Seconda lezione

10 March 2022 14:04

[-O3]=parametro di ottimizzazione opzionale (più aumenta, più aumenterà il tempo necessario per compilare).

Tipi fondamentali, puntatori e riferimenti:

1. **Dichiarazione di variabili:**
   - type varname[default value] questa cella di memoria contiene questo valore altrimenti va ad inserirsi una valore qualsiasi, **QUESTO VALORE RANDOMICO NON E' ZERO.**
   - Type varname[numelements] vado a dichiarare un array **conoscendo a momento di compilazione il numero degli elementi.** Questo array viene allocato sullo stack, utile quando il numero di elementi è estremamente piccolo in quanto potrebbe superare il valore dello stack disponibile.
2. **Tipi fondamentali in C++:**
   - bool (true o false)
   - [unsigned] se è 0 il numero è positivo altrimenti è negativo, in questo caso si va ad indicare che **questo bit NON si utilizza per il segno ma per il numero (da 0 a 2^32).**
   - char, short, int, long int, long long int
   - float, double e long double

ESEMPI:

i. unsigned int x=3;
ii. Double vector[3]={0.1, 2, 5.7}
iii. Long matrix[3][2]={{0,1},{2,3},{4,5}}

3. **Tipi const:**
   - Const type varname= value; **DEVE PER FORZA AVERE UNA DIMENSIONE.** Utile anche nel passaggio dei parametri nella chiamata a funzione.

MEGLIO USARE LA CLASSE STRING!!

4. **Puntatori:**
   - **Area di memoria di cui la dimensione NON DIPENDE DAL DATO ed è stabilito dal tipo di architettura. Il contenuto è l'indirizzo del contenuto.**
   - type *varname[default address]
   - Possibile dare il valore iniziale che non è il valore a cui punta ma l'indirizzo

ESEMPI:

i. char c='a';
ii. char *p=&c;
iii. cout<<p<<endl; ma anche **cout<<*p<<endl;**
iv. **NULL POINTER:** p=nullptr; se faccio una referenziazione vado in segmentation fault

RICORDARE: const char c='b';
char *p=&c; **ERRORE!**
const char*p=&c; **OK! PERCHE' INDICO CHE E' UN PUNTATORE IN SOLA LETTURA**

5. **Void pointer:**
   - Puntatore che non sa a cosa sta puntando
   - void*varname[opzionale indirizzo iniziale]
   - Posso puntare a dati disparati a differenza di quelli precedenti che puntano SOLO a caratteri
   - Posso usare il casting se io so a cosa sta puntando.

ESEMPI:

i. void*p=&c; **OK!**
ii. cout<<*p<<endl; **ERRORE PERCHE' NON NE CONOSCO IL VALORE E NE DEVO POI FARE IL CAST**
iii. **FARE IL CASTING: cout<<*((char*)p)<<endl;** oppure **cout<<*(static_const<char*>(p))<<endl;**
iv. Altra soluzione per il casting: char d=*((char*)p);

6. **Tipi di riferimento**
   - Type & varname=…(Lvalues)
   - Si tratta di riferimenti di Lvalues. Se ho un assegnamento, ho due espressioni: una sinistra dell'uguale ed una a destra. Tutti i tipi che stiamo vedendo sono quelli a sinistra che hanno un indirizzo in memoria (a differenza di quelli a destra che hanno valori temporanei ed è per questo che vengono messi temporaneamente sullo stack (stack frame)).
     Quando viene chiamata una funzione a=f()
   - Un altro nome che viene dato ad una variabile
   - Non viene allocata memoria per un riferimento
   - OBBLIGATORIO dire a cosa si riferisce altrimenti non do semantica all'oggetto

uint e ulong indica un unsigned int e long, che con Visual Studio devo scrivere per intero

ESEMPIO:

```
uint d=7;
uint& e=d;          RIFERIMENTO A D che va ad incrementare proprio la d, infatti mi ritrovo come valore 8
e++;
cout<<d;

uint d=7;
uint*p=&d;          Alloca la memoria del puntatore
(*p)++;
cout<<d;

uint d=7;
uint e=d;           Sto incrementando e ma non sto modificando d
e++;
cout<<d;

Int& i=1;           Errore perché si tratta di una costante e dunque non sto assegnando un indirizzo però ciò
```
posso andare a correggerlo con **const int& i=1;**

7. **Copiare il valore di una variabile all'interno di un'altra oppure spostare dei dati senza copiare**
   - Nel primo caso utilizziamo: **una copy constructor**
   - Nel secondo caso utilizziamo: **una funzione move**
   - Fanno utilizzo ai riferimenti di Rvalue: ObjectType var=std::move(f());, supponiamo che la funzione di ritorno restituisca qualcosa quindi conviene fare uno spostamento di dati andando a spostare gli indirizzi senza andare a toccare i dati che verrebbero buttati via, o anche ObjectType &&var=f();
8. **Allocazione dinamica della memoria**
   - **new()** che equivale ad una malloc() **COSTRUTTORE**
   - **delete()** che equivale ad una delete() **DISTRUTTORE**

ALLOCARE SENZA DEALLOCARE E' GRAVISSIMO!

Esempio:

i. **Nel caso di un array**

```
unit num= espressione;
type*var=new type[num];   viene allocata la memoria necessaria
…
delete[]var;  dealloco solo dopo che l'ho utilizzata
```

ii. **Nel caso di un oggetto**

```
ObjectType*var=new ObjectType();
…
delete var;
```

La new() restituisce SEMPRE un indirizzo ma se sfora la memoria a disposizione va a sollevare un'eccezione chiamata "bad_alloc".

Le delete() si trovano all'interno del distruttore all'interno del quale si incapsulano tutte le delete().