C++: Sintassi di Base

Programmazione Avanzata e Parallela
2022/2023

Alberto Casagrande

Il Linguaggio C

Sviluppato alla fine degli anni '60

È un linguaggio di alto livello molto efficiente

```
unsigned int fact(const unsigned int value) {
    unsigned int res = 1;

    while (value>1) {
        res *= value--;
    }

    return res;
}
```

Nuovi Tipi di Dato in C

Usando le **strutture**

```
typedef struct {
    int numerator;
    int denominator;
} rational;
```

Tuttavia, i nuovi tipi

- non sono “cittadini di prima classe“ del linguaggio
- il modo in cui vengono "maneggiati" non dipende dal tipo di dato

Cosa Possiamo Fare in C...

```
rational sum(const rational a, const rational b);

a = sum(a, b);

a.numerator = a.numerator + b.denominator;
```

che non è molto diverso da

```
account_type *pay(account_type *a, const unsigned int money);

account = pay(account, money);

account.money = 0;
```

Cosa Vorremo Fare...

con i razionali

```
a = a + b;

a.numerator = a.numerator + b.denominator;  // ERRORE: stiamo impicciandoci
                                            // di dati "interni" di un razionale
```

con i conti correnti

```
account.pay(money);

account.money = 0;  // ERRORE: non vogliamo che questo campo
                    // sia modificabile da chiunque!
```

Tipi di Dato e Interfacce

Vogliamo definire non solo il tipo di dato, ma **soprattutto** come il codice può interagire con esso.

Vogliamo un'**interfaccia** specifica per il nuovo tipo di dato.

Es.

- i razionali si sommano
- nei conti si può versare solamente usando la "funzione" `pay`

Classi e Oggetti

Una **classe** è un tipo di dato con la relativa interfaccia

Es. `rational` con somma, prodotto, etc.

Un **oggetto** è un'istanza di una classe

Es. $\frac{3}{5}$ potrebbe essere un oggetto della classe `rational`

C++

È un estensione **orientata agli oggetti** di C

Eredita da C:

- i tipi di base (es. `int`, `float`, `double`, `void`)
- assegnamento
- espressioni Booleane e algebriche (es. `5+3*++x%2`, `!y || x`)
- costrutti condizionali e cicli (es. `while`, `if`, `do-while`)
- definizione e chiamata di funzioni

C++: Qualche Novità...

- classi e oggetti
- il tipo Booleano `bool` e il tipo stringa `std::string`
- le costanti `const`
- l'I/O è effuato tramite streams
- i commenti su una linea iniziano con `//`
- ...

Un Generico Programma C++

```
int main() {

    // Qui succede qualcosa

    return 0;
}
```

La struttura è la stessa di un programma C

Tipi di Base e Dichiarazioni

```
int a;      // char, short, long, long long e unsigned sono altri tipi interi

double b;   // long double e float sono altri tipi a virgola mobile

char c;     // char rappresenta anche i caratteri ascii come in C

bool t;     // bool è un tipo Booleano

int16_t s; // è possibile scegliere il tipo in funzione dei bit che
int32_t m; // vogliamo occupare
int64_t l;
```

`const` e `constexpr`

Una variabile `const` è immutabile

Una variabile `constexpr` è immutabile ed è valutata al tempo di compilazione

```
const int a{3+7};       // può essere valutata in fase di esecuzione

constexpr int b{3+4};   // viene valutata al tempo di compilazione

a = 5;       // errore: 'a' è immutabile
b = 5;       // errore: 'b' è immutabile
```

L'inizializzatore Universale (C++11)

```
int a = 1.1; /* questa linea converte un numero floating point (1.1) in un
                intero perdendo delle informazioni (narrowing).
                La linea è sintatticamente corretta, ma "strana". */

int b{-1};  /* l'inizializzatore universale {} inizializza le variabili.
               La semantica assomiglia all'assignamento dopo la
               dichiarazione, ma il narrowing produce un errore. */

int c{-1.1}; /* questa linea produce un errore perché  chiede il narrowing
                di 1.1 in un intero. */
```

[Vedi Cap 3 di PPP e Cap 6 di CPL]

Assegnamento ed Espressioni

```
int a{2};

a = 6;          // l'assegnamento ha la sintassi <nome var> = <espressione>

a = (a+5)/2;     // le espressioni possono essere numeriche ...

bool b = (a==a/2) && (a!=0) // ... o Booleane
```

