

Modern C++ per chi conosce un po' di C, spiegato di fretta da qualcuno senza alcuna qualifica per insegnare

Naoki Pross

Introduzione

Ultimamente ho così poco da fare che ho deciso di scrivere un corso relativamente spedito di C++. È sorprendente a cosa può portare la noia, non trovate? Okay, iniziando seriamente prima devo presentarvi nella maniera più spiccia possibile un paio di brutte cose.

Cosa serve per iniziare

Non ho voglia di scrivere un corso di programmazione da *zero*, quindi do per assunto che sappiate almeno le fondamenta di programmazione, cosa siano un *compiler* o un *linker* e come funzionino il processo di compilazione del C, siccome è praticamente lo stesso per il C++. Non vi ricordate? Ecco un corso super accelerato:

```
source code > compiler > object > linker > eseguibile  
main.cpp    > g++      > main.o > ld     > my_program
```

Ah e il codice è tutto in inglese, buona fortuna.

Nell'industria il C++ è un casino

Esistendo da troppo tempo (non quanto il C, ma quasi) ogni idiota sulla faccia della terra ha pensato bene di creare una propria variante del C++.

Fortunatamente di recente Bjarne Stroustrup, creatore del C++, ha iniziato a cercare di salvare questo casino aggiornando la *C++ standard library*, dando un paio di direttive su come scrivere il cosiddetto *Modern C++*, versione ≥ 11 .

Quindi da bravo programmatore io in questo documento seguirò le direttive del modern C++, ma sappiate che più o meno la maggior parte della gente non

rispetta queste regole complicando la vita a tutti, quindi preparatevi per quando sarete fuori dalla scuola.

Documenti del C++

Quindi è come il C, ma i file del codice sorgente si chiamano `.cpp` invece di `.c`, giusto? Sarebbe bello; neanche su questo sono d'accordo le persone. Esistono le estensioni `.cpp` e `.cc` per i sorgenti e `.hpp` e `.h` per gli headers. Quale usare? `.cpp` e `.hpp` ovviamente.

Iniziamo con le fondamenta

Arrivando dal C o da un qualsiasi altro linguaggio di programmazione simile, spero sappiate come funzionano le keywords `if`, `while`, `for`, ... quindi le saltiamo per passare alle novità. Prima però vi lascio un programma in C++ minimo, che non fa assolutamente niente.

```
int main(int argc, char *argv[]) {
    return 0;
}
```

Che dovrebbe lasciarsi compilare con un comando come il seguente (o con un qualche maledetto bottone nella vostra grafica, se la preferite).

```
$ g++ -Wall -Werror -std=c++11 -o hello hello.cpp
```

Favoloso. Nulla di nuovo, giusto?

Scope

Una cosa importantissima in C++ è capire cosa è uno *scope*, traducibile dall'inglese come ambito, scopo o raggio.

```
{
    int hello_there = 10;
}
```

Lo *scope* della variabile `hello_there` sono quelle parentesi graffe. Perché come sapete, una volta fuori dalle parentesi `hello_there` cessa di esistere, va *out of scope*. Quindi praticamente ogni elemento come un `if`, un `while` o una funzione creano uno *scope*, utilizzando le parentesi graffe.

Uno scope eredita l'accesso alle variabili del suo scope parente, mentre non è possibile il contrario, ossia:

```

{
    int x = 10;
    {
        int y = 1;
        // we can access 'x' from the outer scope
        x += 10;
    }
    // but we CANNOT access this variable 'y'
    // from the inner scope above
    // this DOES NOT compile
    y = 10;
}

```

Tra l'altro, se ve lo steste chiedendo, è assolutamente valido mettere delle parentesi {} senza alcuna *keyword* come `while` o `if` e si usa per ordinare il codice gestendo gli scopi o *lifetime* delle variabili.

Namespaces

Un problema del C è che siccome, giustamente, non possono esserci funzioni o tipi con lo stesso nome il codice diventa orribile in fretta. Per esempio è tipico (e corretto) prendere le funzioni e i tipi di una libreria con il nome della libreria. Esempio:

```
GtkWidget* gtk_window_new(GtkWindowType type);
```

Ma non essendo una cosa obbligatoria, spesso il codice diventa disordinato quando ci si dimentica di applicare questa regola.

