



IMPARARE IL C++ in 2 ore e mezzo? Modulo-1

Roberto Santinelli-Perugia

Outline Modulo1

- Introduzione OO-C++.Perché?
 - Getting started
 - Variabili&costanti
 - Statements e blocchi: if in C++
 - Funzioni
 - Classi e oggetti
 - Looping
 - Puntatori
-



I punti dell'object oriented

Mentre all'inizio I programmi dovevano fare meno cose (calcolo e maneggio di raw data) erano semplici ,gli utenti dovevano essere esperti di computer.

OGGI I programmi sono facili da usare ma nel loro cuore racchiudono una grande complessita'

- **Linguaggio compilato (=velocità) e non interpretato (si può runnare l'eseguibile anche su macchine che non hanno il sorgente e il compilatore)**
- **Facilità di sviluppi successivi in base alle esigenze future (capacita' di isolare la parte di programma da modificare).**
- **Divide et impera(=tante componenti,una per ogni azione che si deve svolgere per risolvere un problema complesso)**
- **Possibilità di usare questi “oggetti” anche in altri applicativi(tipico di OO e non di altri linguaggi es.Un programma C++ può chiamare delle classi Java o delle interfaccia a DB Oracle)**
- **Event-driven(=maggiore interazione utente programma non piu' un flusso predefinito ma possibilita' di scegliere come far andare le cose tramite finestre e pulsanti...)**

OO cerca di soddisfare tutte queste richieste provvedendo tecniche per maneggiare le enormi complessita'dei problemi odierni.

In linea di principio trattando dati e le procedure che agiscono sopra i dati stessi come un unico singolo oggetto, autocontenuto e con sue caratteristiche proprie e “adattabili” alle esigenze del momento.

I 4 pilastri di un linguaggio OO

- Encapsulation
 - Data Hiding
 - Inheritance
 - Polymorphism
-

Encapsulation

Un ingegnere che necessita nel suo progetto un resistore non lo costruisce da zero ma semplicemente lo usa. Quello che lui userà sarà un oggetto assolutamente ben definito, di cui conosce le caratteristiche e i risultati di una sua implementazione, ma creato da qualcunaltro

Data-Hiding

- Con l'encapsulation si effettua anche una data hiding: l'ingegnere necessita delle caratteristiche esterne del resistore ma non di come lavora al suo interno (una black box), né delle proprietà intrinseche che appunto sono incapsulate nell'oggetto resistore. Bisogna solo saper usare questo oggetto in base alle nostre necessità.
 - In C++ encapsulation e data-hiding sono rinchiusi in un oggetto chiamato classe
-

Inheritance(derivazione)

Se vuoi una BMW530 con 7 marce anziché 5 con turbo intercooler, nella realtà l'ingegnere di Monaco non progetta da zero una nuova BMW ma prende “spunto” dal modello circolante e aggiunge le nuove caratteristiche.

Il nuovo oggetto (classe) sarà quindi adatto alle nuove esigenze (problemi) anche se non avrà richiesto una mole di lavoro come per crearlo ex-novo ma semplicemente sarà una estensione di qualcosa di preesistente da cui erediterà le caratteristiche usate fino a quel momento allargandole.

Polymorphism

Una classe che è derivata da un'altra viene utilizzata allo stesso modo (es. monto in macchina, giro la chiave, ingrano la marcia accelero) ma porta a risultati differenti.

(6sec100Kmh contro i 10!)

Polimorfismo indica quella proprietà che uno stesso oggetto con uno stesso nome (es. BMW530) può avere associate più forme e quindi conseguenze differenti nello stesso utilizzo delle sue differenti manifestazioni.



Getting started

Getting started

```
1.#include <iostream.h>
2.int main()
3.{
4. Cout << "Ciao a tutti" <<endl;
5.return 0;
5.}
```

Le parti del programma

Un programma in C++ consiste di di variabili, funzioni, classi, ed altri enti che scopriremo di volta in volta.

Ogni volta che si compila un programma entra in funzione anche un preprocessore che guarda nelle linee del tuo codice a cercare un simbolo (#) agendo su queste linee prima del compilatore.

L'istruzione include dice al preprocessore di fare quello che c'e' nelle linee di codice del file fra <>.

Ogni programma inizia con una main() che e' una speciale funzione in quanto non chiamata da altre funzioni ma sempre presente.

