Guida C++

Aspetti Avanzati

#Name: Guida C++: Aspetti Avanzati
#Author: Giuseppe_N3mes1s
#Licence: GNU/GPL
#E-Mail: dottorjekill_mister@tiscali.it
#Development: 30/12/2008

Con questa Guida vorrei apportare le mie conoscenze sul C++ ad altri utenti, rendendomi partecipe di aumentare un po la conoscenza su questo Linguaggio di programmazione.

La guida presuppone che l'utente abbia gia' una discreta se non ottima preparazione basilare sull'arogomento, in quanto, la codesta non parte da come si crea un “hello world!”, ma il suo percorso inizia da, su cose' la programmazione modulare in C++, fino ad arrivare alle Strutture dati Astratte.

Il tutto e' composto da guide scritte da me, prendendo spunto da testi universitari.

Dopo aver fatto questa semplice premessa possiamo iniziare con l'indice degli argomenti e poi passeremo ad approfondire ognuno di essi.

Saluti e Buon Proseguimento di Studio

Giuseppe_N3mes1s

Indice:

Programmazione modulare in C++

Meccanismi per la modularizzazione

Aspetti concettuali relativi alla modularizzazione

Nella fase del progetto di un software vengono utilizzate adeguate metafore per raggruppare le parti di software che costituiscono nel loro insieme un programma; questa operazione viene detta Modularizzazione del programma e viene realizzata utilizzando opportuni meccanismi di modularizzazione.

Il ciclo di sviluppo dei programmi

Il ciclo di sviluppo di un programma prevede una serie di fasi atte a conferire significative flessibilità:

- la composizione di un programma di grosse dimensioni attraverso piu' moduli;
- l'uso di biblioteche di funzioni gia' compilate;
- il collegamento di moduli gia' precedentemente scritti con diversi linguaggi di origine;
- la possibilita' di porre sul mercato, programmi o moduli non in forma origine ma in forma oggetto; e' questa la scelta di molti produttori per motivi di tutela della proprieta'.

Il ciclo di sviluppo dei programmi ricalca le seguenti fasi che verranno elencate:

Preparazione testo origine:

- Il testo origine viene preparato mediante un editor e viene salvato con estensione .cpp;
- Il codice sorgente di una applicazione viene, in generale, strutturata in piu' moduli, redatti da piu' programmatori;
- Ciascuna unita' di traduzione viene sottoposta separatamente alle successive fasi di precompilazione e compilazione;

La struttura di un generico modulo e' il seguente:

- Direttive di precompilazione;
- Dichiarazioni di nomi di tipo;
- Dichiarazioni di costanti e variabili;
- Dichiarazioni di funzioni (prototipi);
- Definizione di costanti e di variabili;
- Definizione di funzioni.

Precompilazione

- Il testo origine (sorgente) contenuto nel file .c/.cpp viene elaborato da un programma detto preprocessore (precompilatore) che modifica il codice seguendo alcune direttive dategli dal programmatore;

- Le direttive al preprocessore sono contenute nelle righe di testo che cominciano col carattere # (cancelletto, sharp, pound);

- L'output del precompilatore e' codice sorgente che va al compilatore vero e proprio.

Compilazione delle unita'

- Le unita' del programma vengono compilate separatamente mediante l'attivazione del compilatore;

- Per ciascuna unita' di traduzione, il testo origine viene trasformato in file oggetto e viene salvato in un file con estensione .obj (oppure .o).

Il file oggetto

- Il formato del file oggetto e' indipendente dal linguaggio ad alto livello;

- Un file contiene, oltre al codice macchina:

    - una tabella di simboli (funzioni e variabili esterne o globali) definite nel corrispondente file sorgete;

    -- una tabella dei simboli utilizzati ma non definiti nel modulo (quindi definiti in altri moduli);

- Tra le informazioni presenti talvolta vi sono anche i dati per il debugging.

Collegamento (linkage)

- I diversi moduli oggetti costituenti il programma (p1.o,..,pn.o) vengono combinati fra loro ed assieme al supporto a tempo di esecuzione, mediante un programma collegatore(linker);

- Il collegatore traduce il testo eseguibile del programma in un unico file (esempio: prog.exe);

- Anche se il programma consta di un solo file sorgente, alcune librerie sono fondamentali (ad esempio quelle di ingresso/uscita);

- Il linker puo' segnalare errori, tipicamente perche' non e' possibile associare ad un simbolo richiesto da un modulo alcun simbolo tra quelli esportati nei moduli;

- Cio' puo' avvenire in alcuni dei seguenti casi:

-- ci si e' dimenticati di indicare qualche file.cpp al linker;

-- manca l'implementazione di qualche funzione;

-- esistono piu' definizioni di uno stesso nome di funzione o variabile globale.

Ambiente di sviluppo

Un ambiente di sviluppo di programmi di un generico linguaggio offre come strumenti essenziali:

- Compilatore;
- Collegatore;

Alcuni strumenti molto diffusi sono:

- Editor;
- Analizzatori/verificatori di codice;
- Programma di gestione automatica dei passi di compilazione e collegamento (make);
- Debugger;
- Programma di creazione di librerie di modulo oggetto;
- Generatori automatici di codice;
- Software di gestione della configurazione software.

Gli ambienti di sviluppo integrati (detti IDE) raggruppano tali strumenti sotto un'unica interfaccia:

- Borland Turbo Pascal (interfaccia testuale a menu');
- Borland C++ Builder (interfaccia grafica e programmazione visuale);
- Dev-C++ (interfaccia grafica e programmazione visuale);

Modularizzazione

L'uso disciplinato di alcuni meccanismi del linguaggio C++ consente una corretta strutturazione di un programma in moduli.
Tra i principi di meccanismo ci sono:

- la compilazione separata;

- l'inclusione testuale;
- le dichiarazione *extern*;
- l'uso dei prototipi di funzione;

Specifica ed implementazione

E' buona norma tenere separata la specifica di un modulo dalla sua implementazione.
Un programma utente di un modulo A deve conoscere la sua specifica, ma disinteressarsi dei dettagli della sua implementazione. Cio' puo' essere realizzato scrivendo un file di intestazione o *header file* (con estensione .h) contenente le dichiarazioni che costituiscono l'interfaccia di A, ed un file separato per l'implementazione di A.
L'header file deve essere incluso (mediante la direttiva al preprocessore *#include*) nella implementazione, ed ogni modulo utente, affinche' il compilatore possa eseguire i controlli necessari.

esempio:

un programma C++ consiste di piu' file sorgenti che sono individualmente compilati in file oggetto, questi poi sono collegati tra loro per produrre la forma eseguibile.

```
//fact.h
long fact(long);

//fact.cpp
#include "fact.h"
long fact(long f){
 if(f<=1)return 1;
 return f* fact(f-1);
}

//main.cpp
#include "fact.h"
#include <iostream>

int main(){
     cout <<"Fattoriale (7)= "<<fact(7)<<"\n";
     return 0;
}
```

Si noti, fin tanto che l'interfaccia resta intatta, l'implemetazione puo' essere modificata senza dover ricompilare il modulo utente (ma occorre ricollegare i moduli oggetto).
La specifica, contenuta nel file di intestazione, puo' essere guardata come una sorta di contratto sottoscritto tra l'implementatore e l'utente.
Quando piu' programmatori lavorano simultaneamente ad un progetto di grandi dimensioni, una volta accordatisi sulla specifica dei vari moduli, possono procedere

all'impelementazione dei ripettivi moduli indipendentemente l'uno dall'altro.

Librerie dei moduli software

Queste tecniche di sviluppo modulare consentono lo sviluppo su base professionale di librerie di moduli software;
Il produttore di una libreria distribuisce:

- il file intestazione (che devono essere inclusi dall'utilizzatore nel codice sorgente) dei moduli che fanno parte della libreria;
- i moduli della libreria in formato oggetto (gia' compilati), che l'utilizzatore deve collegare assieme ai proprio moduli oggetto;

Tale scelta e' tipicamente motivata da esigenze di tutela della proprieta', ed inoltre evita di dover ricompilare i moduli di libreria.

Perche' dividere il programma in piu' file sorgenti

I vantaggi nel dividere il programma in piu' moduli sergente sono:

- La struttura di un programma medio-grande e' piu' chiara se il programma e' suddiviso in piu' file sorgenti;
- Piu' programmatori possono lavorare allo stesso tempo su file diversi;
- Non occorre ricompilare l'intero programma ogni volta che si effettua una piccola modifica;
- Si possono usare librerie senza dovere includere fisicamente il codice sorgente nel programma(spesso il sorgente non e' disponibile).

