Introduzione al C++
Parte 2

Sommario

- Le differenze fra C e C++
  - il qualificatore const
  - i namespace
  - gli stream

Evitare modifiche accidentali

- Il qualificatore const indica che la variabile non può più essere modificata dopo la sua inizializzazione
- Puo' essere usato nella dichiarazione dei tipi dei parametri passati ad una funzione

- **Come e quando utilizzare const:**

  Principio del privilegio minimo:

  Ad una funzione si devono accordare i privilegi di accesso minimi indispensabili ai parametri passati per completare la propria funzione e nulla più

Usare CONST

- Si puo' dichiarare const una variabile globale:

```
const int finger=5;
int main(){
    int hands=2;
    cout<<hands*finger<<endl;
    return 0;
}
```

- ..o una variabile automatica:

```
double f(double a){
    const double pi=3.1415656;
    return 2*a*pi;
}
```

- ..o un parametro

```
double f(double a, const double pi=3.1415656){
    return 2*a*pi;
}
```

Privilegi di accesso

- ..ma l'uso principale e' nel controllare i privilegi di accesso tramite puntatori
- **La questione:**
  - Quando ad una funzione si passa una variabile per copia siamo sicuri di non modificare il parametro originale
  - Ma passando una variabile per indirizzo si potrebbe modificare il parametro originale
  - Come fare se si vuole passare una struttura dati grande tramite puntatori ma la si vuole rendere non modificabile?
- Le possibili combinazioni sono:
  - puntatore non-costante a dato non-costante
  - puntatore non-costante a dato costante
  - puntatore costante a dato non-costante
  - puntatore costante a dato costante

Esempio:

- Puntatore non-costante a dato non-costante

```
void f(int*, const int);

main(){
    int array[]={0,1,2,3,4,5,6,7,8,9};
    int* a=array;
    f(a,10);
}

void f(int* array, const int dim){
    for(int i=0;i<dim;i++){
    *array += 1;
    array++;
    }
}
```

Esempio:

- Puntatore non-costante a dato costante

```
void f(const int*, const int);

main(){
    int array[]={0,1,2,3,4,5,6,7,8,9};
    int* a=array;
    f(a,10);
}

void f(const int* array, const int dim){
    for(int i=0;i<dim;i++){
    cout<< *array;
    array++;
    }
}
```

- Nota: non si puo' modificare il dato puntato

Esempio:

- Puntatore costante a dato non-costante

```
void f(int* const, const int);

main(){
    int array[]={0,1,2,3,4,5,6,7,8,9};
    int* a=array;
    f(a,10);
}

void f(int* const array, const int dim){
    for(int i=0;i<dim;i++)
    *(array + i) += 1; //o equivalente array[i]+=1;
}
```

- Nota: non si puo' incrementare il puntatore

Esempio:

- Puntatore costante a dato costante

```
void f(const int* const, const int);

main(){
    int array[]={0,1,2,3,4,5,6,7,8,9};
    int* a=array;
    f(a,10);
}

void f(const int* const array, const int dim){
    for(int i=0;i<dim;i++)
    cout<<*(array + i);  //si può solo leggere il dato
}
```

- Nota: non si puo' incrementare il puntatore, ne' modificare i dati puntati

Note

- Un puntatore dichiarato di tipo costante deve essere sempre inizializzato (altrimenti è inutilizzabile)

```
int a;
int* const b=&a; //ok: fase di inizializzazione

int* const c;
c=&a; //ERRORE non si può più cambiare il contenuto di c
```

Passaggio di alias

- Quanto detto per i puntatori vale anche per gli alias

```
void f(const int&, int&);
//void f(const int* const, int* const);

main(){
    int a,b;
    a=100;
    b=10;
    f(a,b);
    cout<<a<<" "<<b;
}

void f(const int& data, int& datb){
    datb++;
    cout<<data<<" "<<datb;  //si può solo leggere il dato
}
```

In sintesi

- Per il compilatore il passaggio per alias è equivalente al passaggio per riferimento con puntatori *const*

- Passare un oggetto di grandi dimensioni tramite alias a dato costante combina le prestazioni del passaggio per riferimento con la facilita' di sintassi e la sicurezza del passaggio per valore

Namespace

- **Cosa:** I namespace permettono di raggruppare classi, funzioni e variabili sotto un unico nome.
- **Scopo:** E' un modo per dividere l'ambito di visibilità globale (global scope) in sotto-ambiti (sub-scopes)

- **Sintassi:** Per dichiarare un namespace:

```
namespace identificatore{
    corpo
}
```

- **Esempio:**

```
namespace general{
    int a,b;
}
```

Accesso al namespace

- **Uso:** Per accedere alle classi,funzioni o variabili di un namespace si deve risolvere la visibilità tramite l'operatore di risoluzione di visibilita' ::
- **Esempio:**

```
general::a
general::b
```

Utilità del namespace

- Il concetto di namespace è utile quando vi è la possibilità che vi siano due o piu' un oggetti/variabili globali o funzioni che abbiano lo stesso nome

```
namespace first{
   int var=5;
}

namespace second{
   double var=3.14;
}

void main(){
   cout<<first::var<<" "<<second::var<<endl;
}
```