Come in C, abbiamo due divisioni: intera e a virgola mobile

La selezione dipende dal tipo degli operandi

Pre/Post Incremento/Decremento

```
int a{1}, b;

b = 2 + a++;  // prima valuta `a`, poi la incrementa
              // qui `a` vale 2 e `b` vale 3

a = 1;

b = 2 + ++a;  // prima incrementa `a`, poi la valuta
              // qui `a` vale 2 e `b` vale 4

--a;          // `a` viene decrementato
```

Abbreviazioni di Assegnamento

```
int a{1};

a += 4;         // a = a+(4)
a -= 3*a;       // a = a-(3*a)
a *= a+0;       // a = a*(a+0)
a /= 6+a;       // a = a/(6+a)
```

Lo specificatore di tipo `auto` (C++11)

Il compilatore "deduce" il tipo delle variabili dichiarate del tipo `auto`

```
auto a = 1;          // int
auto b = true;       // Boolean
auto c = 'c';        // char
auto d = 1lu;        // unsigned long int
auto f = 1.5;        // double

auto g = f+a;        // double

auto p = &a;         // int*
```

Tipo di un'Espressione e `decltype` (C++11)

Il tipo di un'espressione può essere ottenuto usando la funzione `decltype`

```
auto a = 1;
auto b = 3.2;

decltype(a) c;      // dichiariamo 'c' con lo stesso tipo di 'a'

decltype(a+b) d;    // dichiariamo 'd' con lo stesso tipo di 'a+b'
```

I Blocchi di Istruzioni (Come in C)

Un **blocco di istruzioni** è:

- un'istruzione
- una sequenza di istruzioni tra parentesi grafe

```
int a{1};

{  // inizio un nuovo blocco
  float a{1.2};  // questa dichiarazione "nasconde" la 'a' dichiarata sopra
  a = a / 3;     // questa 'a' è quella del blocco più interno
} // qui smette di esistere la 'a' del blocco più interno

a++; // questa 'a' è quella del blocco più esterno
```

I Costrutti Condizionali (Come in C)

- `if-then`

```
if (a==1)
  a++;   // questo è il blocco del costrutto condizionale
         // viene eseguito solamente se la condizione è vera
```

- `if-then-else`

```
if (a==1)
  a++;
else {    // se la condizione è falsa, viene eseguito il seguente blocco
  a = 0;
}
```

I Costrutti per Ciclare (Come in C)

- `for-loop`

```
for (int a=0; a<100; ++a)
  b += a;  // questo è il blocco del costrutto for
           // viene iterato fintanto che la condizione è vera
```

- `while-do`

```
while (a<100) { // viene iterato fintanto che la condizione è vera
  b += a;

  a++;
}
```

I Costrutti per Ciclare 2 (Come in C)

- `do-while`

```
do {
  b += a;
} while (++a<100) // viene iterato fintanto che la condizione è vera
```

[Vedi Cap 4.4 di PPP]

Puntatori (Come in C)

I **puntatori** sono indirizzi nell'area di memoria

Usando `&` otteniamo l'area di memoria di una variabile

Dato un indirizzo possiamo accedere ai dati puntato usando `*`

```
int a{1};     // 'a' è una variabile di tipo int

int *p;       // 'p' è una variabile del tipo puntatore a interi

p = &a        // ora 'p' punta all'area di memoria di 'a'

*p += 5;      // modifico i dati puntati da 'p', i.e., 'a'
              // ora 'a' è uguale a 6, i.e., 1 + 5
```

Aritmetica di Puntatori (Come in C)

Possiamo sommare e sottrarre degli interi ai puntatori

Se il tipo è `T*`, sommando `c` incremento il puntatore del numero di byte necessari a contenere `c` valori del tipo `T`

```
int *pi{0}; double *pd{0}; char *pc{0};

pi = pi+1       // Sul mio laptop il risultato è "8 16 2" perché
pd = pd + 1     // sizeof(int)=4, sizeof(double)=8, sizeof(char)=1
pc = pc+1
```

[Vedi Cap 2.2.5 di CPL e Cap 17.4.5 in PPP]

Puntatori e `const`

Possiamo dichiarare costante:

1. l'area di memoria puntata dal puntatore, es. `const int* vc{&a}`
2. la variabile puntatore, es. `int* const pc{&a}`
3. entrambi, es. `const int* const pvc{&a}`