L'inclusione multipla

Si puo' verificare che la stessa libreria venga inclusa da due differenti librerie entrambi incluse, a loro volta, in uno stesso programma. In questo caso, in assenza di precise precauzioni, si includerebbero nella compilazione piu' copie dello stesso codice (contenuto nel file header), rallentando il processo di compilazione. Poiche' l'header file contiene esclusivamente dichiarazioni, non ha alcun vantaggio includerlo piu' volte.

Precauzioni

La precauzione per evitare la duplicazione di codice consiste nell'utilizzare opportunamente le direttive di precompilazione.
I particolare si usano le direttive:

- *#ifndef*

- *#define*

- *#endif*

Esempio: l'header file della libreria iostream e' realizzato cosi':

```
//iostream.h
#ifndef _IOSTREAM_H
#define _IOSTREAM_H
... //qui le dichiarazioni della libreria
#endif  /* _IOSTREAM_H */
```

Questo approccio e' sempre utilizzato da chi scrive librerie.

Unicita' delle etichette

Nonostante cio' risolva il problema, si rende necessario che nell'ambito di un intero progetto i nomi delle etichette(es: _IOSTREAM_H) siano univoci. Se questo requisito viene a mancare si possono generare errori dovuti alla mancata inclusioni delle librerie.

Conflitti tra nomi

In progetti di dimensioni medio-grandi e' possibile che ci siano conflitti tra nomi esportati tra due differenti librerie. Questo fenomeno e' noto come "inquinamento dello spazio dei nomi".
Il C tenta di risolvere il problema attraverso l'uso di simboli che "accompagnano" gli identificatori:

- uso del carattere underscore: _identifier.

Il C++ offre una metodologia che risolve sistematicamente il problema:

- il meccanismo dei NAMESPACE.

Dichiarare un namespace

A livello di modulo e' possibile dichiarare un namespace nel quale vengono poi definiti tutti i simboli del modulo:

esempio: studenti.h

```
#ifndef __STUDENTI__
#define __STUDENTI__
namespace p1tlc {
     typedef struct {
```

```
          int prefisso;
          int matricola;
          char* Cognome;
          char* Nome;
     } Studente;
     extern Studente Rappresentante;
}
#endif /* __STUDENTI__ */
```

Utilizzare un namespace

Per utilizzare un nome definito nell'ambito di un namespace bisogna esplicitamente "aprire" lo spazio dei nomi. In caso contrario si incorre in errori di compilazione.

esempi:

```
#include "studenti"
int main() {
     Studente s;   //genera un errore."Studente" non e' un
     identificatore definito
     ...
}

#include "studenti"
int main() {
     p1tlc::Studente s; //OK!
     ...
}

#include "studenti
using namespace Studenti; //"apre" tutto lo spazio dei nomi
int main() {
     Studente s; //OK!
     ...
}
```

Lo standard C++

Convenzionalmente, in C++ gli header file i cui nomi terminano con .h non fanno uso dei namespace. Quelli invece che non hanno estensione, al contrario, li usano.
La libreria standard, quella che definisce anche cin e cout, definisce tutti i simboli nello spazio dei nomi std.

esempi:

```
//stile c
#include <iostream.h>
int main() {
```

```
     cout << "Funziona!\n";
}

//errore
#include <iostream>
int main() {
     cout << "Non funziona!\n";
}

//stile c++ con risolutore di scope
#include <iostream>
int main() {
     std::cout << "FunzionLo standard C++
```

Convenzionalmente, in C++ gli header file i cui nomi terminano con .h non fanno uso dei namespace. Quelli invece che non hanno estensione, al contrario, li usano.
La libreria standard, quella che definisce anche cin e cout, definisce tutti i simboli nello spazio dei nomi std.

esempi:

```
//stile c
#include <iostream.h>
int main() {
     cout << "Funziona!\n";
}

//errore
#include <iostream>
int main() {
     cout << "Non funziona!\n";
}

//stile c++ con risolutore di scope
#include <iostream>
int main() {
     std::cout << "Funziona!\n";
}

//stile c++ con direttiva using
#include <iostream>
using namespace std;
int main() {
     cout << "Funziona!\n";
}
```

La Ricorsione

Il concetto di ricorsione informatica si riconduce al concetto di induzione matematica.

Sia P un predicato sull'insieme N dei numeri naturali e sia vero il predicato P(0); se per ogni K intero, dal predicato P(K) discende la verita' di P(K+1) allora P(n)
e' vero in qualsiasi n.

La dimostrazione della forma induttiva viene dunque svolta secondo i seguenti passi:
Passo base dell'induzione: dimostrare P(0)=vero;
Passo induttivo: dimostrare che per ogni k>0 P(k) -> P(k+1)

Cosi' come nella funzione induttiva la verita' di P(k+1) descende dalla verita' dello stesso predicato di P(k), il calcolo di una funzione ricorsiva avviene mediante
il calcolo della stessa funzione in un passo successivo.

Ricorsione

Il calcolo ricorsivo sta nel fatto di calcolare una funzione attraverso se stessa.

esempio:
la funzione fattoriale N! (per interi non negativi)
a)0!=1
b)se n>0 allora n! =n*(n-1)!

In altre parola la potenza della ricorsione sta nel fatto di poter definire un insieme infinito di oggetti con una regola finita.

Algoritmi ricorsivi

Sono caratterizzati da:

Una funzione generatrice
- x...=g(x.)
- che produce una sequenza di x.,x.,x., ..., x.

Un predicato p= p(x) vero per x=x.
Una funzione f(x) detta ricorsiva tale che
- f(x.)=c[h(x.),f(x..)]
- f(x.)= f(segnato)

Schema di algoritmi ricorsivi

Sono codificati mediante un sottoprogramma che richiama se stesso:

```
P (T1 x) {
      ...
      if p(x) F;
```

```
     else C(S,P);
     R
}

int fatt(int x){
     int f;
     if(x<=1) f=1;
     else f=x*fatt(x-1);
     return f;
}
```

Terminazione

La chiamata ricorsiva deve quindi essere subordinata ad una condizione che ad un certo istante risulti non soddisfatta (il predicato p).
Il numero delle chiamate necessarie viene detto profondita' della ricorsione.

Meccanismo interno della ricorsione

Un sottoprogramma ricorsivo e' un sottoprogramma che richiama direttamete o indirettamente se stesso.

Non tutti i linguaggi realizzano il meccanismo di ricorsione. Quelli che lo utilizzano fanno uso di queste tecniche:

- Gestione LIFO di piu' copie della stessa funzione, ciascune con il proprio insieme di variabili locali
- gestione mediante record di attivazione; un'unica copia del sottoprogramma ma ad ogni chiamata e' associata ad un record di attivazione.

Dalla forma ricorsiva a quella iterativa

Un problema ricorsivo puo' essere risolto in forma iterativa.

Nel caso in cui la ricorsione e' in coda il caso e' semplice:

ricorsivo:

```
P(T1 x) {
     ...
     if p(x) F;
     else{S;P(x);}
     return
}
```

iterativo:

```
while !p(x)
```

```
     S;
F;
```

calcolo del fattoriale:

esempio:

forma ricorsiva:

```
int fatt(const int x){
     int f;
     if (x<1) f=1
     else f=x*fatt(x-1);
     return f;
}
```

forma iterativa:

```
int fatt(const int x){
     int f=1,i;
     for(i=1;i<x;i++)
          f=f*i;
     return f;
}
```

Ricorsione ed iterazione

Ricorsione:

- Uso del costrutto di selezione
- Condizione di terminazione
- Non convergenza

Iterazione:

- Uso del costrutto di iterazione
- Condizione di terminazione
- Loop infinito

A differenza dell'iterazione, la ricorsione richiede un notevole sovreccarico (overhead) a tempo di esecuzione per la chiamata di funzione.

Quando usare/non la ricorsione

Algoritmi che per la loro natura sono ricorsivi piuttosto che iterativi dovrebbero essere formulati con procedure ricorsive.
Ad esempio alcune strutture dati sono inerentemente ricorsive:

- Strutture ad albero
- Sequenze

- ...

E la formulazione di strutture ricorsive su di esse risulta piu' naturale.

La ricorsione deve essere evitata quando esiste una soluzione iterativa ovvia e in situazioni in cui le prestazioni del sistema son un elemento critico

Funzioni: aspetti avanzati

Il passaggio dei parametri

Un parametro e' di INGRESSO per un sottoprogramma se il suo valore viene solo utilizzato (mai assegnato ne' modificato) dal sottoprogramma.
Un parametro e' di USCITA per un sottoprogramma se il suo valore viene assegnato dal sottoprogramma.
Un parametro e' di INGRESSO-USCITA per un sottoprogramma se il suo valore viene prima utilizzato, poi riassegnato o modificato dal sottoprogramma.

