Capitolo 1

Fondamenti

Per prima cosa, uccidiamo
tutti gli avvocati delle lingue.
– Enrico VI, Parte II

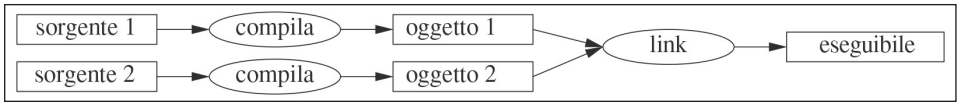
In questo capitolo

1.1 Introduzione

In questo capitolo saranno presentati in modo informale la notazione di C++, il suo modello di memoria e computazione, e il meccanismo di base per organizzare il codice in un programma. Questi sono gli elementi del linguaggio che supportano gli stili visti soprattutto in C e talvolta chiamati *programmazione procedurale.*

1.2 Programmi

C++ è un linguaggio compilato. Per poter eseguire un programma, il suo testo sorgente deve essere elaborato da un compilatore per produrre oggetti, che saranno poi combinati da un linker per produrre un eseguibile. Un programma in C++ è costituito tipicamente da molti file di codice sorgente (di solito chiamati semplicemente *file sorgente*).



Un programma eseguibile viene creato per una specifica combinazione hardware/sistema, e non non risulta portabile, per esempio, da un dispositivo Android a un PC Windows. Quando parliamo di portabilità dei programmi C++, generalmente ci riferiamo alla portabilità del codice sorgente, nel senso che il codice sorgente può essere compilato ed eseguito con successo su vari sistemi.
Lo standard ISO C++ definisce due tipi di entità:

- *caratteristiche interne del linguaggio*, quali i tipi incorporati (per esempio, `char` e `int`) e i cicli (per esempio, istruzioni `for` e istruzioni `while`);
- *componenti della libreria standard*, quali i contenitori (per esempio, `vector` e `map`) e le operazioni di I/O (per esempio, `<<` e `getline()`).

I componenti della libreria standard sono normalissimo codice C++ fornito da qualsiasi implementazione di C++. Cioè, la libreria standard C++ può essere implementata in C++ e lo è (con un impiego minimo di codice macchina per operazioni quali il cambio di contesto dei thread). Ciò implica che C++ sia abbastanza espressivo ed efficiente per le attività di programmazione dei sistemi più esigenti.
C++ è un linguaggio con tipi statici: il tipo di ogni entità (per esempio, oggetto, valore, nome ed espressione) deve essere noto al compilatore e determina il set di operazioni applicabile a essa e il suo layout in memoria.

1.2.1 Hello, World!

Il programma C++ più piccolo è

```
int main() { }   // il programma C++ più piccolo
```

Definisce una funzione denominata `main`, che non accetta argomenti e non fa nulla.
Le parentesi graffe, `{ }`, esprimono il raggruppamento in C++, e qui indicano l'inizio e la fine del corpo della funzione. La doppia barra, `//`, apre un commento che si estende fino alla fine della riga; il commento è per il lettore umano; il compilatore ignora i commenti.
Ogni programma in C++ deve avere esattamente una funzione globale denominata `main()`. Il programma inizia eseguendo tale funzione. Il valore intero `int` restituito da `main()`, se esiste, è il valore restituito dal programma al "sistema". Se non viene restituito alcun valore, il sistema riceverà un valore che indica il completamento dell'esecuzione. Un valore non zero proveniente da `main()` indica il fallimento. Non tutti i sistemi operativi e gli ambienti di esecuzione usano questo valore restituito: gli ambienti basati su Linux/Unix lo fanno, mentre quelli basati su Windows lo fanno solo raramente.
Di solito, un programma produce un output. Ecco un programma che scrive `Hello, World!`:

```
import std;

int main()
{
    std::cout << "Hello, World!\n";
}
```

La riga `import std;` istruisce il compilatore a rendere disponibili le dichiarazioni della libreria standard. Senza queste dichiarazioni, l'espressione

```
std::cout << "Hello, World!\n"
```

non avrebbe senso. L'operatore `<<` ("put to") scrive il suo secondo argomento sul primo. In questo caso, il letterale stringa `"Hello, World!\n"` viene scritto sul flusso standard di output `std::cout`. Un letterale stringa è una sequenza di caratteri racchiusa tra doppie virgolette, e al suo interno il carattere barra inversa `\` seguito da un altro carattere denota un singolo "carattere speciale". In questo caso, `\n` è il carattere newline, e questo fa sì che i caratteri scritti siano `Hello, World!` seguiti da un newline.
`std::` specifica che il nome `cout` si trova nel namespace della libreria standard [§3.3]. Di solito tralascio `std:` nella trattazione delle caratteristiche standard; §3.3 mostra come rendere visibili i nomi di un namespace senza alcuna qualificazione esplicita.
La direttiva `import` è una novità di C++20 e introdurre tutta la libreria standard come modulo `std` non è ancora standard, come verrà spiegato in §3.2.2. Se avete problemi con `import std;`, provate il vecchio e convenzionale

```
#include <iostream>        // include le dichiarazioni per la libreria di flussi I/O

int main()
{
    std::cout << "Hello, World!\n";
}
```

Questo verrà spiegato in §3.2.1 e funziona su tutte le implementazioni C++ dal 1998 [§19.1.1].
Essenzialmente tutto il codice eseguibile viene inserito nelle funzioni e chiamato direttamente o indirettamente da `main()`. Per esempio:

```
import std;              // importa le dichiarazioni per la libreria standard

using namespace std;     // rende visibili i nomi di std senza usare std:: [§3.3]

double square(double x) // eleva al quadrato un numero in virgola mobile a doppia precisione
{
    return x*x;
}

void print_square(double x)
{
    cout << "the square of " << x << " is " << square(x) << "\n";
}

int main()
{
    print_square(1.234);  // stampa: il quadrato di 1.234 è 1.52276
}
```

Un "tipo restituito" `void` indica che una funzione non restituisce alcun valore.