Nel primo caso, non possiamo usare `vc` per modificare `a`, ma `a` può essere modificato

[Vedi Cap 7.5 di CPL]

Type-casting con lo stile del C

Possiamo usare quello del C

```
int i{5}; double d{9.1};

int* pi{&i}; double* pd{&d}; const int* pic{&i};

pd = (double *)pi;    // OK: anche se i tipi puntati sono molto diversi

// *pic = 2;          // ERRORE: 'pic' è un puntatore 'const'

*((int *)pic) = 2;    // OK: anche se 'pic' "diventa" non costante
```

La semantica di questo type-cast è un po' troppo permissiva

Type-casting in C++: static_cast

Converte tipi "legati" tra loro durante la compilazione

```
int i{5}; double d{9};
int* pi{&i}; const int* pic{&i}; double* pd{&d}; void *pv{&d};

// pd = pv;                              // ERRORE: 'pv' non è un valore 'double*'
pd = static_cast<double *>(pv);          // ma 'pv' potrebbe essere un 'double*'

// pd = static_cast<double *>(pi);       // ERRORE: un valore 'int*' non è 'double*'

// pi = static_cast<int *>(pic);         // ERRORE: il valore di 'pic' è costante
```

[Vedi Cap 11.5.2 di CPL]

Type-casting in C++: reinterpret_cast

Reinterpreta i bit durante la compilazione

```
int i{5}; double d{9};
int* pi{&i}; const int* pic{&i}; double* pd{&d}; void *pv{&d};

pd = reinterpret_cast<double *>(pv);     // come per lo static_cast

pd = reinterpret_cast<double *>(pi);     // meglio dello static_cast

// pi = reinterpret_cast<int *>(pic);    // ERRORE: 'pic' è costante
```

[Vedi Cap 11.5.2 di CPL]

Type-casting in C++: const_cast

Aggiunge/rimuove il `const` durante la compilazione

```
int i{5}; double d{9};
int* pi{&i}; const int* pic{&i}; double* pd{&d};

pi = const_cast<int *>(pic);

// pd = const_cast<double *>(pi);  // ERRORE: non possiamo usare per cambiare
                                   // completamente il tipo di un'espressione
```

... ce ne sarebbe un quarto tipo. Lo vedremo più avanti.

[Vedi Cap 11.5.2 di CPL]

Puntatori e Tipi (Come in C)

L'area di memoria occupata da un dato è "neutra" rispetto al tipo del contenuto

Il compilatore interpreta il dato in funzione del tipo del puntatore

```
int a{1}; char *pc;    // 'pc' è un puntatore a un carattere

pc = reinterpret_cast<char *>(&a);  // 'pc' punta all'area di memoria di 'a'

*pc = 'c';    // modifica l'area di memoria di 'a' salvandoci un carattere

std::cout << "a=" << a << " *pc=" << *pc << std::endl;
```

Riferimenti

I **riferimenti** sono "alias" per i nomi delle variabili

La memoria associata alla variabile e al riferiemento è la stessa

```
int a{1};          // 'a' è una variabile di tipo int

int &r{a};         // 'r' è un riferimento a 'a'

r += 5;            // modifico i dati puntati da 'p', i.e., 'a'
                   // ora 'a' è uguale a 6, i.e., 1 + 5
```

[Vedi Cap 17.9 di PPP e Cap 7.2, 7.7 di CPL]

I/O in C

```c
#include <stdio.h>

int main() {
  char name[30];    // questo array può contenere al più 29 caratteri ascii
  int age;

  printf("Come ti chiami? ");
  scanf("%s", &name);
  printf("Quanti anni hai? ");
  scanf("%d", &age);

  printf("Ciao, %s (%d)", name, age);

  return 0;
}
```

I/O in C++

```cpp
#include <iostream>    // l'header per l'input output cambia
#include <string>      // serve per poter utilizzare il tipo string

int main() {
  std::string name;  // la stringa ha un tipo specifico
  unsigned int age;

  std::cout << "Come ti chiami? ";     // per stampare a video
  std::cin >> name;                    // per leggere da tastiera
  std::cout << "Quanti hanni hai? ";
  std::cin >> age;

  std::cout << "Ciao, " << name << " (" << age <<")" << std::endl;
  return 0;
}
```