Tipi, qualificatori, costanti letterali

I tipi fondamentali (non strutturati)

- Booleani: `bool`
- Carattere:
    1. *narrow character type:* `char`, `signed char`, `unsigned char`
    2. *wide character type:* `wchar_t`, `char16_t`, `char32_t`
- Interi standard con segno: `signed char`, `short`, `int`, `long`, `long long`
- Interi standard senza segno: `unsigned char`, `usigned short`, `unsigned int`, ...

Tutti i tipi suddetti sono detti tipi integrali. Booleani, caratteri narrow e short sono detti tipi integrali "piccoli", in quanto *potrebbero* avere una dimensione (`sizeof`) inferiore a int e pertanto sono soggetti a promozioni di tipo.

- Floating point: float, double, long double
- Void: ha un insieme vuoto di valori; serve 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 prodotto
```

- `std::nullptr_t`: è un tipo puntatore convertibile implicitamente in qualunque altro tipo puntatore; ha un solo valore possibile, la costante letterale `nullptr`, che indica il puntatore nullo (non dereferenziabile).

Torna all'indice

I tipi composti

- Riferimenti a `lvalue`: `T&`
- Riferimenti a `rvalue`: `T&&`
- Puntatori: `T*`
- Tipi array: `T[n]`
- Tipi funzione: `T(T1, ..., Tn)`
- Enumerazioni e `class` / `struct`

Torna all'indice

Qualificatore const

I tipi elencati sopra sono detti non qualificati. Nel linguaggio C++ esistono i qualificatori `const` e `volatile`. Nel discorso che segue consideriamo solo il qualificatore `const`, il cui uso è essenziale per una corretta progettazione e uso delle interfacce software e come strumento di ausilio alla manutenzione del software.

Dato un tipo `T`, è possibile fornirne la versione qualificata `const T`. L'accesso ad un oggetto (o una parte di un oggetto) attraverso una variabile il cui tipo è dotato del qualificatore const è consentito in sola lettura (cioè, non sono consentite le modifiche).

> Si noti che nel caso di tipi composti è necessario distinguere tra la qualificazione del tipo composto rispetto alla qualificazione delle sue componenti.

Per esempio:

```
struct S {
 int v;
 const int c;
 S(int cc) : c(cc) { v = 10; } // lista di inizializzaione delle classi basi e dei dati membro
};
```

```
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
}
```

Si noti inoltre che lo stesso oggetto può essere modificabile o meno a seconda del percorso 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'.
}
```

Torna all'indice

Costanti letterali

Il linguaggio mette a disposizione varie sintassi per definire valori costanti; a seconda della sintassi usata, 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: `<nessuna>`
- 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: `<nessuna>`
- int: `12345`

Torna all'indice

Note varie

- L'elenco NON è esaustivo; serve a dare un'idea delle potenziali complicazioni. Per esempio, nel caso degli interi consideriamo solo la sintassi decimale, ma esistono anche:
  - la sintassi binaria (`0b1100`, che rappresenta il numero decimale 12)
    - la sintassi ottale (`014`, che rappresenta il numero decimale 12)
    - la sintassi esadecimale (`0xC`, che rappresenta il numero decimale 12)
- In assenza di suffissi `(U, L, LL)` ad una costante *decimale* intera viene attribuito il *primo* tipo tra `int`, `long` e `long long` che sia in grado di rappresentarne il valore. Il tipo dipende quindi dalla particolare implementazione utilizzata: ad un valore molto grande può essere assegnato il tipo `long` o `long long`.
- Le regole per le altre sintassi (booleana, ottale, esadecimale) prendono in considerazione anche i tipi `unsigned`.
- I suffissi delle costanti intere e floating point sono case insensitive; le convenzioni di solito privilegiano la versione maiuscola (raramente la minuscola) per maggiore leggibilità.

Torna all'indice

Altre costanti letterali

In presenza del suffisso `U`, si sceglie la variante `unsigned`. In presenza del suffisso `L`, l'ampiezza è scelta tra `long` e `long long`. In presenza del suffisso `LL`, l'ampiezza è `long long`. Il suffisso U può comparire insieme a `L o LL`.