1.3 Funzioni

Il modo principale per ottenere qualche risultato in un programma C++ è chiamare una funzione. La definizione di una funzione consente di specificare come dev'essere eseguita un'operazione. Non è possibile chiamare una funzione che non sia stata dichiarata prima. La dichiarazione di una funzione enuncia il nome della funzione, il tipo di valore restituito (se esiste), e il numero e i tipi di argomenti che devono essere forniti in una chiamata. Per esempio:

```
Elem* next_elem();    // nessun argomento; restituisce un puntatore a Elem (an Elem*)
void exit(int);       // argomento int; non restituisce nulla
ouble sqrt(double);   // argomento double; restituisce un double
```

Nella dichiarazione di una funzione, il tipo restituito viene prima del nome della funzione, mentre i tipi degli argomenti vengono dopo il nome racchiuso tra parentesi.
La semantica del passaggio degli argomenti è identica a quella dell'inizializzazione [§3.4.1]: i tipi degli argomenti vengono controllati, e quando è necessario avviene la conversione implicita del tipo degli argomenti [§1.4]. Per esempio:

```
double s2 = sqrt(2);          // chiama sqrt() con l'argomento double{2}
double s3 = sqrt("three");    // errore: sqrt() richiede un argomento di tipo double
```

Il valore del controllo e della conversione di tipo in fase di compilazione non va sottovalutato.
La dichiarazione di una funzione può contenere i nomi degli argomenti. Questo può essere di aiuto al lettore di un programma, ma a meno che la dichiarazione sia anche la definizione della funzione, il compilatore ignorerà tali nomi. Per esempio:

```
double sqrt(double d);   // restituisce la radice quadrata di d
double square(double);   // restituisce il quadrato dell'argomento
```

Il tipo di una funzione è costituito dal suo tipo restituito seguito dalla sequenza dei tipi degli argomenti tra parentesi. Per esempio:

```
double get(const vector<double>& vec, int index); // tipo: double(const vector<double>&,int)
```

Una funzione può essere un membro di una classe [§2.3, §5.2.1]. Per tale *funzione membro*, anche il nome della sua classe fa parte del tipo di funzione. Per esempio:

```
char& String::operator[](int index);   // tipo: char& String::(int)
```

Il codice deve essere comprensibile poiché questo è il primo passo verso la possibilità di manutenzione. Il primo passo verso la comprensibilità è suddividere le attività computazionali in blocchi significativi (rappresentati come funzioni e classi) e assegnare loro un nome. Tali funzioni forniscono il vocabolario di base della computazione, così come i tipi (incorporati e definiti dall'utente) forniscono il vocabolario di base dei dati.
Gli algoritmi standard di C++ (per esempio, `find`, `sort` e `iota`) costituiscono un buon punto di partenza (Capitolo 13). Successivamente possiamo comporre funzioni che rappresentino attività comuni o speciali in computazioni più grandi.
Il numero di errori nel codice è fortemente legato alla quantità e alla complessità del codice stesso. Entrambi i problemi sono affrontabili usando un maggior numero di funzioni più brevi. L'uso di una funzione per un'attività specifica spesso può evitarci la necessità di scrivere un brano di codice specifico nel bel mezzo di altro codice; rendere l'attività una funzione ci obbliga a darle un nome e a documentare le sue dipendenze.

Se non riusciamo a trovare un nome adatto, è molto probabile che sorgano problemi di progettazione.
Se due funzioni sono definite con lo stesso nome, ma hanno tipi di argomenti diversi, il compilatore sceglierà la funzione più appropriata da invocare per ogni chiamata. Per esempio:

```
void print(int);      // accetta un argomento intero
void print(double);   // accetta un argomento in virgola mobile
void print(string);   // accetta un argomento stringa

void user()
{
     print(42);            // chiama print(int)
     print(9.65);          // chiama print(double)
     print("Barcelona");   // chiama print(string)
}
```

Se possono essere chiamate due funzioni alternative, ma nessuna delle due è migliore dell'altra, la chiamata è ritenuta ambigua e il compilatore segnala un errore. Per esempio:

```
void print(int,double);
void print(double,int);

void user2()
{
     print(0,0);   // errore: ambiguo
}
```

Definire più funzioni con lo stesso nome è noto come *sovraccaricamento della funzione* ed è una delle parti essenziali della programmazione generica [§8.2]. Quando una funzione è sovraccaricata, tutte le funzioni con lo stesso nome devono implementare la medesima semantica. Le funzioni `print()` sono un esempio in questo senso: ciascuna `print()` stampa il suo argomento.

1.4 Tipi, variabili e aritmetica

Tutti i nomi e tutte le espressioni hanno un tipo che determina le operazioni eseguibili. Per esempio, la dichiarazione

```
int inch;
```

specifica che `inch` è di tipo `int`, ovvero che `inch` è una variabile intera.
Una *dichiarazione* è un'istruzione che introduce un'entità nel programma e ne specifica il tipo:

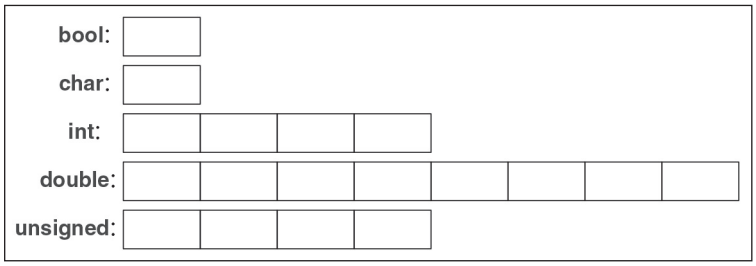
- un *tipo* definisce un set di possibili valori e uno di possibili operazioni (per un oggetto);
- un *oggetto* è una porzione di memoria che contiene un valore di qualche tipo;
- un *valore* è un set di bit interpretato secondo un tipo;
- una *variabile* è un oggetto con nome.