Si introducono dunque i `namespace`, che non fanno altro che raggruppare varie parti di codice sotto un nome comune. Permettendo inoltre funzioni e tipi con lo stesso nome! Per esempio:

```

namespace rs232 {
    int open(unsigned device_id);
}

namespace file {
    const int WRITE = 0;
    int open(const char *path, int mode);
}

```

Che successivamente si possono utilizzare con la seguente sintassi.

```

// calls open() from the 'rs232' namespace
rs232::open(0);

```

```
// calls open() from the 'file' namespace
int fd = file::open("/tmp/log.txt", file::WRITE);
```

Esiste dunque un `namespace` che contiene la *standard library* (spesso chiamata anche STL) che giustamente è chiamato `std`. `std` contiene molte cose molto utili come per esempio la `std::string`.

Allocazione di memoria dinamica

Come sapete è spesso utile se non necessario allocare della memoria “dinamica”, ossia non controllata a tempo di compilazione. In C esistono le funzioni `malloc()` e `free()` per richiedere e liberare della memoria. In C++ la funzione è stata integrata con delle *keywords* quali `new` e `delete`, che si usano come segue.

```
// C: int *data = malloc(10 * sizeof(int))
int *data = new int[10];

// C: free(data);
// data = NULL;
delete data;
```

References

I puntatori del C sono molto utili, non possiamo negarlo, ma come tutti sanno non sono esattamente comodi. Per essi e i loro problemi abbiamo un intero documento dedicato (Pointers, the best worst feature of C).

Dunque malgrado in C++ i puntatori esistono e funzionano esattamente come in C, esiste un nuovo concetto di *referenza*. Vediamo da dove originano con il solito esempio stupido in C.

```
void increment(int *v)
{
    if (v == NULL)
        return;

    (*v)++;
}
```

Che si utilizzerebbe come segue.

```
int x = 10;
increment(&x);
// 'x' is now 11
```

Questa funzione è sorprendentemente complicata rispetto alla semplicità di ciò che deve fare. Insomma non sarebbe più comodo se non si dovesse controllare che non è NULL e se non si dovesse dereferenziare il puntatore? Qualcosa come:

```
void increment(int& v) {
    v++;
}
```

Ecco, questa è una referenza in C++, come un puntatore ma sempre valido e che non si deve dereferenziare. E non necessita nemmeno della & per convertire una variabile in puntatore quando si chiama la funzione.

```
int x = 10;
increment(x); // no need to take the address
// 'x' is now 11
```

Questo ovviamente era il caso di un argomento (parametro) di una funzione, ma le referenze possono esistere anche come i puntatori. Ma non mi pare di averle mai viste utilizzate in questo modo.

```
int x = 10;
int& y = x;

y *= 2;
// 'x' (and 'y' referencing it) is now 20
```

Overloading

La parola *overloading*, tradotta dall'inglese potrebbe essere qualcosa come “sovraccaricare” ed è un concetto estremamente utile. Esso serve a risolvere un problema facile da riscontrare. Mettiamo per esempio che stiamo creando una libreria integrante delle funzioni matematiche.

```
namespace math {
    int pow(int base, int exponent);
    double pow(double base, double exponent);
}
```

Volendo avere le funzioni che funzionano sia per numeri interi che per numeri a virgola mobile a doppia precisione, abbiamo due funzioni con lo stesso nome!

In C si creerebbero degli abomini come `powi()` e `powdf()` per differenziare i due. Ma in C++ non è necessario! Il codice sopra è assolutamente valido e il compiler è abbastanza intelligente da capire quale dei due si vuole utilizzare quando la funzione è chiamata.

```
// uses function with signature
// int math::pow(int, int)
int a = math::pow(2, 3); // 8
```

```
// uses function with signature  
// double math::pow(double, double)  
double v = math::pow(4.0, 0.5); // 2
```

Cosa succede se mischiamo le due cose?

```
double mixed = math::pow(2, 0.5) // ??
```

Esempio con g++ (Ubuntu 7.3.0-27ubuntu1-18.04) 7.3.0

```
ex.cpp:22:37: error: call of overloaded 'pow(int, double)' is ambiguous  
    double mixed = math::pow(4, 0.5);  
                          ^  
ex.cpp:4:9: note: candidate: int math::pow(int, int)  
    int pow(int base, int exponent) {  
      ^~~  
ex.cpp:13:9: note: candidate: int math::pow(double, double)  
    int pow(double base, double exponent) {  
      ^~~
```

Giustamente non compila, perché il compiler non sa quale delle due funzioni si vuole utilizzare. Ci propone però le varianti che ha trovato “`note: candidate`” che potrebbero essere valide.