (n * Fatt(n-1));
}

int main() {
    int fat, n=0;
    cin >> n;
    fat = Fatt(n);
    return 0;
}
```

La ricorsione viene gestita attraverso l'uso opportuno dello stack: ogni chiamata della funzione ricorsiva genera un nuovo record di attivazione (con memorizzazione dei parametri attuali e allocazione di memoria delle variabili locali) che "maschera" la precedente invocazione.
Le funzioni ricorsive consentono di scrivere codice estremamente compatto e di creare versioni molto chiare e semplici di molti algoritmi.

Tuttavia troppe chiamate ricorsive ad una funzione possono causare la saturazione dello stack (*stack overflow*).

## Funzioni inline

Una funzione in C++ può essere dichiararata *inline*.
In questo caso ad ogni invocazione della funzione il codice viene direttamente inserito nel codice oggetto, senza utilizzare il meccanismo standard di trasferimento di controllo (creazione del record di attivazione, inserimento nello stack...). Si dice che la funzione viene *espansa inline*.

Esempio:

```
inline int max(int a, int b) {
    return( a > b ? a : b);
}
```

E' un meccanismo che si usa quando si devono realizzare funzioni semplici e non si vuole appesantire con esse il tempo di esecuzione del programma.
La specifica inline è un "suggerimento" al compilatore C++.

## Overload di funzioni

In C++ due o più funzioni possono condividere lo stesso nome a condizione che le loro dichiarazioni di parametri siano diverse. Questa caratteristica è denominata *function overloading*.

```
// L'overload di funzione avviene tre volte
#include <iostream>
using namespace std;

void f(int i);         // parametro intero
void f(int i, int j); // due parametri interi
void f(double k);      // un parametro double

int main()
{
    f(10);      // chiama f(int)
    f(10, 20); // chiama f(int, int)
    f(12.23);  // chiama f(double)
    return 0;
}
```

E’ necessario fare attenzione all’ambiguità: in alcune situazioni il compilatore non è in grado di scegliere fra due o più funzioni che sono state (correttamente) sottoposte ad overload → viene generato un errore in compilazione.
L’ambiguità è causata principalmente dalla conversione automatica di tipo.

```
// Ambiguità nell’overload.
#include <iostream>
using namespace std;

float myfunc(float i);
double myfunc(double i);

int main()
{
  // senza ambiguità, chiama myfunc(double) perché per default tutte
  // le costanti in virgola mobile sono convertite in double
  cout << myfunc(10.1) << " ";

  // ambigua: il compilatore non sa se convertire in float o double
  cout << myfunc(10);

  return 0;
}
```

## Organizzazione dei programmi C++

In C++ un programma è generalmente distribuito su più file;  ogni file costituisce un componente (modulo) dell'applicazione.
Ogni modulo è solitamente decomposto in due file:

- Le dichiarazioni vengono poste in file detti *header* che hanno estensione ".h"
  → *con tale file si dichiarano i servizi offerti dal modulo*
- Le definizioni vengono poste in file che hanno estensione ".C" o ".cpp"
  → *tale file costituisce l'effettiva implementazione dei servizi*

## Struttura di uno header file

```
// File nomefile.h
#ifndef NOMEFILE_H
#define NOMEFILE_H

< sequenza di dichiarazioni di costanti >
< sequenza di dichiarazioni di tipi >
< sequenza di dichiarazioni di variabili >
< sequenza di dichiarazioni di funzioni >

#endif
```

L'identificatore NOMEFILE_H è solo un nome che associamo alle dichiarazioni contenute nel file.

## Comunicazione tra componenti separate

Quando si vuole usare in un file A funzionalità (funzioni, dati e classi) definite altrove, si *includono* in A le loro **dichiarazioni** (contenute nei file header) e si rendono così visibili all'interno di del file A.
Per includere un file in un altro si usa la direttiva di compilazione:

**#include <..>** (librerie di sistema)
oppure:
**#include "../"** (librerie definite dal programmatore)

In questa maniera si disaccoppia l'uso di una componente dalla sua effettiva implementazione.
Questo consente tra l'altro la compilazione separata delle componenti; il **linker** ha poi il compito di associare nella maniera corretta le invocazioni alle definizioni.

# Organizzazione di programmi su più file

alfa.h