C++ offre svariati tipi fondamentali. Potete trovarli tutti nelle fonti di riferimento, come [Cppreference] sul Web. Alcuni esempi sono:

```
bool      // Booleano, i valori possibili sono true e false
```

```
char       // carattere, per esempio 'a', 'z' e '9'
int        // intero, per esempio -273, 42 e 1066
double     // numero in virgola mobile a doppia precisione, per esempio -273.15, 3.14
           // e 6.626e-34
unsigned   // intero non negativo, per esempio 0, 1 e 999 (utilizzare per operazioni
           // logiche bit a bit)
```

Ciascun tipo fondamentale corrisponde direttamente a un elemento hardware e ha una dimensione fissa che determina l'intervallo di valori che è possibile memorizzarvi:



Una variabile `char` è della dimensione naturale adatta a contenere un carattere in una certa macchina (di solito un byte a 8 bit), e le dimensioni degli altri tipi sono multipli della dimensione di un `char`. La dimensione di un tipo è definita dall'implementazione (cioè può variare a seconda della macchina) ed è ottenibile dall'operatore `sizeof`; per esempio, `sizeof(char)` equivale a `1` e `sizeof(int)` è spesso `4`. Quando vogliamo un tipo di una dimensione specifica, usiamo un alias di tipo della libreria standard, come `int32_t` [§17.8]. I numeri possono essere in virgola mobile o interi.

- I letterali in virgola mobile sono riconosciuti da un punto decimale (per esempio, `3.14`) o da un esponente (per esempio, `314e-2`).
- I letterali interi sono di default decimali (per esempio, `42` significa quarantadue). Un prefisso `0b` indica un letterale intero binario (base 2) (per esempio, `0b10101010`). Un prefisso `0x` indica un letterale intero esadecimale (base 16) (per esempio, `0xBAD12CE3`). Un prefisso `0` indica un letterale intero ottale (base 8) (per esempio, `0334`).

Per rendere i letterali lunghi più leggibili per gli umani, possiamo usare un singolo apice (') come separatore di cifre. Per esempio, π è circa `3.14159'26535'89793'23846'26433'83279'50288` o se preferite una notazione esadecimale `0x3.243F'6A88'85A3'08D3`.

1.4.1 Aritmetica

Per combinare in modo appropriato i tipi fondamentali è possibile usare gli operatori aritmetici:

```
x+y   // più
+x    // più unario
x-y   // meno
-x    // meno unario
x*y   // per
x/y   // diviso
x%y   // resto (modulo) per interi
```

Così come gli operatori di confronto:

```
x==y   // uguale
x!=y   // non uguale
x<y    // minore di
x>y    // maggiore di
x<=y   // minore di o uguale a
x>=y   // maggiore di o uguale a
```

Inoltre, sono disponibili anche operatori logici:

```
x&y    // and bit a bit
x|y    // or bit a bit
x^y    // or esclusivo bit a bit
~x     // complemento bit a bit
x&&y   // and logico
x||y   // or logico
!x     // not logico (negazione)
```

Un operatore logico bit a bit restituisce un risultato del tipo di operando per cui l'operazione è stata eseguita su ogni bit. Gli operatori logici `&&` e `||` restituiscono semplicemente `true` o `false` a seconda dei valori dei loro operandi.
Negli assegnamenti e nelle operazioni aritmetiche, C++ esegue tutte le conversioni sensate tra i tipi elementari per consentire di mischiarli liberamente:

```
void some_function()   // funzione che non restituisce alcun valore
{
     double d = 2.2;   // inizializza un numero in virgola mobile
     int i = 7;        // inizializza un intero
     d = d+i;          // assegna la somma a d
     i = d*i;          // assegna il prodotto; (troncando il double d*i a un int)
}
```

Le conversioni usate nelle espressioni sono dette *conversioni aritmetiche consuete* e il loro scopo è garantire che le espressioni siano valutate alla massima precisione degli operandi. Per esempio, la somma di un `double` e di un `int` viene calcolata in virgola mobile a doppia precisione.
Osservate che `=` è l'operatore di assegnazione e che `==` è il test di uguaglianza.
Oltre agli operatori aritmetici e logici convenzionali, C++ offre operazioni più specifiche per la modifica di una variabile:

```
x+=y   // x = x+y
++x    // incremento: x = x+1
x-=y   // x = x-y
--x    // decremento: x = x-1
x*=y   // scala: x = x*y
x/=y   // scala: x = x/y
x%=y   // x = x%y
```

Questi operatori sono concisi, convenienti e usati molto di frequente.
L'ordine di valutazione è da sinistra a destra per `x.y`, `x->y`, `x(y)`, `x[y]`, `x<<y`, `x>>y`, `x&&y` e `x||y`. Per gli assegnamenti (per esempio, `x+=y`), l'ordine è da destra a sinistra. Per ragioni storiche legate all'ottimizzazione, l'ordine di valutazione di altre espressioni (per esempio, `f(x)+g(y)`) e degli argomenti di funzioni (per esempio, `h(f(x),g(y))`) non è purtroppo specificato.

1.4.2 Inizializzazione

Prima che un oggetto possa essere utilizzato, gli deve essere assegnato un valore. C++ offre una varietà di notazioni per esprimere l'inizializzazione, come il simbolo = usato sopra, e una forma universale basata su elenchi di inizializzatori delimitati da parentesi graffe:

```
double d1 = 2.3;          // inizializza d1 a 2.3
double d2 {2.3};          // inizializza d2 a 2.3
double d3 = {2.3};        // inizializza d3 a 2.3 (il simbolo = è facoltativo con {...})

complex<double> z = 1;          // un numero complesso con scalari in virgola mobile
                                // a doppia precisione
complex<double> z2 {d1,d2};
complex<double> z3 = {d1,d2};   // il simbolo = è facoltativo con {...}

vector<int> v {1, 2, 3, 4, 5, 6};  // un vettore di interi
```

La forma = è tradizionale e risale al C, ma nel dubbio usate la forma generale di lista tra {}. Se non altro vi evita conversioni con perdita di informazioni:

```
int i1 = 7.8;   // i1 diventa 7 (sorpresa?)
int i2 {7.8};   // errore: conversione da virgola mobile a intero
```

Sfortunatamente, le conversioni con perdita di informazioni (*narrowing conversions*), come quella da `double` a `int` e da `int` a `char`, sono consentite e applicate in modo implicito quando si usa il simbolo = (ma non quando si usano le {}). I problemi causati dalle conversioni narrowing sono il prezzo della compatibilità con il C [§19.3].