Funzione = blocco di codice che fa delle azioni delimitato da parentesi grafe e che deve essere sempre dichiarata nel tipo in uscita e nelle variabili in ingresso (es. `int main()` ritorna un intero)

Cout e' un oggetto usato d'abitudine per printare nello schermo stringhe fra " " "Alla fine ci sono piu' opzioni <<endl; =nuova linea

Il mio programma conosce "cout" grazie a `iostream.h` che l'header della relativa classe implementata di questo oggetto?????(DON'T WORRY)

Uso di cout ,di cin e dei commenti

```
cout <<"Pippo\n"; //nuova linea→ commento
```

```
cout <<"Pippo\t"; //tab
```

```
cout <<"Pippo ha "<< 3+5<<"anni"<<endl;
```

```
/* qui nessuno andra' a leggere nulla se non chi  
sviluppa il codice*/
```

Cin e' un oggetto che come cout viene conosciuto tramite iostream e permette di immettere valori dall'utente ad una data variabile definita prima.

```
cin >>a;
```

Un esempio più complesso

```
#include <iostream.h>
int Add(int x , int y)
{
    cout<<"dentro add(),ricevuti:"<< x<<"e"<<y<<"\n";
    return (x+y);
}
int main()
{
    cout <<"dentro main\n";
    int a,b,c;
    cout<<"dammi due numeri:";
    cin a;
    cin b;
    cout <<"\nSto chiamando la funzione Add()\n";
    c=Add(a,b);
    cout <<"la somma richiesta e':"<<c<<endl;
    return 0;
}
```



Variabili e costanti

Variabili e costanti

I programmi maneggiano dati che sono stored sotto forma di variabili e costanti.

Una variabile e' la label di una locazione di memoria in cui store un valore senza conoscerne l'indirizzo dello spazio riservato (pensa alla rubrica di un cellulare...)

A seconda del tipo con cui dichiarare una variabile il compilatore associa lo spazio di memoria a quella data variabile.

Sizeof() e' una funzione del compilatore che ti dice quanti bytes sono associati ad un tipo di variabile

Tipi di variabili

*buona cosa:dare dei nomi concernenti il tuo problema (evita pippo, a, b,x,y se puoi!)

short int numEvent;

float transvMoment;

long int totNumbOfTrack;

char histoName;

double pseudoRapidity;

Inoltre si puo' dichiarare un intero (corto o lungo) pure **unsigned** (solo positivo 0-65535) o **signed** (anche negativo da -32768 a +32767).

Dichiarazione di variabili

Piu' modi per dichiarare e associare un valore ad una variabile:

1. unsigned short int lunghezza;

unsigned short int larghezza;

lunghezza=5;

larghezza=7;

2. unsigned short int lunghezza=5, larghezza=7;

O addirittura:

3. typedef unsigned short int USHORT;

USHORT lunghezza=5, larghezza=7;

Costanti in C++

Come le variabili le costanti sono locazioni di memoria dove storing dati ma NON cambiano.

Ci sono due tipi di costanti: letterali e simboliche

Letterale:

e' un valore che digiti nel tuo programma quando e' necessario

(es `int myAge=28;` //28 e' una costante letterale)

Simbolica:

`StudentiPerClasse=15;`

`StudentiLiceo=classi*StudentiPerClasse`

(serve solo a rendere piu' comprensibile il codice)

Definizione delle costanti

(vecchio metodo con il preprocessore)

```
#define StudentiPerClasse 15
```

(miglior metodo)

```
const unsigned short int StudentiPerClass=15;
```

(costanti enumerate)

Puoi creare nuovi tipi di costanti per agganciarti alla realta'

```
enum Color {RED,BLUE,GREEN,WHITE<BLACK};
```

In questo modo fai due cose

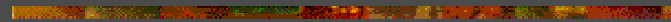
1. Crei Color come il nome di una numerazione che e' un nuovo tipo

2. RED e' una costante simbolica con il valore 0, BLUE 1 e cosi via

Esempio



```
Int main();
{
    enum Days{ Sunday, MonDay, Tuesday, Wednesday, Thursday, Friday, Saturday};
    /* Domenica=0, Lunedì=1....., Sabato=6*/
    Days Dayoff; //dichiara Dayoff una costante del tipo Days
    int x;
    cout<<"che giorno vuoi eliminare?<<endl;
    cin>> x;
    Dayoff=Days(x);
    if (Dayoff != MonDay) cout<<"\nSei completamente pazzo a non eliminare Lunedì";
    return 0;
}
```





Statements e blocchi di statement

Statements e Blocchi

Uno statement controlla la sequenza di esecuzione, valuta una espressione o non fa nulla (null statement). TUTTI gli statement finiscono con il “ ; ”.

```
x=a+b; //assegna a x la somma di a e b
```

Un blocco (compound of statements) e' un insieme di statements eseguiti in cascata che funzionano come un unico statement e sono delimitati da { }

```
{  
temp=a;  
a=b;      //swappa i valori delle variabili a e b  
b=temp;  
} //non c'è la semicolon!
```

Operatori-regole di base.