I paramatri di ingressi per un sottoprogramma vanno scambiati mediante passagio per valore:

```
void stampa(const int x)
// x parametro formale
{ cout<<x; }

int b;
stampa(b); // b param. effett.
```

Il passaggio per valore di un argomento viene realizzato dal compilatore copiando il valore del parametro effettivo nel corrispondente parametro formale, allocato nello stack frame della funzione.

I parametri di uscita, o quelli di ingresso-uscita vanno scambiati tramite puntatori o mediante passaggio per riferimento.

Il passaggio tramite puntatore e' totalmente a cura del programmatore, e consiste nel fatto di passare per valore il puntatore alla variabile:

```
void incr(int *v) {
     *v += 1;
}

int a = 4;
incr(&a);// ora "a" vale 5
```

I parametri di riferimento

In alternativa in C++ esiste il passagigio per riferimento.

Un parametro di riferimento e' un alias della variabile argomento.
Il passaggio per riferimento e' efficiente in quanto evita la copia del valore.

Il parametro formale in realta' e' un puntatore che contiene l'indirizzo di riferimento del parametro effettivo.
Il passaggio dell'indirizzo (dereferencing del riferimento) e' a cura del compilatore.

```
void incr(int &v) {
     v += 1;
}
int a = 4;
incr (a); // ora "a" vale 5
```

I riferimenti

I riferimenti sono in qualche modo equivalenti a dei puntatori costanti:

C& r equivale a C* const p

La differenza sta nel fatto che nel caso dei riferimenti e' lasciato al compilatore il compito di distinguere tra il riferimento ed il valore riferito;
nel caso dei puntatori il programmatore deve esplicitamente indicare dove vuole operare, se sul puntatore o sul valore puntato.

L'uso dei riferimento e' dunque meno incline agli errori.

Essi devono rispettare le seguenti regole:

- un riferimento deve essere inizializzato quando e' creato
- una volta creato un riferimento non puo' essere modificato
- non e' possibile avere un riferimento NULL(0)

esempio:

```
int x;
int & r = x; //r e' un riferimento ad x
r=3; // r e' un l-value e dunque un indirizzo
int j=r; // r e' un r-value e dunque un valore
```

Gli argomenti di dafault

Gli utlimi oppure tutti i parametri di una lista possono avere dei valori di dafault.

E' il prototipo che deve specificare gli argomenti di default.

Se l'invocazione della funzione non specifica gli ultimi argomenti, questi vengono assunti uguali agli argomenti di dafualt.

esempio:

```
void abc(int x, int y=1, char *p="hello");
          ...
void xyz() {
     abc(1, "yahtzee!");//il compilatore associa yahtzee al
parametro y e segnala errore
     }
```

L'overloading delle funzioni

In C++ e' possibile dare a funzioni diverse lo stesso nome, a condizioni che le funzioni abbiano liste di parametri diversi (in numero e/o tipo).
Il compilatore e' in grado di associare in modo univoco ciascuna chiamata a una della funzioni, distinte in base alla lista degli argomenti.

esempio:

```
// versione A
void incr(int *v) {
     *v++;
}

// versione B
void incr(int *v, int dx) {
     *v += dx;
}

// versione C
void incr(float *v) {
     *v = *v + 1.0;
}

float x=0.;
incr(x); // viene invocata
// la versione C
```

Avere tipi di ritorno diversi non e' sufficiente a distinguere le funzioni.
Nell'istruzione di chiamata infatti puo' non essere evidente il tipo del valore di ritorno.
Un uso poco attento dei parametri di default puo' causare ambiguita'.

```
void ovl(char='a');
void ovl(int=0);

??? ovl();
```

Quando uno stesso nome e' associato a piu' funzioni, si dice che questo nome e' sovraccaricato (overloaded) di significato.

- FIRMA (signature) di una funzione:

nome + tipo dei parametri
- In presenza della chiamata di una funzione overloaded, il compilatore riconosce quella meglio adatta alla chiamata
- Il tipo restituito non fa parte della firma della funzione.

L'overloading delle funzioni si usa:
- Quando si devono definire funzioni concettualmente simili da apllicare a dati di tipo diverso
- L'algoritmo utilizzato da ciascuna funzione e' anch'esso diverso a seconda dei tipi di dato a cui la funzione deve essere applicata.

Le funzioni inline

Le funzioni inline sono funzioni la cui chiamate non sono tradotte in linguaggio macchina dal compilatore mediante salto a sottoprogramma
Il codice della funzione viene inserito in linea, cioe' nel punto della chiamata
L'uso delle funzioni inline deve essere limitato a funzioni dal corpo molto breve altrimenti:
- il miglioramento del tempo di esecuzione e' irrilevante
- l'aumento della dimensione del codice puo' essere notevole

Le funzioni inline devono essere definite nello stesso file in cui sono invocate, ovvero in un file di intestazione (header file) da importare, perche' il compilatore deve conoscerne il corpo per sostituirlo ad ogni chiamata:
- non sempre una funzione inline viene sviluppata in linea dal compilatore
- non puo' essere esportata

Una funzione inline puo' specificare argomenti di dafault:

```
inline int plusOne(int x=0) { return ++x; }

int main()
{
     cout <<"\n inline:"<<plusOne();
     system("PAUSE");
     return 0;
}
```

Introduzione alla programmazione ad Oggetti e il C++

Alcuni principi dell'ingegneria del software:

- Astrazione;
- Modularita';
- Incapsulamento ed information hiding;
- L'importanza delle astrazioni;

L'Astrazione

L'astrazione e' il processo che porta ad estrarre le proprieta' rilevanti di un'entita', ingnorando pero' i dettagli inessenziali.

Le propieta' estratte definiscono una vista dell'entita';
Una stessa entita' puo' dar luogo a viste diverse;

La Modularita'

La modularita' e l'organizzazione in moduli di un sistema, in modo che lo stesso risulti piu' semplice da capire e manipolare.

Un modulo di un sistema software e' un componente che:
realizza un'astrazione;
e' dotata di una chiara separazione tra:
- interfaccia;
- corpo;

L'interfaccia specifica cosa fa il modulo e come si utilizza.
Il corpo descrive come l'astrazione e' realizzata.

L'incapsulamento e information hiding

L'incapsulameno consiste nel "nascondere" e "proteggere" alcune informazioni di un'entita'.
L'accesso in maniera controllata alle informazioni nascoste e' possbile grazie a varie operazione descritte dall'interfaccia.

L'importanza delle astrazioni

Tutti i linguaggi di programmazione offrono diversi tipi di astrazione:

- tipi semplici;
- strutture di controllo;
- sottoprogrammi;

L'astrazione fondamentale dei linguaggi ad oggetti e' l'astrazione sui dati.

Meccanismi di astrazione

Nella progettazione di un sistema software e' opportuno adoperare delle tecniche di astrazione atte a dominare la complessita' del sistema da realizzare.
I meccanismi di astrazione piu' diffusi sono:

- ASTRAZIONE SUL CONTROLLO;
- ASTRAZIONE SUI DATI;

Astrazione sul controllo

Consiste nell'astrarre una data funzionalita' nei dettagli della sua implementazione;
E' ben supportata dai linguaggi di programmazione attraverso il concetto di sottoprogramma.

Astrazione sui dati

Consiste nell'estrarre entita' (oggetti) constituenti il sistema, descritte in termini di una struttura dati e delle operazioni possibili su di essa;
Puo' essere realizzata con uso opportuno delle tecniche di programazione modulare nei linguaggi tradizionali;
E' supportata da appositi costrutti nei linguaggi di programmazione ad oggetti.

Metodologie di progetto: Top-Down e Bottom-Up

La metodologia discendente, cioe' top-down, e' basata sull'approccio di decomposizione funzionale nella definizione di in sistema software, cioe' sull'individuazione
delle funzionalita' del sistema da realizzare e su raffinamenti successivi, da iterare finche' la scomposizione del sistema individua sottoinsieme di complessita' accettabili.

La metodologia ascendente, cioe' bottom-up, e' basata sull'individuazione delle entita' facenti parte del sistema, delle loro proprieta' relazioni tra di esse.
Il sistema viene costruitoassemblando componenti on un'approccio dal basso verso l'alto.

La programmazione procedurale

Usa come metodologia di riferimento la decomposizione funzionale, con approccio discendente quindi top-down.
Si scompone ricorsivamente la funzionalita' principale del sistema da sviluppare in funzionalita' piu' semplici.
Si termina la scomposizione qunado le funzionalita' individuate sono cosi' semplici da permettere una diretta implementazione come funzioni.
Si divide il lavoro di implementazione, eventualmente tra diversi programmatori, sulla base delle funzionalita' individuate.