Non è consentito lasciare una costante non inizializzata [§1.6], e le variabili non inizializzate dovrebbero essere presenti solo in circostanze estremamente rare. Non introducete un nome se non avete un valore adeguato corrispondente. I tipi definiti dall'utente (come `string`, `vector`, `Matrix`, `Motor_controller` e `Orc_warrior`) possono essere definiti come inizializzati implicitamente [§5.2.1].

Nella definizione di una variabile, non è in realtà necessario dichiararne il tipo esplicitamente quando è possibile dedurlo dall'inizializzatore:

```
auto b = true;      // un bool
auto ch = 'x';      // un char
auto i = 123;       // un int
auto d = 1.2;       // un double
auto z = sqrt(y);   // z è di qualsiasi tipo sia restituito da sqrt(y)
auto bb {true};     // bb è un bool
```

Con `auto` si tende a usare = poiché non è coinvolta alcuna conversione di tipo che potrebbe causare problemi, ma se invece preferite usare l'inizializzazione {} in modo coerente, potete farlo.

Usiamo `auto` nei casi in cui non vi sia una ragione specifica per menzionare il tipo esplicitamente. Le "ragioni specifiche' includono:

- la definizione è in un grande ambito nel quale vogliamo rendere il tipo chiaramente visibile ai lettori del codice;
- il tipo dell'inizializzatore non è ovvio;

- desideriamo essere espliciti sull'intervallo o sulla precisione di una variabile (per esempio, `double` anziché `float`).

Usando `auto`, evitiamo la ridondanza e la necessità di scrivere nomi lunghi per i tipi. Questo è particolarmente importante nella programmazione generica, dove un programmatore può avere difficoltà a conoscere il tipo esatto di un oggetto e i nomi dei tipi possono essere molto lunghi [§13.2].

1.5 Ambito e durata

Una dichiarazione introduce un nome in un ambito (o campo di visibilità).

- *Ambito locale.* Un nome dichiarato in una funzione [§1.3] o in una lambda [§7.3.2] è chiamato *nome locale*. Il suo ambito si estende dal punto della sua dichiarazione alla fine del blocco in cui essa avviene. Un *blocco* è delimitato da una coppia `{ }`. I nomi degli argomenti delle funzioni sono considerati nomi locali.
- *Ambito della classe.* Un nome è detto *nome del membro* (o *nome del membro della classe*) se è definito in una classe [§2.2, §2.3, Capitolo 5], all'esterno di qualsiasi funzione [§1.3], lambda [§7.3.2] o `enum class` [§2.4]. Il suo ambito si estende dalla `{` di apertura della dichiarazione che la racchiude fino alla `}` corrispondente.
- *Ambito del namespace.* Un nome è detto *nome del membro del namespace* se è definito in un namespace [§3.3] all'esterno di qualsiasi funzione, lambda [§7.3.2], classe [§2.2, §2.3, Capitolo 5] o `enum class` [§2.4]. Il suo ambito si estende dal punto di dichiarazione alla fine del suo namespace.

Un nome non dichiarato all'interno di qualsiasi altro costrutto è detto *nome globale* e lo si definisce come interno al *namespace globale*.
Inoltre sono possibili oggetti senza nome, come gli oggetti temporanei e quelli creati usando `new` [§5.2.2]. Per esempio:

```
vector<int> vec;    // vec è globale (un vettore globale di interi)

void fct(int arg)   // fct è globale (una funzione globale)
                    // arg è locale (un argomento intero)
{
     string motto {"Who dares wins"};   // motto è locale
     auto p = new Record{"Hume"};       // p punta a un Record senza nome (creato da new)
     // ...
}
struct Record {
     string name;    // name è un membro di Record (un membro string)
     // ...
};
```

Un oggetto dev'essere costruito (inizializzato) prima di essere usato e sarà distrutto alla fine del suo ambito. Per un oggetto namespace il punto di distruzione è la fine del programma. Per un membro, il punto di distruzione è determinato dal punto di distruzione dell'oggetto di cui è membro. Un oggetto creato da `new` "vive" fino a quando non è distrutto da `delete` [§5.2.2].

1.6 Costanti

C++ supporta due concetti di *immutabilità* (un oggetto con uno stato non modificabile).

- `const`: in linea di massima significa "prometto di non cambiare questo valore". È usato principalmente per specificare interfacce, in modo che i dati possano essere passati a funzioni usando puntatori e riferimenti senza timore che siano modificati. Il compilatore applica la promessa fatta da `const`. Il valore di una costante può essere calcolato in fase di esecuzione.
- `constexpr`: in linea di massima significa "da valutare all'atto della compilazione". È usato principalmente per specificare costanti, per consentire il posizionamento dei dati nella memoria di sola lettura (dove esistono meno possibilità che vengano corrotti) e per migliorare le prestazioni. Il valore di una `constexpr` può essere calcolato dal compilatore.

Per esempio:

```
constexpr int dmv = 17;        // dmv è una costante con nome
int var = 17;                  // var non è una costante
const double sqv = sqrt(var);  // sqv è una costante con nome, possibilmente calcolata
                               // in fase di esecuzione

double sum(const vector<double>&);   // sum non modificherà il suo argomento [§1.7]

vector<double> v {1.2, 3.4, 4.5};    // v non è una costante
const double s1 = sum(v);            // OK: sum(v) è valutato in fase di esecuzione
constexpr double s2 = sum(v);        // error: sum(v) non è un'espressione costante
```

Per poter essere usata in un'*espressione costante*, ovvero un'espressione che sarà valutata dal compilatore, una funzione dev'essere definita `constexpr` o `consteval`. Per esempio:

```
constexpr double square(double x) { return x*x; }

constexpr double max1 = 1.4*square(17);   // OK: 1.4*square(17) è un'espressione costante
constexpr double max2 = 1.4*square(var);  // errore: var non è una costante, quindi
                                          // square(var) non è una costante
const double max3 = 1.4*square(var);      // OK: può essere valutato in fase di esecuzione
```