Assegnamento : `x=10.0;` (=)

Incremento : `c=c+1;` oppure `c++;(++)`

Decremento: `c=c-1;` oppure `c--;(--)`

****** `int a=++x;` (assegna ad a il valore incrementato di x)

`int b=x++` (assegna a b il valore di x e poi incrementa x)

Stesse regole di precedenza che siete soliti con il fortran che possono essere stravolte dall'uso di parentesi.

Operatori relazionali

1. 0=falso

2. Tutto il resto =vero

3. == compara due variabili (o
costante/variabile)

4. >, >=, <, <=, != sono tutti operatori usati per
comparare

Blocco if

Sintassi

If(expression) statement; *se l'espressione e' vera esegue altrimenti*

Oppure *va al next statement*

```
If(expression)  
{  
statements;  
}
```

Uso di else

```
if(expression) statement;  
else statements;
```

If concatenati

```
If (expression1)
```

```
{
```

```
    if(expression2)
        statement1;
```

```
    else
```

```
{
```

```
    if(expression3)
        statement2;
```

```
    else
```

```
        statement3;
```

```
    }
```

```
}
```

```
else
```

```
    statement4;
```

*indentatura come gli spazi e'
ininfluente (solo stile) in C++*

*Da sottolineare solo l'uso delle parentesi
a indicarci chi e' l'else di quale if.....*

AND → &&

OR → ||

NOT → !

Ancora sull'if statement

Operatore Condizionale (ternario) (?:)

E' l'unico operatore a prendere tre termini

(expression1) ? (expression2) : (expression3)

Se l'espressione 1 e' vera ritorna l'espressione 2 altrimenti
l'espressione 3

Un esempio:

$Z = (x > y) ? X : Y$

** In C++ 0 e' falso e non e' raro vedere espressioni tipo

`if(x) x=0;` che serve per azzerare variabili

(leggi come: se x non e' nullo setta il valore zero)



Funzioni

Funzioni

E' a tutti gli effetti un sottoprogramma che agisce su dati e tira fuori un valore.

Ogni programma ha almeno una funzione che e' main().Ogni funzione puo' chiamarne altre purché' **dichiarate prima.**

Ogni volta che si incontra il nome della funzione il flusso del programma entra nella funzione per poi riprendere dalla linea successiva.

Per dichiarare una funzione ci sono due modi:

- o Scrivi il tuo prototipo in un file e quindi usi #include
- o Scrivi il prototipo nel file nel quale la funzione sara' usata
- o Definisci la funzione prima che sia chiamata da ogni altra funzione (funzione A e funzione B:chi deve essere dichiarata prima se l'una chiama l'altra????)

Funzioni: definizioni

Funzione prototipo e' uno statement (;) che consiste: del tipo del valore che ritorna la funzione, del nome e della lista dei parametri che entrano

```
es.    long Area(int base,int altezza=20); //20 e' un default qualora
        //non immetti l'altezza nell'implementazione
```

Definizione di funzione: consiste di un header e del corpo della funzione stessa.

Header:=prototipo con I parametri che devono avere un nome e non ci sono (;) (vedi il primo esempio più complicato all'inizio).

Es.

```
Long Area(long l,long h)           //puoi passare come argomenti
{                                  //anche altre funzioni o se stessa
    return l*h;                   //(recursione)
}
```

Variabili locali e globali

- Locale=definita dentro una funzione(blocco) e visibile solo dentro una funzione (blocco)(vale solo all'interno delle parentesi grafe e non puo' cambiare una variabile con lo stesso nome fuori del blocco).
 - Globale=definite fuori ogni funzione e visibili da ogni funzione del programma.,
(sono molto pericolose in C++ perche' la stessa variabile puo' essere cambiata da piu' funzioni in maniera invisibile alle altre)
-