La programmazione ad oggetti

Si individuano la classi di oggetti che caratterizzano il dominio applicativo:
le diverse classi vengono poi modellate, progettate ed implementate;
ogni classe e' descritta da un'interfaccia che specifica il comportamento degli oggetti della classe.

L'applicazione si costituisce con l'approciio ascendente, bottom-up, assemblando oggetti e individuando la modalita' con cui questi devono collaborare per realizzare le funzionalita' deverse
dell'applicazione.

Tipi di dati astratti

Il concetto di tipo di dato in un linguaggio di programmazione tradizionale e' quello di insieme dei valori che un dato puo' assumere (variabile).
Il tipo di dati stratto include in questo insieme, estendendone la definizione, anche l'insieme di tutte e sole le operazioni possibili su dati di quel tipo.
La struttura dati completa e' incapsulata nelle operazione da essa definite.

Non e' possibile accedere alla struttura dati incapsulata, ne in lettura ne in scrittura, se non con le operazioni definite su di essa .

Interfaccia: specifica del TDA(tipo dato astratto), descrive la parte direttamente accessibile dall'utilizzatore;
Realizzazione: implementazione del TDA.

Produttore e utilizzatore

Il cliente fa uso del TDA per realizzare procedura di un'applicazione, e per costruire TDA piu' complessi;
Il Produttore realizza le astrazioni e le funzionalita' previste per il dato.

Una modifca aola alla realizzazione del TDA non influenza i moduli che ne fanno uso.

Tecniche di programmazione ad oggetti

Si parla di programmazione con oggetti con riferimento a tecniche di programmazione basate sul concetto di oggetto.

Si parla di programmazione basata sugli oggetti (object-based programming) con riferimento alle tecniche di programmazione basate sui concetti di:
Tipo di dato astratto o Classe(tipo);
Oggetto (istanza del tipo).

Si parla di programmazione orientata agli oggetti(object oriented programming, OOP) co riferimento alle tecniche basate sui concetti:

- Classe;
- Oggetto;
- Ereditarieta';
- Polimorfismo.

Linguaggi ad oggetti

E' possibile adottare tecniche di programmazione con oggetti o basate sugli oggetti anche in lunguaggi tradizionali(pascal, c) adoperando opportune discipline di programmazione, aderendo cioe' ad un insieme di regole il cui uso pero' non puo' essere ne

forzato ne verificato dal compilatore.

Un linguaggi di programmazione ad oggetti offre costrutti espliciti per la definizione di entita', oggetti, che incapsulano la struttura dati nelle operazioni su di essa.

Alcuni linguaggi, in particolare il C++, consentono di definire tipo astratti, e quindi istanze di un dato di tipo astratto.

In tal caso il linguaggio basato sugli oggetti presenta costrutti per la definizione di classi e di oggetti.

Esistono dunque linguaggi ad oggetti:

Non tipizzati:
- Es Smalltalk
- E' possibile definire oggetti senza dichiararne il tipo
- In tali linguaggi gli oggetti sono entita' che incapsulano una struttura dati nelle operazioni possibili su di essa.

Tipizzati:
- Es C++, Java
- E' possibile definire tipo di dati astratti ed istanziarli.
- Gli oggetti devono appartenere ad un tipo.
- In tali linguaggi, una classe e' una implementazione di un tipo di dato astratto. Un oggetto e' una istanza di una classe.

Il linguaggio C++

C++ e' un linguaggio di programmazione general-purpose che supporta:
- La programmazione procedurale
- La programmazione orientata agli oggetti
- La programmazione generica

Il c++ e' dunque un linguaggio "ibrido" nel senso che supporta piu' paradigmi di programmazione.

Classi ed oggetti

Nel linguaggi a oggetti, il costrurro classe consente di definire nuovi tipi di dati astratti e le relative operazione connessi ad essi.
Sotto forma di operatori o di funzioni (dette funzioni membro), i nuovi tipi di dato possono essere gestiti quasi allo stesso modo dei tipi predefiniti dal linguaggio:
- si possono creare istanze;
- si possono eseguire operazioni su di esse;

Un oggetto e' una variabile istanza di una classe.
Lo stato di un oggetto e' rappresentato dai valori delle variabili che costituiscono la

struttura dati concreta sottostante il tipo astratto.

Ereditarieta' e Polimorfismo

L'ereditarieta' consente di definire nuove classi per specializzazioni o estensioni per classi gia' preesistenti, in modo incrementale.

Il polimorfismo consente di invocare operazioni su un oggetto, pur non essendo nota a tempo di compilazione la classe a cui fa riferimento l'oggetto stesso.

Ereditarieta'

Il mecaniscmo di ereditarieta' e' fondamentale nella programmazione ad oggetti, in quanto prevede la strutturazione gerarchica nel sistema software da costruire.
L'ereditarieta' infatti consente di realizzare relazioni tra calssi di tipo generalizzazione-specializzazione, in cui, una classe, detta base, realizza il comportamento
generale comune ad un insieme di entita', mentre la classi derivate realizzano comportamenti specializzati rispetto a quelli della classe base.

Generalizzazione: dal particolare al generale
Specializzazione o particolarizzazione: dal generale al particolare

Esiste pero' un altro motivo per utilizzare l'ereditarieta' oltre a quello di organizzazione gerarchica, ed e' quello del riuso del software.
In alcuni casi si ha una classe che non corrisponde proprio alle nostre esigenze, anziche' scartare il tutto e riscrivere il codice, con l'ereditarieta' si puo' seguire
un approccio diverso, costruendo una nuova classe che eredita il comportameto di quella esistente, salve pero' che per quei combiamenti che si ritiene necessario apportare.
Tali cambiamenti posso portatre all'aggiunta' di nuove funzionalita' o alla modifica di quelle esistenti.

In definitiva l'ereditarieta' offre un vantaggio anche in termini di sviluppo perche' riduce i tempi, in quanto:
minimizza la qunatita' di codice da scrivere
non e' necessario conoscere in dettaglio il funzionamento del codice da riutilizzare ma e' sufficiente modificare la parte di interesse.

Polimorfismo

Per polimorfismo si intende la propieta' dell'entita' di assumere forme diverse nel tempo.
Con un'entita' polimorfa si puo' fare riferimento a classi diverse.

Si consideri una funzione Disegna_Figure(), che contiene il seguente ciclo di istruzioni:

```
for i = 1 to N do
A[ i ].disegna()
```

L'esecuzione del ciclo richiede che sia possibile determinare dinamicamente (cioe' a tempo d'esecuzione) l'implementazione della operazione disegna() da eseguire, in funzione del tipo corrente
dell'oggetto A[i].

L'istruzione A[i].disegna() non ha bisogno di essere modificata in conseguenza dell'aggiunta di una nuova sottoclasse di Figura (ad es.: Cerchio), anche se tale sottoclasse non era stata neppure prevista all'atto della
stesura della funzione Disegna_Figure().

Il polimorfismo dunque supporta la proprieta' di estensibilita' di un sistema, nel senso che minimizza la quiantita' di codice che occorre modificare quando si estende un sistema, cioe' si introducono nuove classi e nuove funzionalita'.

Un meccanismo con cui viene realizzato il polimorfismo e' quello del binding dinamico

Il binding dinamico (o late binding) consiste nel determinare a tempo d'esecuzione, anziche' a tempo di compilazione, il corpo del metodo da invocare su un dato oggetto.

Vantaggi della programmazione OO

Rispetto alla programmazione tradizionale, la programmazione object oriented offre vantaggi in termini di:

- modularita': la classi sono i moduli del sistema software;
- coesione dei moduli: una classe e' un componente software ben coeso in quanto rappresentazione di un unica entita';
- disaccopiamento dei moduli: gli oggetti hanno un alto grado di disaccoppiamento in quanto i metodi operano sulla struttura dati interna ad un oggetto; il sistema complessivo viene costruitocomponendo operazioni sugli oggetti;
- information hiding: sia le strutture dati che gli algoritmi possono essere nascosti alla visibilita' dall'esterno di un oggetto;
- riuso: l'ereditarieta' consente di riusare la definizione di una classe nel definire nuove (sotto)classi; inoltre e' possibile costruire librerie di classi raggruppate per tipologia di applicazioni;
- estensibilita': il polimorfismo agevola l'aggiunta di nuove funzionalita' minimizzando le modifiche necessarie al sistema esistente quando si vuole estenderlo.