Una funzione `constexpr` può essere usata per argomenti non costanti, ma in questi casi il risultato non è un'espressione costante. Consentiamo la chiamata a una funzione `constexpr` con argomenti non da espressione costante nei contesti che non richiedono espressioni costanti, così da non dover definire due volte essenzialmente la stessa funzione: una volta per le espressioni costanti e una volta per le variabili. Quando vogliamo che una funzione sia usata solo per la valutazione in fase di compilazione, la dichiariamo `consteval` anziché `constexpr`. Per esempio:

```
consteval double square2(double x) { return x*x; }

constexpr double max1 = 1.4*square2(17);  // OK: 1.4*square(17) è un'espressione costante
const double max3 = 1.4*square2(var);     // errore: var non è una costante
```

Le funzioni dichiarate `constexpr` o `consteval` sono la versione C++ della nozione di *funzioni pure*. Non hanno effetti collaterali e possono utilizzare solo le informazioni trasmesse loro come argomenti. In particolare, non possono modificare variabili non locali, ma possono avere cicli e utilizzare le proprie variabili locali. Per esempio:

```
constexpr double nth(double x, int n)   // assume 0<=n
{
     double res = 1;
     int i = 0;
     while (i<n) {   // while-loop: do while the condition is true [§1.7.1]
          res *= x;
          ++i;
     }
     return res;
}
```

In alcuni punti, le espressioni costanti sono richieste dalle regole del linguaggio (per esempio, nei limiti degli array [§1.7], nelle etichette `case` [§1.8], negli argomenti valore dei template [§7.2] e nelle costanti dichiarate con `constexpr`). In altri casi, la valutazione all'atto della compilazione è importante per le prestazioni. A prescindere dai problemi di prestazioni, il concetto di immutabilità (un oggetto con stato non modificabile) è un'importante questione progettuale.

1.7 Puntatori, array e riferimenti

La raccolta di dati più elementare è una sequenza di elementi dello stesso tipo allocata in modo contiguo, chiamata *array*. Questo è fondamentalmente ciò che l'hardware offre. Un array di elementi di tipo `char` può essere dichiarato come segue:

```
char v[6];   // array di 6 caratteri
```

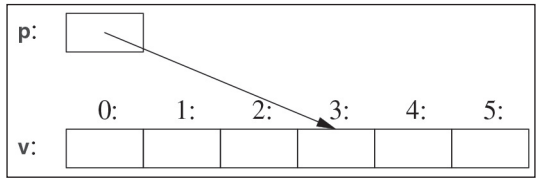
In modo simile, un puntatore è dichiarabile come segue:

```
char* p;     // puntatore a carattere
```

Nelle dichiarazioni, `[ ]` significa "array di" e `*` significa "puntatore a". Tutti gli array hanno `0` come limite minimo, quindi `v` ha sei elementi, da `v[0]` a `v[5]`. La dimensione di un array deve essere un'espressione costante [§1.6]. Una variabile puntatore può contenere l'indirizzo di un oggetto del tipo appropriato:

```
char* p = &v[3];   // p punta al quarto elemento di v
char x = *p;       // *p è l'oggetto a cui punta p
```

In un'espressione, il prefisso unario `*` significa "contenuto di" e il prefisso unario `&` significa "indirizzo di". Eccone una rappresentazione grafica:



Considerate la stampa degli elementi di un array:

```
void print()
{
     int v1[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```
    for (auto i=0; i!=10; ++i)   // stampa elementi
        cout << v[i] << '\n';
    // ...
}
```

Questa istruzione `for` può essere letta come "imposta `i` a zero; finché `i` non è `10`, stampa l'`i`esimo elemento e incrementa `i`". C++ offre anche un'istruzione `for` più semplice, detta istruzione range-`for`, per i cicli che attraversano una sequenza nel modo più semplice:

```
void print2()
{
    int v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    for (auto x : v)                        // per ogni x in v
        cout << x << '\n';

    for (auto x : {10, 21, 32, 43, 54, 65})   // per ogni intero nella lista
        cout << x << '\n';
    // ...
}
```

La prima istruzione range-`for` può essere letta come "per ciascun elemento di `v`, dal primo all'ultimo, poni una copia in `x` e stampala". Osservate che non dobbiamo specificare i limiti dell'array quando lo inizializziamo con una lista. L'istruzione range-`for` può essere usata per qualsiasi sequenza di elementi [§13.1].
Se non volessimo copiare i valori da `v` nella variabile `x`, ma semplicemente ottenere che `x` faccia riferimento a un elemento, potremmo scrivere:

```
void increment()
{
    int v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    for (auto& x : v)   // aggiunta di 1 a ogni x in v
        ++x;
    // ...
}
```

In una dichiarazione, il suffisso unario `&` significa "riferimento a". Un riferimento è simile a un puntatore, con la differenza che non è necessario usare un prefisso `*` per accedere al valore a cui si fa riferimento. Inoltre, non è possibile far sì che un riferimento si riferisca a un altro oggetto dopo la sua inizializzazione.
I riferimenti sono particolarmente utili per specificare gli argomenti delle funzioni. Per esempio:

```
void sort(vector<double>& v);   // ordina v (v è un vettore di double)
```

Usando un riferimento ci accertiamo che una chiamata `sort(my_vec)` non crei una copia di `my_vec`, e che sia proprio `my_vec` a essere ordinato e non una sua copia.
Quando non vogliamo modificare un argomento ma non vogliamo comunque il costo della copia, usiamo un riferimento `const` [§1.6], ovvero un riferimento a una costante. Per esempio:

```
double sum(const vector<double>&)
```

Le funzioni che accettano riferimenti `const` sono molto comuni.