Funzioni: esempio 1

```
#include <iostream.h>
Void myFunction();
Int x=5,y=7;
Int main()
{
    Cout <<"x from main:"<<x<<"/n";           //scrive 5
    Cout <<"y from main:"<<y<<"/n";           //scrive 7
    myFunction();                               //scrive 5 e scrive 10
    Cout <<"x from main:"<<x<<"/n";           //riscrive 5
    Cout <<"y from main:"<<y<<"/n";           //riscrive 7 e non 10 (pointer)
    Return 0;
}
Void myFunction()
{
    int y=10;
    Cout <<"x from function:"<<x<<"/n";
    Cout <<"y from function:"<<y<<"/n";
}
```

Funzioni:esempio2 multiple return

```
#include <iostream>
Int Doubler (int AmountToDouble);
Int main()
{
    Int result=0;
    Int input;
    cout<<:immetti il numero:"<<endl;           //yti chiede un numero
    cin>>input;                                   //lo immetti da tastiera x
    cout <<"result is:"<< result<<endl;           //scrive 0
    result=Doubler(input);                        //esegue il primo statement della funzioe
    cout <<"result is:"<< result<<endl;           //scrive 2*x o -1 con messaggio d'errore
    return 0;
}
Int Doubler(int original)
{
    If(original<=10000)
        return original*2;
    else
        return -1;
    cout <<"non puoi raddoppiare numeri cosi grossi";
}
```

Funzioni: overloading o polimorfismo

Funzioni con stesso nome e ma differente lista di parametri di input (per numero o tipo) e return type si dicono polimorfe.

Per esempio vuoi un funzione che ti raddoppi qualunque cosa tu gli passi (interi, float, double ..)

Es di prototipo:

```
int Double(int);  
float Double(float);  
double Double(double);
```

Analogamente la definizione di queste funzioni.

A seconda di cosa gli passi nel programma lui usa una delle forme di questa funzione Double.



Classi

Il cuore del C++

Classi

In C++ puoi dichiarare una variabile essere intera ma PUOI ANCHE CREARNE DI NUOVI TIPI.

Def: Classe e' una collezione di variabili combinate da un set di relative funzioni che agiscono su loro e portano ad un effetto che la caratterizza

Una classe porta alla costituzione di un nuovo tipo di variabili che chiameremo piu' generalmente oggetti

Es. Pensiamo alla macchina come ad un nuovo tipo di oggetto fatto da membri interni (ruote, volante, motore) e caratterizzato dalla possibilita' di compiere azioni come accelerare frenare o altro.

Una classe ti permette di incapsulare in un unico misterioso e disegnato ad hoc ente, delle proprieta' che ti servono per il tuo scopo. Non ti preoccupare di come e' fatto! Basta che tu sappia cosa fa e che sta in un punto ben definito del tuo Hard Disk~~~~~

Uso delle classi: creazione di un tipo

Se una classe esiste non la creare: usala o modificala.

Per dichiarare una classe usa la parola chiave “class”

es:

```
class Cat
```

```
{ unsigned int peso;  peso e eta sono le variabili interne al gatto  
                               =membri (anche altre classi potrebbero essere membri)
```

```
    unsigned int eta;
```

```
    Meow();              meow() e'una funzione interna del gatto=metodo
```

```
}; ← c'e' la semicolonna
```

Questa istruzione dice al compilatore cosa e' un gatto quali dati contiene e cosa fa (meow()) ma NON ALLOCA MEMORIA per il gatto che e' un tipo come un intero o un float.

Uso delle classi:creazione dell'oggetto

Int peso;

Cat Frisky;

Frisky e' un oggetto del tipo gatto e come tale avra' un peso e un eta e potra eseguire la funzione meow()

Qui la memoria viene allocata a Frisky

Un oggetto e' una individuale istanza di una classe mentre la classe e' solo l'idea.

Un oggetto di tipo gatto non puo' fare altre azioni oltre Meow().

Per accedere ai membri della classe o ai metodi usi:(.)

Frisky.peso=50;

Frisky.Meow() eseguirà la funzione Meow()

Public vs private

Di default tutti i membri di una classe sono privati (=accessibili solo con metodi di quella classe) mentre public significa che possono essere accessibili da tutti.