Ereditarieta'

L'ereditarieta' nella programmazione software

L'ereditarieta' e' di fondamentale importanza nella progettazione ad oggetti, poiche' induce ad una strutturazione di tipo gerarchico nel sistema software

da realizzare. Essa consente infatti di realizzare quel sistema detto gerarchia di generalizzazione-specializzazione, seconda la quale esiste la classe, detta base, che realizza il comportamento generale, comune ad un inseme di entita', mentre le classi derivate realizzano il comportamento particolare degli oggetti della gerarchia, specializzandolo rispetto a quello della base.
Ciascun oggetto della classe derivata e' anche oggetto della classe base ed eredita cosi' tutti i comportamenti della classe base ed in piu' aggiunge i comportamenti implementati in essa. Non e' vero il contrario invece.

L'ereditarieta' come riuso del software

Esiste pero' anche un altro motivo, di ordine pratico, per cui conviene usare l'ereditarieta'. Oltre a quello di descrivere un sistema secondo un ordine gerarchico, ce un altro concetto che e' legato al riuso del software.
In alcuni casi si ha a disposizione una classe che non corrisponde propriamente alle nostre esigenze; anziche' scartare tutto il codice esistente e riscriverlo, si puo' seguire con l'ereditarieta' un approccio diverso, costruendo una nuova classe che eredita il comportamento di quella esistenza, pero' adattandola alle nostre esigenze.
Questo approccio offre il vantaggio di ridurre i tempi dello sviluppo del sotware, perche' minimizza la quantita' di codice da scrivere, riducendo anche la possibilita' di cadere in errori. Inoltre non e' necessari conoscere il funzionamento del codice complessivo, ma e' sufficiente modificare la parte di interesse. Infine poiche' il software e' gia' stato verificato, la parte di test risulta semplificata.

I meccanismi sintattici per la derivazione

Struttura della classe derivata

Supponiamo che sia stata definita una classe base:

File Base.h

```
class Base{
public:
          //funzioni membro F
          void f1(); //funzione membro f1
          ....
          void fn(); //funzione membro fn
private:
          //variabili membro alfa
};
```

Il progettista di una nuova classe, di nome Derivata, puo' definire la nuova classe quale derivata della classe Base:

File Derivata.h

```
#include "Base.h"

classe derivata:public Base{//derivata ereditata da Base

public:
          //funzioni membro G
          void g1();//funzione membro g1
          ...
          void gm();//funziome mebro gm
private:
          //variabili membro beta
};
```

la classe Derivata eredita tutte le funzioni e le variabili membro della classe Base e definisce a sua volta ulteriori variabili membro beta e funzioni G.

Meccanismi e diritti di accesso

Aspetto generale

Per quanto riguarda i diritti di accesso, vale il concetto generale secondo il quale nessun meccanismo puo' violare le protezioni definite dal progettista della classe.
In particolare, una classe derivata non possiede nessun privilegio speciale per accedere alla parte private della classe base: se cosi' fosse si potrebbe infrangere la privatezza di una data classe semplicemente definendo una classe derivata ad essa.
La classe derivata dunque eredita variabile e funzioni membro della classe base ma accede a tali comportamenti secondo i meccanismi che sono stati specificati nella classe base.
Le funzioni membro della classe derivata possono accedere:

- a tutti i membri della classe derivata, indipendentemente dalla loro modalita' di protezione
- ai membri pubblici della classe base

Accesso delle funzioni membro e funzioni utente

Per quanto attiene il programma che utilizza la gerarchia di derivazione, esso puo' definire oggetti della classe base e oggetti della classe derivata e accedere, secondo i consueti meccanismi di accesso, alle componenti pubbliche di Base e Derivata. Inoltre utilizzando i meccanismi forniti dall'ereditarieta', il programma utente puo' accedere alla componenti pubbliche del sotto-oggetto base della classe derivata e puo' applicare a quest'ultimo la funzioni definite dalla classe base.

- Le funzioni membro di Base accedono ai componenti pubblici e privati del sotto-oggetto base;
- Le funzioni membro della Derivata accedono a componenti pubblici del sotto-oggetto base e ai componenti pubblici e privati della parte proprio di Derivata;
- Le funzioni utente di Derivata accedono a componenti pubblici del sotto-oggetto

base e della parte propria di derivata.

Overriding e Overloading

Il meccanismo di ridefinizione viene detto overriding delle funzioni (sovrapposizione con prevalenza) ed e' un meccanismo addizionale e diverso rispetto al sovraccarico della funzioni (overloading).
Il primo prevede che nella classe derivata una funzione venga ridefinita con forma identica a quella della classe base e che l'oggetto della classe derivata sia applicata la funzione ridefinita, in secondo prevede viceversa che due funzioni abbiano nome uguale ma firma diversa e che il compilatore scelga la funzione da chiamare sulla base del miglior adattamento alla forma dell'istruzione chiamata.
I due meccanismi possono coesistere nelle classi derivate, nel senso che una classe derivata ridefinisce una funzione f() della classe base due volte, con la medesima firma(overriding) e con firma diversa(overloading).
Peraltro l'impiego dei due meccanismi rischia di compromettere la leggibilita' del codice.

Modalita' di derivazione privata

Fino ad ora abbiamo visto le modalita' di derivazione pubbliche. E' peraltro prevista anche una modalita' di derivazione privata:

```
class Base{
public:
          void f1(); //funzione membro f1
          void f2(); //funzione membro f2
          int m1; //variabile membro pubblica
private:
          int m2;//variabile membro privata
};
```

File Derivata.h

```
#include "Base.h"

class Derivata: private Base{
//Derivata ereditata nella forma privata
public:
          void g1(); //funzione mebro g1
     //accede a membri pubblici e privati di Derivata n1,n2,f2()
     //accede a membri pubblici di Base m1,f1(),f2()
     //non accede a membri privati di Base m2
          void f2();//ridefinisce f2 di Base
          int n1;//variabile membro pubblica
private:
          int n2;//variabile membro privata
};
```

La modalita' di derivazione non ha riferimento ai diritti di accesso della classe derivata alle componenti dell'oggetto di appartenente alla classe base, ma piuttosto alla trasmissione dei diritti di acceso alle componenti della classe base alle ulteriori classi derivate e alle funzioni che utilizzano oggetti della classe derivata.
La modalita' di derivazione privata blocca l'ereditarieta', nel senso che sia le successive classi derivate sia le funzioni utente non possono accedere alle componenti della classe base.

esempio:

File Base.h

```
class Base{
public:
          void f1(); //funzioni membro f1
          void f2();//funzioni membro f2
          int m1;//variabile membero intera
private:
          int m2;variabile membro privata
};
```

File Derivata.h

```
#include "Base.h"

class Derivata:public Base{ //derivata erediatata da base
public:
          void g1();//funzione membro g1
          //accede a memberi pubblici e privati di Derivata
n1,n2,f2()
          //accede a memberi pubblici di Base m1,f1(),f2()
          //non accede a membri privati di Base m2
          void f2();//ridefinisce f2 di Base
          int n1;/7variabile membro pubblica
private:
          int n2;//variabile membro privata
};
```

Le funzioni membro della classe derivata possono accedere alla componente privata di Base m2 solo attraverso le funzioni di accesso, se definite dal progettista della classe.

Ampliamento dei meccanismi di protezione e di derivazione

Allo scopo di concedere specifici privilegi di accesso alle classi derivate, il C++ ha ampliato le modalita' di protezione e di derivazione, prevedendo oltre alla modalita' pubblica e privata, la modalita' di protezione
e di derivazione protetta.

I membri protetti di una classe B sono accessibili alla classe derivata D ma non agli utenti di B o D. La derivazione protetta rende la classe accessibile da altre classi ma non dagli utenti di queste.
Per quanto attiene alle modalita' di derivazione e di protezione si opera tipicamente come:

- la modalita' di derivazione viene definita pubblica, allo scopo di consentire la trasmissione dei diritti di accesso alle successive classi della gerarchia e agli utenti;
- le funzioni membro vengono definite pubbliche, allo scopo di consentire agli utenti della gerarchia di derivazione di applicare i metodi richiesti agli oggetti della gerarchia;
- le variabili membro vengono definite protette, allo scopo di consentire l'accesso alla struttura dati a tutte le classi della gerarchia di derivazione ma non agli utenti.

Questo e' in un caso generale, ma possono essere ampliate anche con aggiunta di altre accortezze:

- alcune funzioni membro possono essere dichiarate private se si tratta di funzioni di servizio che devono essere utilizzate solo all'interno di una classe;
- alcune funzioni membro possono essere dichiarate protette se si tratta di funzioni di servizio che devono essere utilizzate solo da funzioni membro delle classi derivate ma non dagli utenti.

esempio:

File Base.h

```
class Base{
public:
          //funzioni membro
protected:
          //variabili membro
};
```

File Derivata.h

```
#include "Base.h"
class Derivata:publlic Base{//Derivata ereditata da Base
public:
          //funzioni membro aggiuntive o sostitutive
protected:
          //funzioni membro aggiuntive
};
```

La classe Derivata eredita da Base l'insieme di funzioni membro e di variabili membro e accede ai membri pubblici o protetti di Base. L'utente della classe Derivata accede solo ai membri pubblici di Derivata e Base, ma non ai membri protetti.