Per i ***floating point*** si può scegliere tra notazione decimale e "scientifica":

- float: `123.45F`, `1.2345e2F`
- double: `123.45`, `1.2345e2`
- long double: `123.45L`, `1.2345e2L`
- void: `<nessuna>`
- std::nullptr_t: `nullptr`
- Letterali stringa: `"Hello"`

Il tipo associato al letterale è `const char[6]`, cioè un array di 6 caratteri costanti (5 + 1 per il terminatore 0). E' possibile specificare un prefisso di encoding `(u8, u, U, L)` come nel caso delle costanti carattere, che modifica in modo analogo il tipo degli elementi dell'array.

Nel nuovo standard sono stati implementati anche i letterali di stringa grezza (raw string literal), che utilizzano il prefisso R, un delimitatore a scelta e le parentesi tonde, come segue: `R"DELIMITATORE(...)DELIMITATORE"`

Al posto dei ... è possibile inserire qualunque tipo di carattere escludendo le stringhe "DELIMITATORE(" e ")DELIMITATORE", ad esempio posso scrivere: `R"$(Esempio di scrittura all'interno della stringa)$"`

Torna all'indice

User Defined Literal

Il C++ 2011 ha reso possibile anche la definizione dei cosiddetti letterali definiti dall'utente. Si tratta di una notazione che consente di aggiungere ad un letterale (intero, floating o stringa) un suffisso definito dall'utente: il letterale verrà usato come argomento per invocare una funzione di conversione implicita definita anch'essa dall'utente.

Per esempio, a partire dal C++ 2014, il tipo di dato delle stringhe stile C++ (`std::string`) fornisce la possibilità di usare il suffisso `'s'` per indicare che un letterale stringa deve essere convertito in `std::string`. L'operatore di conversione è definito nel namespace `std::literals`, per cui

```
#include
#include

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)
}
```

Torna all'indice

Gli alias di tipo

Gli alias di tipo, implementati dopo il 2011, utilizzano la keyword `using` e sono utili per riuscire a creare un codice più ordinato, seguendo la filosofia *"write once"*.

```
using typeAlias = int;
// in questo caso `typeAlias` sarà un alias per i tipi interi

int main(){
 typeAlias x = 1; // il tipo di `x` sarà quello attribuito a `typeAlias`
}
```

L'utilizzo degli alias può tornare utile in codici lunghi e complessi così da non riscontrare problemi nel caso in cui avvenagano dei cambiamenti di tipo.

```
#include

using typeAlias = int;

typeAlias fact(typeAlias n){
 if(n == 0)
  return 1;
 return n * fact(n - 1);
}

int main(){
 for(typeAlias i = 0; i < 10; i++){
  std::coutv<< "fact(" << i << ") = " <
```

Gli alias seguono lo scope del blocco in cui si trovano.

Torna all'indice

La keyword auto

Nel 2011 è stata attribuita la keyword `auto`, che precedentemente aveva un altro utilizzo, per inizializzazione di una varibile senza esplicitarne il tipo.

```
#include

int main(){
 auto a = 1; // `a` è di tipo intero
}
```

Torna all'indice

La libreria GMP

Molto spesso i tipi di dato implementati non sono sufficienti per rappresentare le informazioni richieste. Per riuscire a capire qual è il valore massimo che può essere rappresentato si può fare come segue:

```
#include
#include
int main(){

 std::cout << "Il valore massimo rappresentabile da un intero e' " << std::numeric_limits::max() << std::endl;

 std::cout << "Il valore massimo rappresentabile da un long e' " << std::numeric_limits::max() << std::endl;

 std::cout << "Il valore massimo rappresentabile da un long long e' " << std::numeric_limits::max() << std::endl;

 return 0;
}
```

Per "superare" i limiti imposti dai qualificatori di base si possono utilizzare delle librerie apposite. Una di queste è *GNU (multiple precision library)*: una libreria open-source che permette di allocare spazio sulla ram per riuscire a rappresentare i numeri richiesti. Un esempio pratico:

```
#include
#include
#include  // "interfaccia" per C++

using typeAlias = mpz_class; // sfrutto un alias

typeAlias fact(typeAlias n){
 if(n == 0)
  return 1;
 return n * fact(n - 1);
}

int main(){

 for(typeAlias i = 0; i < 50; i++)
  std::cout << "fact(" << i << ") = " << fact(i) << std::endl;

 return 0;
}
```

In fase di compilazione devo esplicitare le librerie:

- gmpxx
- gmp (libreria C)

```
g++ -Wall -Wextra -o fact fact.cpp -lgmpxx -lgmp
```

Torna all'indice