Se scrivo nell'esempio di prima

```
Frisky.peso=50;
```

avrò un errore perché di default il peso è privato e si può agire su di esso solo tramite funzioni di quella classe. Dovrò creare una funzione pubblica SetPeso() all'interno dell'idea gatto o dichiaro pubblico il membro peso nella dichiarazione tramite

```
Class cat {
```

```
Public:
```

```
    int peso, eta;
```

```
Private:
```

```
    Meow();}
```

La regola migliore è mantenere le variabili private (visibili solo da metodi interni) e creare metodi pubblici per agire su tali variabili in maniera da mantenere incapsulata la classe

= metodi pubblici di accesso

Uso classi:un primo esempio

```
#include <iostream.h>
class Car
{
public:                                     //metodi pubblici
void Start();
void accelerate();
void setYear(int Year);    //dichiarazione della classe Car
int GetYear();
private:
int theYear;               //variabili private
char Model[255];
};
Car BMW530;
Int immatricolazione;
BMW530.SetYear(i99);
immatricolazione=BMW530.getYear(); //che hanno sara' la BMW?????
```

Costruttori e distruttori

Sono metodi particolari

Int anno; e poi anno=99; // per le variabili normali

Oppure int anno=99;

Stessa cosa la puoi immaginare per le classi. L'inizializzazione combina la dichiarazione e l'assegnazione di valori di default usando speciali metodi che NON ritornano valori ma che prendono parametri con cui settare i membri all'inizio.

Il costruttore per la classe Car si definirà:

```
Class Car
```

```
{
```

```
public:
```

```
    Car(int initialAge);
```

```
    ~Car();           //Metodo distruttore:ogni volta che dichiari un costruttore
```

```
    int GetYear(); int SetYear(int anno);
```

```
private: int TheYear;
```

```
};
```

Implementazione di un metodo

Una funzione accessoria e' un interfaccia fra i dati privati di un oggetto e il resto del mondo.

I metodi come ogni funzione sono implementati dopo essere dichiarati
Ogni definizione di una funzione membro avviene con la sintassi:

tipo Classe-tipo::nome metodo { azione; }

Es int Car::GetYear{ return theYear ;}

Es void Car::SetYear(int anno) {theYear=anno;}

Quindi:

```
int main() {  
    Car BMW530;  
    BMW530.SetYear(99);  
    Cout<< "anno di immatricol:"<<BMW530.GetYear;  
    Return 0; }
```

Implementazione del costruttore

Dopo la dichiarazione di sopra implementi i metodi:

Cat::Cat(int initialAge)

{ theYear=initialAge;} → non ci vuole il ; come ogni fun.

Cat::~~Cat() {} //puo' anche non fare nulla ma lo devi fare

A questo punto quando crei un oggetto tipo BMW530 lo puoi creare anche come:

Car BMW530(99); che assume di default l'anno 99 di immatricolazione

Interfaccia contro implementazione

Una classe da come si vede si divide in due parti :

1. Dichiarazione
2. Implementazione

La maggior parte delle volte tu non devi scrivere ma usare una classe e informi il tuo programma di leggerse la da file con include.

In genere vale la regola che nel file header (.h) si metta la dichiarazione e il path dove trovare il file di dimensione .cc dove invece c'è l'implementazione di tutti i metodi.

Il contratto con il cliente della classe sta nel .h dove sono contenute tutte le informazioni della classe.

Esistono certi metodi implementati nello stesso punto dove sono dichiarati: si usa inline int.....

Classi con altre classi come membri: un esempio

Classi complesse possono anche usare come dati altre classi piu' semplici.

Vediamo questo esempio completo dell'uso di classi in un main.

```
//begin rect.h
```

```
#include <iostream.h> → lo usa solo rect.h e quindi non c'e' bisogno altrove
```

```
Class Point
```

```
{ public:
```

```
Void SetX(int x) {itsX =x;} → implementazione inline
```

```
Void SetY(int y) {itsY =y;} → implementazione inline
```

```
Int GetX() {return itsX;}
```

```
Int GetY() {return itsY;}
```

```
private:
```

```
int itsX,itsY;
```

```
};
```

Segue esempio

```
Class Rectangle
{
Public:
Rectangle (int top,int left,int bottom,int right);
~Rectangle() {}; →inline implementazione
int GetTop() {return itsTop;}
int GetLeft() {return itsLeft;}
int GetBottom() {return itsBottom;}
int GetRight() {return itsRight;}
void SetTop(int Top) {itsTop=Top;}
void SetLeft(int Left) {itsLeft=Left;}
void SetBottom(int Bottom) {itsBottom=Bottom;}
void SetRight(int Right) {itsRight=Right;}
Point GetUpperLeft() {return itsUpperLeft;} //metodo di rectangle di tipo Point
Point GetLowerLeft() {return itsLowererLeft;}
Point GetUpperRight() {return itsUpperRight;}
Point GetLowerRight() {return itsLowerRight;}
Int GetArea();
```