```
#ifndef ALFA_H
#define ALFA_H
......
int f(char a);      //dichiarazione di f
......
#endif
```

mioProgramma.cpp

```
#include "alfa.h"
…
main() {
 …
 a = f(b);          // uso di f
 …
}
```

## alfa.cpp

```
#include <iostream.h>
#include "alfa.h"
…
int f(char a) { // implementazione di f
    …
}
```

## Utilizzo di sorgenti C e C++

Spesso nella stesura di programmi C++ capita di compilare sorgenti in C insieme ai sorgenti in C++, lasciando al linker il compito di unire i simboli definiti nei vari sorgenti.
Come già visto, il compilatore C++ è in grado di gestire l'overload di funzioni, perché internamente il simbolo di una funzione viene *decorato* con informazioni derivanti da numero e tipo dei parametri formali.
Ad esempio, una funzione

```
float f(int a, char b);
```

viene decorata dal compilatore C++ in un simbolo che generalmente è nella forma **_f_int_char**.
La stessa funzione, scritta e compilata in C, assume il simbolo **_f**, senza *name decoration*, dal momento che il compilatore C non supporta l'overloading.
La funzione f assume simboli diversi nella funzione chiamante e nella sua implementazione, e il risultato è un errore di link (*unresolved symbol*).

Quindi come è possibile far convivere sorgenti C e C++, in modo che si possa inserire la chiamata ad una funzione C all'interno di codice C++? Nel file header dove è contenuto il prototipo di f, è necessario fare uso della specifica di compilazione **extern "C"**, in modo da indicare al compilatore che il nome della funzione non deve essere decorato.

```
extern "C" float f(int a, char b);
```

In questo modo anche il compilatore C++ assegna alla funzione il simbolo **_f**, ed è possibile chiudere il link senza errori.

E' possibile anche definire un gruppo di dichiarazioni con *alternate linkage C* :

```
extern "C" {
  float f(int a, char b);
  double d(int a, char b);
}
```

In alternativa, è possibile fare uso anche della preprocessor macro **__cplusplus**.

# Esempio 1

## beta.cpp

```
#include <iostream>

extern "C" {
#include "gamma.h"
}
…
int prova(char a) {
    …
    int val = f(2, 'A');
    …
}
```

## gamma.c

```
#include "gamma.h"

float f(int a, char b) {
     …
     return 0.0;
}
```

## gamma.h

```
…
float f(int a, char b);
…
```

# Esempio 2

## beta.cpp

```
#include <iostream>
#include "gamma.h"
…
int prova(char a) {
    …
    int val = f(2, 'A');
    …
}
```

## gamma.c

```
#include "gamma.h"

float f(int a, char b) {
    …
    return 0.0;
}
```

## gamma.h

```
#ifdef __cplusplus
extern "C" {
#endif

float f(int a, char b);

#ifdef __cplusplus
}
#endif
```

NB: la macro `__cplusplus` dipende dal compilatore.

## Dipendenze fra files

Ogni volta che viene modificato qualcosa in uno header file è necessario ricompilare tutti i file sorgente che lo includono. Questo perché potrebbe essere stato modificato il prototipo di una funzione, e bisogna essere certi che tutte le chiamate alla funzione siano coerenti con la sua dichiarazione.

Ma come si può essere certi di ricompilare tutti i file sorgenti? Una soluzione è ricompilare l'intero applicativo → potrebbe essere un'operazione molto onerosa.
Gli ambienti di sviluppo integrati C++ (*IDE – Integrated Development Environment*) tengono traccia delle *dipendenze*, ossia dell'albero dei legami fra files (a include b che include c…), e ad ogni modifica di uno header file provvedono automaticamente a ricompilare tutti i sorgenti che dipendono da esso.