Quando sono utilizzati nelle dichiarazioni, gli operatori (quali `&`, `*` e `[ ]`) sono chiamati *operatori dichiaratori*:

```
T a[n]   // T[n]: a è un array di n T
T* p     // T*: p è un puntatore a T
T& r     // T&: r è un riferimento a T
T f(A)   // T(A): f è una funzione che accetta un argomento di tipo A restituendo
         // un risultato di tipo T
```

1.7.1 Il puntatore null

Cerchiamo di garantire che un puntatore punti sempre a un oggetto, in modo che la sua dereferenziazione sia valida. Quando non disponiamo di un oggetto a cui puntare, o se dobbiamo rappresentare il concetto di "nessun oggetto disponibile" (per esempio per la fine di una lista), passiamo al puntatore il valore `nullptr` ("il puntatore null'). Esiste un solo `nullptr` condiviso da tutti i tipi di puntatori:

```
double* pd = nullptr;
Link<Record>* lst = nullptr;   // puntatore a un Link a un Record
int x = nullptr;               // errore: nullptr è un puntatore, non un intero
```

Spesso è opportuno verificare che l'argomento di un puntatore punti realmente a qualcosa:

```
int count_x(const char* p, char x)
     // conta il numero di occorrenze di x in p[]
     // si presume che p punti a un array di char con termine zero (o a nulla)
{
     if (p==nullptr)
          return 0;
     int count = 0;
     for (; *p!=0; ++p)
          if (*p==x)
               ++count;
     return count;
}
```

È possibile spostare il puntatore in modo che punti al successivo elemento di un array usando `++` e anche omettere l'inizializzatore in un'istruzione `for` se non è necessario.
La definizione di `count_x()` presuppone che il `char*` sia una *stringa in stile C*, ovvero che il puntatore punti a un array di `char` con termine zero. I caratteri in un letterale stringa sono immutabili, quindi per gestire `count_x("Hello!")` ho dichiarato `count_x()` un argomento `const char*`.
Nel codice più datato, di solito si usano `0` o `NULL` anziché `nullptr`. Tuttavia, l'impiego di `nullptr` elimina la possibile confusione tra interi (quali `0` o `NULL`) e puntatori (quali `nullptr`). Nell'esempio di `count_x()`, non usiamo la parte dell'inizializzatore dell'istruzione `for`, quindi possiamo usare la più semplice istruzione `while`:

```
int count_x(const char* p, char x)
     // conta il numero di occorrenze di x in p[]
     // si presuppone che p punti a un array di char con termine zero (o a nulla)
{
     if (p==nullptr)
          return 0;
     int count = 0;
```

```
    while (*p) {
        if (*p==x)
            ++count;
        ++p;
    }
    return count;
}
```

L'istruzione `while` è eseguita fino a quando la sua condizione non diviene `false`.
Un test di un valore numerico (per esempio, `while (*p)` in `count_x()`) equivale al confronto del valore a `0` (per esempio, `while (*p!=0)`). Un test di un valore puntatore (per esempio, `if (p)`) equivale al confronto del valore con `nullptr` (per esempio, `if (p!=nullptr)`).
Non esiste il "riferimento null". Un riferimento deve essere sempre diretto a un oggetto valido (e le implementazioni partono da questo presupposto). Vi sono modi oscuri e ingegnosi per violare questa regola, ma non fatelo.

1.8 Test

C++ offre un set convenzionale di istruzioni per esprimere selezione e cicli, come le istruzioni `if`, le istruzioni `switch`, i cicli `while` e i cicli `for`. Per esempio, ecco una semplice funzione che richiede l'input dell'utente e restituisce un booleano che indica la risposta:

```
bool accept()
{
    cout << "Do you want to proceed (y or n)?\n";   // scrive domanda
    char answer = 0;   // inizializza a un valore che non apparirà sull'input
    cin >> answer;     // legge risposta

    if (answer == 'y')
        return true;
    return false;
}
```

All'operatore di output `<<` è abbinato l'operatore di input `>>`; `cin` è il flusso dello standard input (Capitolo 11). Il tipo dell'operando a destra di `>>` determina l'input accettato, e l'operando di destra è l'obiettivo dell'operazione di input. Il carattere `\n` alla fine della stringa di output rappresenta un newline [§1.2.1].
Notate che la definizione di `answer` appare quando è richiesta (e non prima). Una dichiarazione può apparire ovunque possa apparire un'istruzione.
L'esempio può essere migliorato prendendo in considerazione la risposta `n` (per "no"):

```
bool accept2()
{
    cout << "Do you want to proceed (y or n)?\n";    // scrive domanda
    char answer = 0;              // inizializza a un valore che non apparirà sull'input
    cin >> answer;                // legge risposta

    switch (answer) {
    case 'y':
        return true;
    case 'n':
        return false;
```

```
    default:
        cout << "I'll take that for a no.\n";
        return false;
    }
}
```

Un'istruzione `switch` testa un valore rispetto a un set di costanti. Quelle costanti, chiamate etichette `case`, devono essere distinte, e se il valore testato non corrisponde ad alcuna di esse viene scelto il `default`. Se il valore non corrisponde ad alcuna etichetta `case` e non viene fornito alcun `default`, non viene intrapresa alcuna azione.
Non è indispensabile uscire da una `case` tornando dalla funzione che contiene la sua istruzione `switch`. Spesso si desidera soltanto proseguire l'esecuzione con l'istruzione che segue la `switch`, e possiamo farlo usando un'istruzione `break`. Come esempio, considerate questo parser, estremamente intelligente seppur primitivo, per un banale video game militare:

```
void action()
{
    while (true) {
        cout << "enter action:\n";   // richiede azione
        string act;
        cin >> act;                  // legge caratteri in una stringa
        Point delta {0,0};           // Point contiene una coppia {x,y}

        for (char ch : act) {
            switch (ch) {
            case 'u':   // up (su)
            case 'n':   // north (nord)
                ++delta.y;
                break;
            case 'r':   // right (destra)
            case 'e':   // east (est)
                ++delta.x;
                break;
            // ... altre azioni ...
            default:
                cout << "I freeze!\n";
            }
            move(current+delta*scale);
            update_display();
        }
    }
}
```

Come un'istruzione `for` [§1.7], un'istruzione `if` può introdurre una variabile e testarla. Per esempio:

```
void do_something(vector<int>& v)
{
    if (auto n = v.size(); n!=0) {
        // ... arriviamo qui se n!=0 ...
    }
    // ...
}
```