La direttiva `using`

- Per poter accedere agli elementi di un namespace come se questi fossero definiti nell'ambito globale, si usa la direttiva:

```
using namespace identificatore;
```

- In questo modo non si deve risolvere ogni volta il namespace

```
namespace My{
    double var=3.14;
}
using namespace My;
void main(){
  cout<<var<<endl;
}
```

Il namespace std

- Uno degli esempi più utili di namspace è quello della libreria standard C++
- Tutte le classi, oggetti e funzioni della libreria standard C++ sono definite nel namespace *"std"*

```
#include<iostream>
int main(){
   std::cout<<"Ciao"<<std::endl;
   return 0;
}
```

- oppure:

```
#include<iostream>
using namespace std;
int main(){
   cout<<"Ciao"<<endl;
   return 0;
}
```

Input/Output in C++

- In C++ le operazioni di I/O sono fatte operando su “stream” di bytes.
- Uno stream è una sequenza di byte

- In input lo stream va da i dispositivi (tastiera, memoria di massa, network) verso la memoria principale
- In output la direzione è inversa

Gli oggetti per l'I/O

- Il C++ manipola gli stream con gli *oggetti*:
  - cin: standard input
  - cout: standard output
  - cerr e clog: standard error
- Questi oggetti sono definiti in <iostream>
- In <iomanip> sono definite operazioni per l'I/O formattato

Operatore di inserimento <<

- Le operazioni di output si eseguono tramite l'operatore **<<**

```
#include<iostream>
void main(){
    int num=5;

    cout<<"Hello world";
    cout<<"Hello world"<<flush;
    cout<<"Hello world\n";
    cout<<"Hello world"<<endl;
    cout<<"Hello world "<< (num * num) <<endl;
}
```

- Lo stream fluisce nella direzione delle frecce, dalla memoria verso lo std output

Operatore di estrazione >>

- Le operazioni di input si eseguono tramite l'operatore >>

```
#include<iostream>
void main(){
    int x,y;
    cout<<"Enter two numbers: ";
    cin>> x >> y;
    cout<< "La somma di "<< x << " e "<< y << " è: "<<
      (x+y) <<endl;
}
```

- Lo stream fluisce nella direzione delle frecce, dallo std input verso la memoria

Operatore di estrazione >>

- L'operatore >> restituisce 0 quando raggiunge l'end-of-file

```
#include<iostream>
void main(){
   int num, max_num=-1;
   while(cin >> num)
     if(num>max_num) max_num=num;
   cout<<"Il massimo è: "<<max_num<<endl;
}
```

Input con getline

- **`getline`** è una funzione membro di cin
- **`getline`** prende in ingresso un array di destinazione, una dimensione e un carattere di delimitazione dello stream

```
#include<iostream>
void main(){
  const int SIZE=100;
  char buffer1[SIZE], buffer2[SIZE];

  cout<<"Inserire una frase:"<<endl; //Due parole
  cin<<buffer1;
  cout<<"La frase è: "<< buffer1 <<endl; //La frase è: Due

  cout<<"Inserire una frase:"<<endl; //Due parole
  cin.getline(buffer2, SIZE, '\n');
  cout<<"La frase è: "<< buffer2 <<endl;
  //La frase è: Due parole
}
```

Manipolatori di stream

- **Dove:** Sono definiti in <iomanip>
- **Cosa:** Servono per stabilire l'ampiezza dei campi, la precisione, vari flag di formattazione, etc

- Tratteremo:
  - setbase
  - setprecision
  - setw
  - setf/unsetf

Manipolatore: setbase

- Per cambiare la base di rappresentazione dei numeri in uscita su uno stream si può usare setbase(num) dove num = 10, 8 o 16
- ..oppure usare direttamente i manipolatori: dec, oct, hex

```
int num=100;
cout<<hex<<num;
cout<<dec<<num;
cout<<setbase(8)<<num;
```

- La base rimane modificata fino ad un successivo cambiamento esplicito

Manipolatore: setprecision

- Per controllare il numero di cifre decimali in output si usa setprecision(num)

```
double root2=1.414213562
cout<<setprecision(5)<<root2;
//stampa 1.41421
```

- La precisione rimane modificata fino ad un successivo cambiamento esplicito

Manipolatore: setw

- Per controllare la larghezza del campo in output si usa setw(num)

```
char parola[]="testo";
cout<<setw(2)<<parola; //stampa: testo
cout<<setw(10)<<parola;
//stampa: . . . . . testo
```

- La larghezza ritorna 0 subito dopo una operazione

Manipolatore: setf

- Per controllare il formato scientifico/fisso o la giustificazione in output si usa setf(ios::xxx) dove xxx è un *enum* che può essere:
  - left, right, internal
  - scientific, fixed

```
double num=0.0001
cout<<setw(10)<<setf(ios::right|ios::fixed)<<num;
//stampa: . . . . 0.0001
cout<<setw(10)<<setf(ios::left|
  ios::scientific)<<num;
//stampa: 1E-4
```