Funzioni o classi amiche nella derivazione

Ciascuna classe in una gerarchia di derivazione puo' definire, secondo i consueti meccanismi previsti da linguaggio, funzioni o classe amiche, garantendo ad esse tutti i diritti di accesso.
In particolare, una classe B puo' definire amica una funzione membro di una classe derivata D oppure tutta la classe D. In questo caso D acquisisce diritti completi di accesso a B, ma non puo' trasmettere tali diritti alle
successive classi della gerarchi ne agli utenti.

esempio:

Sia B la classe base, D una classe Derivata contenente una funzione g(). Allo scopo di consentire a g() di accedere alle componenti private della classe B, essa viene definita amica di B:

```
classe B{//classe base della gerarchia
friend D::g();
public:
     ...
private:
     ...
};

class D:public B{//classe derivata
public:
     void g();//g() puo' accedere al sotto-oggetto base
dell'oggetto proprio
     ...
};
```

Un esempio di gerarchia di derivazione

Consideriamo la gerarchia di derivazione di tre classi:

- La classe PERSONA, che e' al classe piu' generale della gerarchia e contiene la struttura dati atta a rappresentare un generico individuo e le funzioni membro atte a definire il comportamento;

- La classe STUDENTE, che e' una specializzazione della classe base, contiene la struttura specifica per rappresentare le informazioni proprie di uno studente e contiene le funzioni membro atte a definire il comportamento specifico
di un elemento di quella classe;

- La classe BORSISTA,che e' un'ulteriore specializzazione della classe studente e contiene altre informazioni specializzate relative alla classe stessa, piu' specifiche

funzioni membro che definiscono il comportamento.

esempio:

Classe PERSONA

File Persona.h

```
class Persona{//classe base
protected:
     char* nome;
     int sesso;
     int eta;
public:
     //costruttore
     Persona(const char*NOME="",int SESSO=1,int ETA=0):nome(new
char[strlen(NOME)+1]),

     eta(ETA),

     sesso(SESSO) {strcpy(nome,NOME);}
     //generica funzione membro
     void chiSei(){
          cout<<"Sono una persona di
nome"<<nome<<"sesso"<<sesso<<"eta'"<<eta<<endl;
     }
};//fine classe Persona
```

Classe Studente

File Studente.h

```
#include "Persona.h"

class Studente:public Persona{//studente derivata da Persona
protected:
     int esami;
     int matricola;
     char* facolta;
public:
     //costruttore
     Studente(const char* NOME="",int SESSO=1,int ETA=0,int
ESAMI=0,long MATRICOLA=111111,char* FACOLTA="")
     //lista di inizializzazione che richiama il costruttore
Persona
     :Persona(NOME,SESSO,ETA),
     esami(ESAMI),matricola(MATRICOLA),
```

```
     facolta(new char[strlen(FACOLTA)+1]){strcpy(facolta,FACOLTA);}
     //fine costruttore studente
     //generica funzione membro
     void chiSei(){
          cout<<"\nSono uno studente di nome"<<nome<<"iscritto alla
facolta'"<<facolta<<"Matricola"<<matricola<<endl;
          }
};//fine classe studente
```

Classe Borsista

File Borsista.h

```
#include "Studente.h"

class Borsista:public Studente{//BOrsista classe derivata da
studente
protected:
     long borsa;
public:
     //costruttore
     Borsista(const char*NOME="",int SESSO=1,int ETA=0,int
ESAMI=0,long MATRICOLA=111111,char*FACOLTA="",long BORSA=10000000)
     //lista di inziailizzazione richiama costruttore studente
     :Studente(NOME,SESSO,ETA,ESAMI,MATRICOLA,FACOLTA),
borsa(BORSA){
     }//fine costruttore Borsista
     //generica funzione membro
     void chiSei(){
     cout<<"\nSono un vorsista di nome"<<nome<<"inscritto alla
Facolta'"<<facolta<<"ho vinto una borsa di studio di
lire"<<borsa<<endl;
     }
};//fine classe Borsista
```

DA NOTARE BENE I DISTRUTTORI NON SONO PRESENTI, MA VENGONO RICHIAMATI IMPLICITAMENTE NELL'ORDINE INVERSO DELLA GERARCHIA DI DERIVAZIONE: DA BORSISTA A STUDETE.

Programma Principale

File Main.cpp

```
#include "Borsista.h"//Borsista contiene al suo interno studente e
persona

main(){
Persona franco("Verdi Franco",1,28);
Studente giorgia("Rossi Giorgia",2,24,5,461125,"Ingegneria");
```

```
Borsista carlo("Bianchi
Carlo",1,27,9,461134,"Ingegneria",1500000);
     franco.chiSei();//richiama Persona::chiSei
     giorgia.chiSei();//richiama Studente::chiSei
     carlo.chiSei();//richiama Borsista::chiSei
}
```

Polimorfismo

Motivazioni per il polimorfismo

In una gerarchia di derivazione, alcune funzioni della classe base sono ridefinite nelle classe derivate. Se si vuole conferire la codice la caratteristica di dinamicita' e di adattamento che consenta di richiamare la funzione adeguata al tipo dell'oggetto per cui essa e' invocata. Polimorfismo sta proprio ad indicare che il software si adegua alla forma (tipo) dell'oggetto in cui viene applicato.
Ad esempio in una gerarchia di derivazione che comprende la classe base Figura e le classi derivate Punto, Cerchio, Cilindro e si consideri una funzione disegna() diveramente definita per ciascuna classe derivata. Sia inoltre A un array di oggetti della classe Figura o delle classi derivate, e si consideri il seguente ciclo:

```
for(i=1;i<=n;i++)
     a[ i ].disegna();
```

Si desidera naturalmente per ciascun specifico oggetto, richiamare la funzione disegna() specializzata.
A tale scopo percio' occorre che sia possibile determinare dinamicamente a tempo di esecuzione, la funzione disegna() da eseguire, in dipendenza del tipo dell'oggetto a[i].
Nella programmazione tradizionale occorre utilizzare in questo caso il costrutto CASE avente quale argomento il tipo dell'oggetto; nella programmazione ad oggetti questa prestazione viene fornita dal sistema. Il polimorfismo percio' nella programmazione ad oggetti migliora la dinamicita' del sistema.
Il polimorfismo permette anche di migliorare l'estensibilita' del sistema, perche' consente di aggiungere una nuova sottoclasse, introducendo varianti minime al codice esistente.
Il polimorfismo viene realizzato nel C++ attraverso un'azione combinatoria del compilatore e del programma collegatore, e consiste nel rinviare al tempo di esecuzione la scelta della funzione da richiamare: in altri termini, il legame tra chiamata della funzione e il codice che la implementa e' rinviato a tempo di esecuzione (LATE BINDING, legame ritardato). Per contrapposizione si usa il termine legame anticipato (EARLY BINDING) nel caso comune in cui la scelta della funzione da richiamare viene compiuta al tempo di compilazione.

Meccanismi sintattici per il polimorfismo

Sia f() una funzione definita di una classe base della gerarchia e ridefinita nelle classi derivate; il C++ consente di definire staticamente(cioe' a tempo di compilazione) il legame

tra la funzione f() e l'oggetto a cui deve essere applicata (binding statico), sia di rinviare il legame a tempo di esecuzione, allo scopo di ottenere il polimorfismo.
Per esprimere il late binding il C++ prescrive quanto segue:

- le funzioni cui e' applicabile il late binding devono essere specificatamente indicate, a tale scopo esse devono essere individuate tramite il qualificatore VIRTUAL e vengono di conseguenza chiamate funzioni virtuali;

- la chiamata di funzione deve assumere una forma particolare per esprimere la richiesta di istituire effettivamente il legame in forma ritardata: nella chiamata della funzione f() l'oggetto proprio deve essere designato per mezzo di un puntatore (o di un riferimento);

In sintesi la notazione e':

```
p -> f()
```

Esegue la chiamata polimorfa di f(), nel senso che la funzione effettivamente chiamata e' quella associata alla classe C cui appartiene l'oggetto c puntato da p.

