

05_Types

Table of contents

- [Tipi](#)
 1. [Fondamentali](#)
 2. [Composti](#)
 3. [Qualificati](#)
 1. [`const`](#)
 2. [`volatile`](#)
 3. [Costanti letterali](#)
 1. [Letterali definiti da utente](#)

Tipi

Fondamentali

- Tipi booleani
`bool`
- Tipi carattere
`char`, `signed char`, `unsigned char` → narrow character type
`wchar_t`, `char16_t`, `char32_t` → wide character type
- Tipi interi standard con segno
`signed char`, `short`, `int`, `long`, `long long`
- Tipi interi standard senza segno
`unsigned char`, `unsigned short`, `unsigned int`
- Tipi floating point
`float`, `double`, `long double`
- Tipo `void`
ha insieme vuoto di valori, per indicare che una funzione non ritorna alcun valore o, usando un cast esplicito, che il valore di una espressione deve essere scartato

```
(void) foo(3); // chiama foo(3) e scarta il risultato qualunque esso sia
```

- Tipo `std::nullptr_t`
tipo puntatore convertibile implicitamente in qualunque altro tipo puntatore, l'unico valore che può assumere è `nullptr`, il puntatore nullo

Composti

- `T&` riferimento a lvalue, dove possiamo scrivere un valore
- `T&&` riferimento a rvalue, riferimento a cosa può stare soltanto a destro di assegnamento (come `a=5`, dove `5` è un valore che non può stare a sinistra)
- `T*` puntatore
- `T[n]` tipi array, dove `n` costante intera valutabile a tempo di compilazione
- `T(T1, ..., Tn)` tipi funzione
- enumerazioni e classi/struct

Qualificati

`const`

L'oggetto può essere accesso solo per lettura e mai scrittura.

Fornito tipo `T`, possiamo fornire la versione qualificata tramite `const T`, stando attenti quando `T` è composto:

```
struct S {
    int v;
    const int c;
    S(int cc) : c(cc) { v = 10; } // creiamo l'oggetto e inizializziamo
    // lista inizializzazione classi basi e membro, 'v' viene creato ma non inizializzato
};

int main() {
    const S sc(5)
    sc.v = 20; // errore: sc è const e anche le sue componenti

    S s(5);
    s.v = 20; // legittimo: s non è const e S::v non è const
    s.c = 20; // errore: s non è const, ma S::c è const
}
```

Lo stesso oggetto può essere modificato a seconda del modo usato per accedervi:

```
struct S { int v; };
void foo() {
    S s;
    s.v = 10; // legittimo
    const S& sr = s; // riferimento a s, qualificato const
    sr.v = 10; // errore: non posso modificare s passando da sr.
}
```

I riferimenti all'oggetto `const` deve essere anche lui stesso un `const`, perché altrimenti si violerebbe la regola base.

volatile

Costanti letterali

Varie sono le disposizioni per definire valori costanti; al valore viene associato un tipo specifico che in alcuni casi dipende dall'implementazione.

- `bool`
false, true
- `char`
'a', '3', 'Z', '\n' (ordinary character literal)
u8'a', u8'3' (UTF-8 character literal)
- `signed char`, `unsigned char`:
(-)
- `char16_t`
u'a', u'3' (prefisso case sensitive)
- `char32_t`
U'a', U'3' (prefisso case sensitive)
- `wchar_t`:
L'a', L'3' (prefisso case sensitive)
- `short`, `unsigned short`
(-)
- `int`
12345

In assenza di suffissi, viene attribuito il primo tipo tra `int`, `long` e `long long` che sia capace di rappresentare il valore. I suffissi sono intesi come:

- `U` per `unsigned` (può comparire insieme agli altri);
- `L` per `long` e `long long`;
- `LL` per `long long`

```
12345L
12345LL
12345U
12345ULL
```

Per i floating point, abbiamo la decisione tra decimale o "scientifica":

```
123.45F // float
1.2345e2F // float

123.45 // double
1.2345e2 // double

123.45L // long double
1.2345e2L // long double
```

Per `void` non c'è alcuna costante letterale.

Per `nullptr_t` sarebbe `nullptr`.

Per i letterali di stringa, come la stringa `"Hello"`, sarebbe un array di 6 caratteri (con il delimitatore, `char[6]`); per le stringhe "grezze", usiamo dei delimitatori per creare sequenze di caratteri per non poi preoccuparci di fare "escaping" dei caratteri.

```
// dentro a 'string' possiamo scrivere di tutto
R"DELIMITATORE(string)DELIMITATORE"
```

Letterali definiti da utente

Possibili letterali definiti da utente: il letterale da noi definito avrà lo scopo di invocare una funzione di conversione implicita, anche lei definita da noi.

Un esempio sono le stringhe stile C++ 2014 con il suffisso `s`, per indicare una conversione implicita da stringhe di tipi C, a stringhe tipo `std::string`.

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

int main() {
    using namespace std::literals;
    std::cout << "Hello"; // stampa la stringa C (tipo const char[6])
    std::cout << "Hello"s; // stampa la stringa C++ (tipo std::string)
}
```





Particolarità dei tipi

Se ci chiedessimo quale fosse il valore massimo di un intero, possiamo usare la libreria standard `limits` per risponderci:

```
#include <iostream>
#include <limits>
int main() {
    std::cout
        << "type\t| lowest()\t| min()\t\t| max()\n"
```

```
<< "int\t| "  
<< std::numeric_limits<int>::lowest() << "\t| "  
<< std::numeric_limits<int>::min() << "\t| "  
<< std::numeric_limits<int>::max() << '\n';  
}
```

type	lowest()	min()	max()
int	-2147483648	-2147483648	2147483647

 metodologie >  CODE >  Types >  MaxInt.cpp

01/03/2023