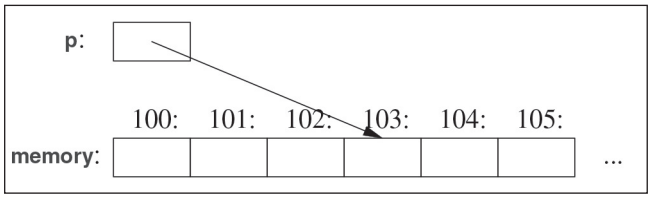
Qui, l'intero `n` è definito per l'uso all'interno dell'istruzione `if`, inizializzato con `v.size()` e subito testato dalla condizione `n!=0` dopo il punto e virgola. Un nome dichiarato in una condizione è all'interno dell'ambito delimitato dalle parentesi graffe dell'istruzione `if`. Come con l'istruzione `for`, lo scopo della dichiarazione di un nome nella condizione di un'istruzione `if` è quello di mantenere limitato l'ambito della variabile per migliorare la leggibilità e ridurre al minimo gli errori.
Il caso più comune è testare una variabile rispetto a `0` (o `nullptr`). Per farlo, limitatevi a tralasciare la menzione esplicita della condizione. Per esempio:

```
void do_something(vector<int>& v)
{
     if (auto n = v.size()) {
          // ... arriviamo qui se n!=0 ...
     }
     // ...
}
```

Cercate di usare questa forma più concisa e più semplice quando potete farlo.

1.9 Mappatura sull'hardware

C++ fornisce una mappatura diretta sull'hardware. Quando utilizzate una delle operazioni elementari, l'implementazione è ciò che l'hardware offre, in genere una singola operazione macchina. Per esempio, sommando due `int`, `x+y` esegue un'istruzione macchina per la somma di interi.
Un'implementazione C++ vede la memoria di una macchina come una sequenza di locazioni di memoria in cui può posizionare oggetti (tipizzati) e indirizzarli utilizzando puntatori:



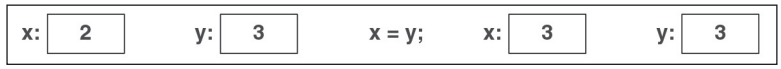
Un puntatore è rappresentato in memoria come un indirizzo macchina, quindi il valore numerico di `p` in questa figura sarebbe `103`. Se questo assomiglia molto a un array [§1.7] è perché un array è l'astrazione di base di C++ di "una sequenza contigua di oggetti in memoria".
La semplice mappatura dei costrutti fondamentali del linguaggio sull'hardware è cruciale per le prestazioni grezze di basso livello per le quali C e C++ sono famosi da decenni. Il modello macchina di base di C e C++ fa riferimento all'hardware del computer anziché a qualche forma di matematica.

1.9.1 Assegnazione

Un'assegnazione di un tipo incorporato è una semplice operazione macchina di copia. Considerate:

```
int x = 2;
int y = 3;
x = y;       // x diventa 3; quindi otteniamo x==y
```

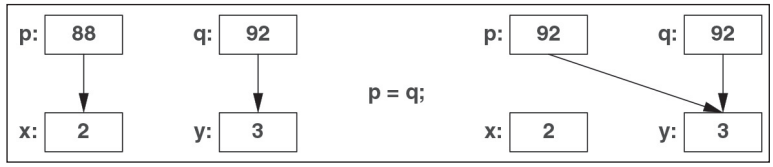
Questo è ovvio, e possiamo rappresentarlo graficamente così:



I due oggetti sono indipendenti. Possiamo cambiare il valore di `y` senza influenzare il valore di `x`. Per esempio, `x=99` non cambierà il valore di `y`. A differenza di Java, C# e altri linguaggi, ma come C, questo vale per tutti i tipi, non solo per gli `int`.
Se vogliamo che oggetti diversi facciano riferimento allo stesso valore (condiviso), dobbiamo dirlo. Per esempio:

```
int x = 2;
int y = 3;
int* p = &x;
int* q = &y;   // p!=q e *p!=*q
p = q;         // p diventa &y; ora p==q, quindi (ovviamente) *p==*q
```

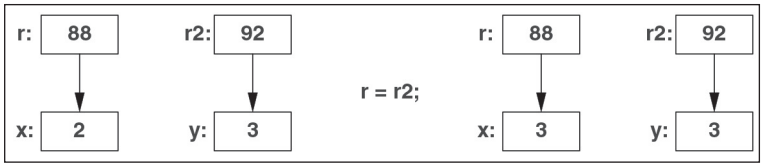
Possiamo rappresentarlo graficamente così:



Ho scelto arbitrariamente `88` e `92` come indirizzi degli `int`. Di nuovo, possiamo vedere che l'oggetto su cui avviene l'assegnazione ottiene il valore dall'oggetto assegnato, producendo due oggetti indipendenti (qui, puntatori), con lo stesso valore. Cioè, `p=q` dà `p==q`. Dopo `p=q`, entrambi i puntatori puntano a `y`.
Un riferimento e un puntatore fanno riferimento/puntano a un oggetto ed entrambi sono rappresentati in memoria come indirizzo macchina. Tuttavia, le regole linguistiche per il loro utilizzo differiscono. L'assegnazione a un riferimento non modifica ciò a cui è diretto il riferimento ma assegna all'oggetto a cui si fa riferimento:

```
int x = 2;
int y = 3;
int& r = x;    // r fa riferimento a x
int& r2 = y;   // r2 fa riferimento a y
r = r2;        // legge attraverso r2, scrive attraverso r: x diventa 3
```

Possiamo rappresentarlo graficamente così:



Per accedere al valore puntato da un puntatore si usa `*`, e ciò è implicitamente fatto per un riferimento.
Dopo `x=y`, abbiamo `x==y` per ogni tipo incorporato e tipo ben progettato definito dall'utente (Capitolo 2) che offre `=` (assegnazione) e `==` (confronto di uguaglianza).