Dichiarazioni delle classi della gerarchia

```
class B{
public:
     virtual void f();//per f puo' essere eseguito il late binding
     ...
};
class D:public B{//Deredita da B
public:
     virtual void f();// f() viene ridefinita in D
};
```

Nel programma utente si definisce un puntatore alla classe base e a esso viene assegnato l'indirizzo di un qualsiasi oggetto della gerarchia: per attivare il polimorfismo si richiama la funzione virtuale per mezzo del punatatore:

Programma utente della gerarchia

```
B*p;//p punta alla classe base
D d; // d e' un oggetto della classe derivata
p=&d; //p punta a un oggetto della classe derivata
p->f(); //legame ritardato, late binding, richiama la funzione
D::f()

//uso di un array di puntatori
B b1,b2;
D d1,d2;
B* pp[4]={&b1,&d1,&b2,&d2};
for(int i=0;i<4;i++)
```

```
//richiama B::f() o D::f() in relazione al tipo di oggetto puntato
  pp[ i ]->f(); //late binding
```

Funzioni virtuali pure e classi astratte

In alcuni casi, la funzione virtuale e' dichiarata ma non definita nella classe base (funzione virtuale PURA) ed e' invece definita nella classi derivate. La classe base non ha altro che una funzione di interfaccia ma non e' possibile istanziare oggetti di questa classe (la classe viene detta classe ASTRATTA). Per esempio, nela gerarchia Persona (base), Studente, Borsista, la funzione chiSei() della classe persona puo' essere definita come virtuale pura e conseguentemente Persona assume il ruolo di classe astratta:

```
class Persona{
     ...
     //funzione membro virtuale pura
     virtual void chiSei()=0;
};
```

Le successive classi della gerarchia dovranno, in questo caso, definire la funzione chiSei(). In caso contrario la classe derivata eredita la funzione virtuale pura e assume anch'essa rualo di classe Astratta.

Costruttori e distruttori di classi polimorfe

Chiamata di costruttori e distruttori

Come gia' visto in precedenza, costruttori e distruttori non vengono ereditati, dunque ciascuna classe deve provvedere alla loro implementazione.
Il linguaggio non consente di definire costruttori virtuali: la chiamata deve sempre avvenire con binding statico, nell'ordine definito dalla gerarchia di derivazione, in modo da costruire l'oggetto a partire dalla sua parte base fino a pervenire alla parte definita nell'ultima classe derivata.
Per quanto attiene ai distruttori, essi, viceversa, possono (debbono) essere virtuali; Consideriamo il seguente schema:

- B classe base
- D classe derivata, con oggetti con estensione dinamica
- p tipo puntatore a B

ed il frammento di programma:

```
p=new D; //istanzia oggetto di tipo D, con relativa parte dinamica
....
delete p; //dealloca oggetto p
```

L'operatore new richiama il costruttore della classe D che provvede ad istanziare la parte dinamica; l'operatore delete richiama il distruttore per deallocare la parte dinamica.
Il distruttore richiamato viene individuato a mezzo del puntatore p all'oggetto da

deallocare: nel caso di distruttore non virtuale viene richiamato il ditruttore della classe B, essendo p di tipo B*, nel caso del distruttore virtuale viene richiamato quello della classe D e successivamen te quello della classe B (i distruttori vengono richiamati a cascata: a partire dall'ultima classe derivata fino ad arrivare alla classe base risalendo al gerarchia).
In conclusione in una gerarchia di classi che presentano piu' oggetti con estensioni dinamiche occorre definire i relativi distruttori e essi devono essere virtuali.

Polimorfismo all'interno dei costruttori e distruttori

Dall'interno dei costruttori e distruttori possono, in via generale, essere richiamate funzioni membro della gerarchia di derivazione; in questo caso la chiamata viene sempre espressa con legame anticipato.
In altri termini non si puo' applicare il polimorfismo. Il tutto trova giustificazione nel fatto che, in caso contrario, a mezzo di una chiamata polimorfa potrebbe essere chiamata da un costruttore una funzione definita nella parte basse della gerarchia, operante su un componente dell'oggetto non ancora costruito.

Struttura della gerarchia di classi

Funzioni associate alle classi

Allo scopo di meglio esaminare le caratteristiche funzionali di una gerarchia di derivazione dotata di funzioni membro ordinate e virtuali e' opportuno introdurre il concetto di FUNZIONE MEMBRO ASSOCIATA A UNA CLASSE in una gerarchia di derivazione.
Considerando dunque una generica gerarchia di derivazione, con modalita' di derivazione pubblica. Ciascuna classe abbia inoltre funzioni membro pubbliche.
Data una classe C della gerarchia, consideriamo l'insieme U costituito dall'unione di tutte le classi che appartengono ai percorsi che congiungono C con il nodo radice. Possiamo dunque ora individuare le funzioni membro e le variabili membro della
classe C.

- variabili membro: la classe C ha per variabili membro l'unione delle variabili membro della classi dell'insieme U;
- funzioni membro: a ciascuna classe C sono ASSOCIATE le funzioni membro dell'insieme U.

L'utente della classe C puo' accedere a tutte le funzioni membro associate alla classe C, sia che esse siano definite all'interno di classi che precedono C lungo la gerarchia di derivazione.
Nel caso di OVERRIDING (cioe' di ridefinizione di una funzione da parte di una classe derivata), la funzione membro ridefinita sostituisce quella che precede ai fini della successiva gerarchia di derivazione.
Percio' con riferimento a C possiamo considerare le seguenti funzioni membro:

- funzioni definite in classi che precedono C;
- funzioni definite nella classe C;
- funzioni ridefinite in C (overridding);
- funzioni ridefinite in classi che precedono C;

Ciascuna funzione potra' essere, non virtuale, virtuale, o virtuale pura.

Accesso alle funzioni membro della gerarchia

Consideriamo ora le notazioni per l'accesso alle funzioni della gerarchia di derivazione da parte di un utente della gerarchia:

- la notazione o.f() da' luogo ad un binding statico e richiede dunque che il legame con la funzione da richiamare sia definibile al tempo di compilazione;
- la notazione p->f() e' equivalente alla o.f() per le funzioni non virtuali, nel senso che la funzione richiamata viene definita dal compilatore "a partire" dalla classe O cui punta il puntatore;
- la notazione p->f() per funzioni virtuali da' luogo ad un binding dinamico: la funzione richiamata viene definita "a partire" non dalla classe O cui punta il puntatore, ma dalla classe O primo cui appartiene l'oggetto puntato da p;
- e' in ogni caso possibile un'indicazione esplicita della funzione da richiamare a mezzo del nome della classe con le notazioni: o.X::f() oppure p->X::f() in questo caso il binding e' sempre statico indipendentemente dalla funzione.

In base alle regole del linguaggio si hanno ancora i seguenti casi:

- con la notazione o.f() si fa riferimento a una funzione definita nella classe O oppure a una classe che precede O nella gerarchia di derivazione;
- con la notazione p->f() si fa riferimento a una funzione definita nella classe O cui punta p oppure a una classe che segue O nella gerarchia di derivazione: infatti a p puo' essere assegnato l'indirizzo di un oggetto della classe O oppure di una classe derivata da O(up-casting);
- con la notazione o.X::f() oppure p->X::f() si puo' richiamare una funzione definita nella classe O oppure in una classe che precede O lungo la gerarchia di derivazione; quest'ultima eventualita' (down-casting) e' da evitare perche' l'oggetto a cui viene applicata la f() e' "piu' piccolo" dell'oggetto proprio della classe X.

Le classi: notazioni di base nel C++

La classe e' un modulo software con le seguenti caratteristiche:

- e' dotata di un'interfaccia (specifica) e di un corpo (implementazione);
- la struttura dati "concreta" di un oggetto della classe e gli algoritmi che ne realizzano le operazioni sono tenuti nascosti all'interno del modulo che implementa la classe;

- lo stato di un oggetto evolve unicamento in base alle operazioni ad esso applicate;
- le operazioni sono utilizzate in modalita' che prescindono dall'aspetto implementativo, in tal modo e' possibile modificare gli algoritmi senza modificare l'interfaccia.

Le classi nel C++

Il linguaggio C++ supporta esplicitamente la dichiarazione e la definizione di tipi astratti da parte dell'utente mediante il costrutto CLASS; le instanze di una classe vengono dette oggetti.
In una dichiarazione di tipo CLASS bisogna specificare sia la struttura dati che le operazioni consentite su di essa. Una classe possiede in generale una sezione pubblica ed una privata.

La sezione pubblica contiene tipicamente le operazioni, dette anche metodi, consentite ad un utilizzatore della classe. Esse sono tutte e solo le operazioni che un utente puo' eseguire in maniera esplicita od implicita, sugli oggetti.