Segue esempio

```
Private:
    Point itsUpperLeft;
    Point itsLowerLeft;
    Point itsUpperRight;
    Point itsLowererLeft;
        int itsTop;
        int itsLeft;
        int itsBottom;
        int itsRight;
};
//end rect.h ora usiamo quanto dichiarato
#include <rect.h>
Rectangle::Rectangle(int top,int left,int bottom,int right)
{
itsTop=top,itsLeft=left,itsBottom=bottom,itsRight=right;
itsUpperLeft.SetX(left);
itsUpperLeft.SetY(top);
itsUpperRight.SetX(right);
itsUpperRight.SetY(top);
itsLowerLeft.SetX(left);
itsLowerLeft.SetY(bottom);
itsLowerRight.SetX(right);
itsLowerRight.SetY(bottom);
}
```


Segue esempio

```
Int rectangle::GetArea()
{
    int width =itsRight-itsLeft;
    Int height=itstop-itsBottom;
        return (Width*Height);
}
Int main()
{
    Rectangle ilMioRettangolo(100,20,50,80);
        int Area =ilMioRettangolo.GetArea();
    cout<<"Area="<<Area<<"/n";
    cout<<"Upper Left Corner coordinate";           //GetUpperLeft()
    cout <<ilMioRettangolo.GetUpperLeft().GetX()<<","; //ricordo che questo metodo e' del tipo Point e quindi posso
    cout <<ilMioRettangolo.GetUpperLeft().GetY(); //chiamare il metodo GetX come ogni altro oggetto del tipo Point
}
    return 0;
```



Loop in C++

(goto, while, do, for)

Looping

1.) goto "label name" // label=nome seguito dai :

(in genere il nome della label e' loop per non perdere il significato)

```
int main()
{
    int counter=0;          //l'uso di goto conduce a spaghetti codes
loop: counter++;
    cout<<counter<<"/n"; //1,2,3,4,5
    if(counter<5) goto loop;
    cout<<"completato"<<counter<<"/n"; /*completato5*/
    return 0;
}
```

Looping

2.) **while(condizione){...}** //loop:ilblocco "{...}" che viene
//eseguito fin quando la condizione iniziale e' VERA.

```
int main() {  
    int counter=0;  
    while(counter<5) {  
        Counter++;  
        Cout<<counter<<"/n"; //1,2,3,4,5  
    }  
    Cout <<"completato"<<conter<<"/n"; //completato5  
    Return 0;  
}
```

Looping:Continue e break

Se vuoi fare un loop prima che sia finito
l'intero set di statements nel blocco while

Usi **continue** (ritorni alla testa del loop)

Se vuoi uscire dal loop prima che sia finita la
condizione usi **break**.

Utile per esempio in loop infiniti tipo while(1)
per cui ti interrompi se certe condizioni
sono raggiunte.

Looping

3.) **Do {...}while;** l'esecuzione del blocco (o del singolo statement) e' garantita almeno una volta.

```
Int main(){  
    int counter=0;  
    do  
    {  
        counter++;  
        cout<<counter;  
    } while(counter<5); /anche se scrivevo <0 scriveva almeno una volta/  
    return 0;}
```

Looping

4.) `for(inizializzazione;condizione;incremento) {...}`

Questo modo combina i tre step in una sola volta.

```
int main() {  
    for(counter=0;counter<5;counter++)  
        cout<<counter; //potevi scrivere {cout<<counter;}  
    cout<<"completo"<<counter;  
    return 0;  
}
```

altre sintassi:

a. `for(I=0,J=0,I<3 || J<5;I++,J+=2)` // uso di `or` e incremento di 2 la `J`

b. `for(;counter<5;counter++)` //non inizializza ma prende il default

c. `for(; ;)` non fa nulla e loopa all'infinito se non c'è un `break` ipotetico

Un esempio di loop concatenati

```
Int main() {
    int righe,colonne;
    char theChar;
    cout<<"quante righe?"<<"\n";
    cin>>righe;
    cout<<"quante colonne?"<<"\n";
    cin>>colonne;
    cout<<"che carattere ?"<<"\n";
    cin>>theChar;
    for(int I=0; I<righe; I++)
    {
        for(int J=0; J<colonne; J++)
            cout<<theChar;
        cout<<"\n";           //va a capo alla fine di ogni loop su J
    }
    return 0;}
```


Switch statement: evitare if troppo concatenati

Valuta una espressione e divide l'azione a seconda dei valori che tira fuori questa espressione