1.9.2 Inizializzazione

L'inizializzazione differisce dall'assegnazione. In generale, affinché un'assegnazione funzioni correttamente, l'oggetto su cui avviene l'assegnazione deve avere un valore. D'altra parte, il compito dell'inizializzazione è trasformare un pezzo di memoria non inizializzato in un oggetto valido. Per quasi tutti i tipi, l'effetto della lettura da una variabile non inizializzata o della scrittura su di essa non è definito. Considerate i riferimenti:

```
int x = 7;
int& r {x};     // lega r a x (r fa riferimento a x)
r = 7;          // assegna a qualunque cosa r si riferisca

int& r2;        // errore: riferimento non inizializzato
r2 = 99;        // assegna a qualunque cosa r2 si riferisca
```

Fortunatamente, non possiamo avere un riferimento non inizializzato; se potessimo, allora `r2=99` assegnerebbe `99` a una locazione di memoria non specificata; il risultato alla fine porterebbe a risultati negativi o a un arresto anomalo.
Si può usare `=` per inizializzare un riferimento ma cercate di non farvi confondere. Per esempio:

```
int& r = x;     // lega r a x (r fa riferimento a x)
```

Questa è ancora inizializzazione e lega `r` a `x`, anziché una qualche forma di copia del valore. La distinzione tra inizializzazione e assegnazione è cruciale anche per molti tipi definiti dall'utente come `string` e `vector`, dove un oggetto su cui avviene l'assegnazione possiede una risorsa che alla fine deve essere rilasciata [§6.3].
La semantica di base del passaggio degli argomenti e della restituzione del valore della funzione è quella dell'inizializzazione [§3.4]. Per esempio, è così che otteniamo il passaggio per riferimento [§3.4.1].

1.10 Consigli

Questi consigli sono un sottoinsieme delle C++ Core Guidelines [Stroustrup, 2015]. I riferimenti alle linee guida hanno questo aspetto [CG: ES.23], a indicare la 23esima

regola nella sezione Expressions and Statement. In generale, una "linea guida di base" offre ulteriori motivazioni ed esempi.

1. Niente panico! Col tempo tutto si chiarirà; [§1.1]; [CG: In.0].
2. Non utilizzate esclusivamente le funzionalità incorporate. Molte funzionalità di base (incorporate) di solito sono utilizzate al meglio indirettamente tramite librerie, come la libreria standard ISO C++ [Capitoli 9-18]; [CG: P.13].
3. Usate `#include` o (preferibilmente) `import` per includere o importare le librerie necessarie per semplificare la programmazione; [§1.2.1].
4. Non è necessario conoscere tutti i dettagli di C++ per scrivere buoni programmi.
5. Concentratevi sulle tecniche di programmazione, non sulle caratteristiche del linguaggio.
6. Per la parola definitiva sulle questioni di definizione del linguaggio fate riferimento allo standard ISO C++; [§19.1.3]; [CG: P.2].
7. "Impacchettate" le operazioni significative in funzioni accuratamente denominate; [§1.3]; [CG: F.1].
8. Una funzione dovrebbe eseguire un'unica operazione logica; [§1.3]; [CG: F.2].
9. Scrivete funzioni brevi; [§1.3]; [CG: F.3].
10. Usate il sovraccaricamento quando le funzioni eseguono concettualmente la stessa attività su tipi diversi; [§1.3].
11. Se una funzione deve essere valutata in fase di compilazione, dichiaratela `constexpr`; [§1.6]; [CG: F.4].
12. Se è possibile che una funzione debba essere valutata in fase di compilazione, dichiaratela `consteval`; [§1.6].
13. Se una funzione potrebbe non avere effetti collaterali, dichiaratela `constexpr` o `consteval`; [§1.6]; [CG: F.4].
14. Comprendete come mappare le primitive del linguaggio sull'hardware; [§1.4, §1.7, §1.9, §2.3, §5.2.2, §5.4].
15. Utilizzate i separatori di cifre per rendere leggibili i valori letterali grandi; [§1.4]; [CG: NL.11].
16. Evitate espressioni complicate; [CG: ES.40].
17. Evitate conversioni narrowing; [§1.4.2]; [CG: ES.46].
18. Riducete al minimo l'ambito di una variabile; [§1.5, §1.8].
19. Mantenete gli ambiti piccoli; [§1.5]; [CG: ES.5].
20. Evitate le "costanti magiche" e usate quelle simboliche; [§1.6]; [CG: ES.45].
21. Preferite i dati immutabili; [§1.6]; [CG: P.10].
22. Dichiarate (solo) un nome per dichiarazione; [CG: ES.10].
23. Usate nomi comuni e locali brevi, e nomi non comuni e non locali lunghi; [CG: ES.7].
24. Evitate nomi simili; [CG: ES.8].
25. Evitate nomi `TUTTI_MAIUSCOLI`; [CG: ES.9].
26. Preferite la sintassi `{}` per l'inizializzazione in dichiarazioni con un tipo con nome; [§1.4]; [CG: ES.23].
27. Usate `auto` per evitare di ripetere i nomi dei tipi; [§1.4.2]; [CG: ES.11].
28. Evitate variabili non inizializzate; [§1.4]; [CG: ES.20].
29. Non dichiarate una variabile prima di disporre di un valore con cui inizializzarla; [§1.7, §1.8]; [CG: ES.21].

30. Quando dichiarate una variabile nella condizione di un'istruzione `if`, preferite la versione con il test implicito rispetto a `0` o `nullptr`; [§1.8].
31. Preferite cicli range-`for` a cicli `for` con una variabile di ciclo esplicita; [§1.7].
32. Usate `unsigned` solo per la manipolazione dei bit; [§1.4]; [CG: ES.101; CG: ES.106].
33. Mantenete l'uso dei puntatori semplice e diretto; [§1.7]; [CG: ES.42].
34. Usate `nullptr` anziché `0` o `NULL`; [§1.7]; [CG: ES.47].
35. Non dite nei commenti ciò che può essere chiaramente dichiarato nel codice; [CG: NL.1].
36. Dichiarate l'intento nei commenti; [CG: NL.2].
37. Mantenete uno stile costante per i rientri; [CG: NL.4].