Sintassi:: `switch (espressione)`

`{`

`case valore1 :statement1; //o un blocco {}`

`case valore2 :statement2;`

`.....`

`case valoreN :statementN;`

`case default :statement; // ci switcha se nessun caso e' vero`

`} //altrimenti esce senza far nulla`

Un esempio di ricapitolazione

```
#include <iostream.h>
Enum BOOL {FALSE,TRUE}
typedef unsigned short int USHORT;
USHORT menu();
void DoTaskOne();
void DoTaskMany(USHORT);
int main()
{
    BOOL exit =FALSE;
    for ( ; ;)
    {
        USHORT choice =menu();
        switch(choice)
        { case(1):DoTaskOne();
          break;
          case(2) :DoTaskMany(2);
            break;    //pui usare il break per uscire da un do while loop ma anche da un for infinito e da uno switch statement
          case(3):DoTaskmany(3);
            break;
          case(4): break;
          case(5):exit=TRUE;
            break;
          default: cout<<"please seleziona ancora\n";
            break;
        }    //end switch
        if(exit) break; //esce dal loop infinito!
    }
    return 0; }    //fine del main
```

Un esempio di ricapitolazione

```
USHORT menu()  
{  
    USHORT choice;  
    cout << "**** Menu ****\n\n";  
    cout<< "(1) Scegli uno.\n";  
    cout<< "(2) Scegli due.\n";  
    cout<< "(3) Scegli tre.\n";  
    cout<< "(4) Scegli di riproporrm i il menu.\n";  
    cout<< "(5) Esci.\n";  
    cout << ": ";  
    cin>>choice;  
    return choice;  
}  
  
void DoTaskOne() { cout<<"Hai selezionato la task numero 1 \n";}  
void DoTaskMany(USHORT which)  
{  
    if(which==2)  
        cout<<"Hai selezionato la Task numero 2 \n"; //qui ci poteva essere la chiamata dell'armamento di  
    else                                     //una testata nucleare  
        cout<<"Hai selezionato la task numero 3 \n"; //qui digita solo il numero della task  
}
```

implementazione delle funzioni usate



PUNTATORI

Puntatori

Una variabile e' allocata in un unica locazione di memoria nota come il suo indirizzo che viene assegnato dal compilatore una volta dichiarata la variabile.

Per conoscere l'indirizzo di una variabile basta che usi l'operatore di indirizzo & :

```
int var =10;
```

```
cout <<&var;          //scrivera' qualcosa tipo 0x8fc9:fff4
```

NON e' IMPORTANTE CONOSCERE l'indirizzo di una variabile ma poterlo maneggiare, PUOI storing questo indirizzo in una variabile che chiamiamo PUNTATORE

La sintassi: `int *pAge =0;` //pAge e' il puntatore ad un intero (int) ma e' come se tu avessi dichiarato una nuova variabile di tipo indirizzo con il (*)

In questo esempio e' inizializzato a zero e si chiama **PUNTATORE NULLO**

Puntatori

Se non iniziizzi un puntatore devi specificamente assegnare l'indirizzo di qualche variabile (cioe' e' buona norma evitare di avere wild pointer ovvero puntatori NON assegnati) oppure assegnargli zero se non sai che valore dare (puntatore nullo).

```
unsigned short int Eta=50; //dichiara la variabile e assegna un valore
```

```
unsigned short int *pEta= &Eta; //dichiara un pointer e il valore
```

L'asterisco ha anche un'azione inversa: da un puntatore puoi dereferenziare e conoscere il valore della variabile all'indirizzo di cui il puntatore ha il valore e non solo dichiarare

```
int LaMiaeta=*pEta; (* significa il valore storto all'indirizzo pEta)  
*pEta=5;
```

Per evitare confusioni pensa alla seguente equivalenza

variabile → valore come **puntatore → indirizzo di una variabile**

Perche' I puntatori

Se tu hai gia' una variabile con cui accedere ad un certo dato perche' inventarsi allora una variabile che porta informazioni dell'indirizzo di questa variabile con cui ritrovare quel valore??

Tre sono I motivi

- 1.Maneggiare dati nel free store**
- 2.Accedere ai membri e ai metodi di una classe**
(da combinare con il polimorfismo per event-driven)
- 3.Passare variabili a funzioni per riferimento**

Vedremo ora il punto 1.

Memoria usata

- **Stack** variabili locali di funzioni cancellata quando la funzione finisce
- **Code space** riservata al codice
- **Global space name** variabili globali del programma
- **Registro** funzioni interne come per esempio mantenere traccia dell'inizio dello stack.
- **Free store** tutto il resto che viene cancellato alla terminazione del programma

Primo problema: le variabili locali non persistono e l'uso di variabili globali rende difficile da mantenere il codice.

Puntatori:new e delete

Il comando **new** ti permette di allocare memoria nel free store. New ritorna un indirizzo di memoria del free store.

La sintassi e':

int * pPointer=new int; che ora punta ad un intero nel free store.

Quindi puoi usarlo come ogni altro puntatore: ***pPointer=75;**

(assegna il valore 75 all'area nel free store alla quale punta pPointer)

NB.quando hai finito con la tua area di memoria devi usare il comando delete per liberare la memoria

delete pPointer; (libera la memoria cui pPointer puntava:OBBLIGATORIO)

Il vantaggio nel modo di accedere alla memoria piuttosto che con var.globali sta nel fatto che solo funzioni con accesso al puntatore hanno accesso a quei dati "globali" eliminando il problema che una funzione cambia in modo inaspettato e incotrollato quel dato stesso!

Creare oggetti nel free store

Gia come puoi creare un puntatore ad un intero lo puoi fare per una classe e quindi ad un oggetto

```
Cat *pCat =new Cat;  
delete pCat;
```

(Non chiamare piu' volte delete su uno stesso puntatore:avrai un crash del tuo programma .

La soluzione dopo delete e' settare 0 quel puntatore per essere sicuri al 100%)

Delete implica la chiamata del distruttore di quella classe

Accedere ai membri e ai metodi di una classe tramite il puntatore alla medesima

Accedi alle funzioni/membri di una classe con l'operatore (.). Se però tratti con puntatori prima devi dereferenziare (*) il puntatore all'oggetto: **(*pCar).GetYear();** C'è un modo più compatto:

pCar->GetYear(); Esempio:

```
#include<iostream.h>
class Cat
{
public:
    Cat() {itsAge=2;}
    ~Cat() {cout<<"distruttore.....\n";}
    int GetAge() {return itsAge;}
    void SetAge(int age) {itsAge=age;}
Private:
    int itsAge;
};
```

Puntatori: segue esempio

```
Int main()
{
    Cat * Melissa =new Cat;
    cout <<"Melissa e' un gatto di "<< Melissa->GetAge()<<"anni\n";
    Melissa->SetAge(3);
    cout<<"Melissa ora ha"<<Melissa.GetAge<<"anni\n"; //dove e' l'errore????
    delete Melissa;    //qui eseguirà distruttore.....
    return 0;
}
```

Senza fare un altro esempio e' chiaro che itsAge potrebbe essere a sua volta un puntatore ad un intero e non semplicemente un intero come nell'esempio di sopra (int *itsAge;)

Un puntatore si chiama selvaggio se dopo essere stato cancellato (delete) si prova ad riutilizzarlo: ti verra' dato un errore.

Il puntatore this

Ogni membro di una classe ha un parametro nascosto : il puntatore **"this"** In ogni chiamata di un metodo interno (GetAge(),SetAge()) anche il puntatore this per quell'oggetto e' incluso. Tuttavia spesso e' utile utilizzare esplicitamente this per il potere che hanno i puntatori come nell'esempio

```
void SetLenght(int length) {this->itsLength=length;}
```

THIS stora infatti l'indirizzo di memoria dell'oggetto in questione(theCat,Frisky,BMW520 etc.etc)

Precauzioni.

- Cerca sempre di inizializzare un puntatore (magari anche assegnando 0 quando si puo'.)(WILD POINTERS)
- Ogni puntatore nel free store deve essere cancellato prima di essere riassegnato ad una nuova locazione con NEW (altrimenti non vedi la memoria occupata all' indirizzo del primo puntatore fin quando il programma non termina)
- Evitare di provare ad usare un puntatore che e' stato precedentemente cancellato con delete senza riassegnarlo :STRAY POINTERS
- Evitare di cancellare piu' volte uno stesso puntatore:ad ogni new c'e' un delete.Per essere sicuri dopo un delete assegna 0 al puntatore
- Se new non puo' allocare memoria nel free store ritorna il puntatore nullo:quindi mai assegnare ad un esistente puntatore del free store il valore zero.Questo significa che NON sempre e' possibile inizializzare puntatori, ma ricordati che HAI l'obbligo di assegnarli PRIMA O POI.

E' LECITA DUNQUE ANCHE LA SINTASSI :

```
int *pPointer; //senza inizializzazione mi informa solo che pPointer punta  
              //ad un intero
```

```
_pPointer=new int; //crea nel free store Non so che valore dargli ma non gli  
posso dare zero che e' un valore riservato all'errore!!
```