La sezione privata comprende le strutture dati e le operazioni che si vogliono rendere inaccessibili dall'esterno.

esempio:

```
class Contatore {
 public :
      void Incrementa(); // operazione incremento
      void Decrementa(); // operazione decremento
 private:
      unsigned int value; // valore corrente
      const unsigned int max;// valore massimo
};
```

Produzione ed uso di una classe

Il meccanismo delle classi e' orientato specificatamente alla riusabilita' del software
Occorre dunque fare riferimento ad una situazione di produzione del software nella quale operano:

il produttore della classe il quale mette a punto:
- la specifica, in un file di intestazione (header file)
- l'implementazione della classe (un file separato)

l'utilizzatore della classe, il quale ha a disposizione la specifica della classe, crea oggetti e li utilizza nel proprio modulo.

Relazioni d'uso tra le classi

Il modulo utente di una data classe puo' essere il programma principale oppure un altra classe.
Tra le due classi puo' esserci una relazione d'uso, in cui una svolge il ruolo di utente dei servizi offerti dall'altra.

Ruolo "cliente"
Svolto dalla classe che utilizza le risorse messe a disposizione dall'altra

Ruolo "Servente"
Svolto dalla classe che mette a disposizione le risorse utilizzate dall'altra

Uso di una classe da parte di un modulo utente

Un modulo utente della classe:
- include la specifica della classe (contenuta nel file nomeClasse.h)
- definisce oggetti istanze della classe e invoca i metodi sugli oggetti

Istanziazione degli oggetti

Un oggetto e' un'istanza esemplare della classe.
Due esemplari della stessa classe sono distinguibili soltanto per il loro stato, sono i valori dei dati membro, mentre il comportamento potenziale e' identico.

Uso di una classe

La specifica di una classe

Rappresenta un'interfaccia per la classe stessa in cui sono descritte:
- le risorse messe a disposizione ai suoi potenziali utenti
- le regole sintattiche per il loro utilizzo

E' separata dalla implementazione, permette l'utilizzo senza che gli utenti conoscano i dettagli della implementazione.
E' a cura dello sviluppatore della classe.
E' scritta in un apposito file di intestazione.

esempio:

SPECIFICA DELLA CLASSE

```
/ / nome del file C.h
  class C {
     public:
        //prototipi delle funzioni membro
```

```
        T1 f1( ....) ;
        T2 f2( ....) ;
     private:
        //struttura dati interna
        int i;
        char c;
  } ; //fine specifica della classe C.
```

L'implementazione di una classe

E' la codifica in C++ delle singole operazioni presentate nell'interfaccia della classe.
E' una particolare soluzione, perche' puo' cambiare l'implementazione senza che cambi l'interfaccia.
E' a cura dello sviluppatore della classe.
E' scritta in un apposito file di implementazione (estensione .cpp)

esempio:

```
//implementazione della classe
  //nome del file C.cpp
  #include "C.h"
  T1 C::f1(...){
     //realizzazione della funzione f1
     }
  T2 C::f2(...){
     //realizzazione della funzione f2
     }
  //fine del file C.cpp
```

Struttura degli oggetti

Ciascuno oggetti, che e' istanza di una classe, e' costituito:

- da una parte base, allocata per effetto della definizione dell'oggetto nel programma utente in un area di dati statici, nell'area stack o nell'area heap, in base alle classi di memorizzazione;
- da una eventuale estensione, allocata in un area heap.

Ciclo di vita di un oggetto

- definizione dell'oggetto
- allocazione
- inizializzazione

- deallocazione
- definizione dell'oggetto: a cura del programma utente
- allocazione parte base: a cura del compilatore
- allocazione eventuale estensione: mediante una speciale funzione membro detta COSTRUTTORE (a cura percio' del produttore della classe)
- inizializzazione oggetto: a cura del costruttore
- deallocazione eventuale estensione: a cura di una speciale funzione membro detta DISTRUTTORE
- deallocazione parte base: a cura del compilatore

Costruttori

Il costruttore e' una funzione membro:
- che ha lo stesso nome della classe
- non restituisce nessuno risultato(nemmeno void)

Invocazione dei costruttori

Ogni istanziazione di un oggetto della classe produce l'invocazione di un costruttore
l'oggetto e' creato come variabile globale o locale:
```
Account act(100);
```
l'oggetto e' creato come variabile dinamica:
```
Account *act = new Account(100);
```

Se il compilatore non individua il costruttore da chiamare produce un errore.

Se l'oggetto ha dei dati membro di tipo classe, su ciascuno di essi e' chiamato ricorsivamete il costruttore
- ciascuno dei dati membri sono chaimati secondo l'ordine di dichiarazione dei dati
- la chiamata dei costruttori dei dati membro precede la chiamata del costruttore dell'oggetto

esempio:

Programma utente della classe C
```
#include "account.h"

main(){
//Allocazione di un oggetto
Account act(100);//viene invocato il costruttore
```

```
   //act e' una variabile automatica che viene allocata nell'area
stack
   //allocazione dinamica di un oggetto
   Account *pact= new Account(100);//invoca il costruttore
   //il puntatore pact e' una variabile automatica allocata in
area stack;
   //l'oggetto puntato e' allocato in area heap;
   }//fine programma utente
```

Distruttori

Il distruttore e' una funzione membro che:

- e' necessaria solo se l'oggetto presenta un'estensione dinamica
- ha lo stesso nome della classe preceduto da ~(tilde)
- non restituisce nessun valore (nemmeno void) ne' ha alcun parametro
- ha lo scopo di deallocare l'estensione dinamica di un oggetto
- non puo' essere invocato esplicitamente dal programma utete, ma viene invocato implicitamente dal compilatore quando termina il ciclo di vita di un oggetto

Invocazione dei distruttori

Se una classe fornisce un distruttore, questo' e' automaticamente chiamatao ongi volta che un oggetto della classe deve essere deallocato:

- l'oggetto e' locale e si esce dal blocco in cui e' stato dichiarato
- l'oggetto e' nello heap e su di esso viene eseguito un DELETE
- l'oggetto e' dato membro di un altro oggetto e quest'ultimo viene deallocato in questo caso:
  - i distruttori dei dati membri vengono invocati nell'ordine di dichiarazione dei dati
  - la chiamata dei distruttori dei dati membri seguono la chiamata del distruttore dell'oggetto

Se una classe non fornisce un distruttore ne viene creato uno di default che invoca ordinatamente i distruttori di tutti i dati membro.

esempio:

```
 //programma utenta della classe C
   #include "account.h"
   void f(){
   //Allocazione dinamica di un oggetto
   Account *pact=new Account(100);
   if(...){
      Account act(100);//invoca il costruttore
      ...
      //fine del blocco, viene invocato il distruttore, e act
viene deallocato dall'area stack
      }
```

```
   delete pact;//vieni invocato il distruttore per l'area heap
puntata da pact
   }//fine procedura f
```

Tipi di costruttori

- Con zero argomenti
- Con uno o piu' argomenti
- Di copia

esempio:

```
 //specifica della classe
   //nome del file C.h
   class C{
      public:
         //funzioni costruttore
         C();//costruttore a zero argomenti
         C(valori iniziali);//costruttore con piu' argomenti
         C(const C& c1);//costruttore di copia
      private:
         //struttura dati
   }//fine della specifica della classe C
```

Le funzioni membro INLINE

Le funzioni membri inline posso essere definite in uno dei seguenti due modi:

- usando la parola chiave inline nella definizione della funzione membro
- senza la parola chiave inline ma inserendo il corpo della funzione nella stessa interfaccia della classe

I dati membro STATIC

I dati membro, pubblici o privati, di una classe posso essere dichiarati static.

Sono dati creati ed inizializzati una sola volta, indipendentemente dal numero di oggetti istanziati:

- vanno dichiarati nella definizione della classe e definiti ed implementate separatamente
- usando l'operatore di risoluzione dello scope, i dati pubblici, come le normali variabili globali, possono essere usati dovunque nel programma

Le funzioni membro STATIC

Le funzioni membro, pubbliche o private, di una classe posso essere dichiarate static.

Sono funzioni che non ricevono l'argomento THIS e quindi:
- possono accedere solo ai dati membro statici della classe
- possono invocare solo funzioni membro statiche della classe

Le funzioni membro CONST

Le funzioni membro che non vanno a modificare l'oggetto devono essere esplicitamente dichiarate const.
La parola chiave CONST segue la lista degli argomenti e deve essere presente sia nel prototipo della funzione membro che nella definizione.

Gli oggetti CONST

Lu funzioni membro const sono le uniche che possono essere chiamate su un oggetto const (a parte i costruttori ed i distruttori)

Il costruttore di COPIA

Il costruttore di copia di una classe C ha la forma:

```
C(const C&)
```

Viene utilizzato quando si deve realizzare la copia di un oggetto esistente:
- nel creare un clone di un oggetto
- nel passare l'oggetto per valore
- nel restituire l'oggetto per valore

Copia superficiale e copia profonda

Quando la classe non fornisce un costruttore di copia, il compilatore ne crea uno di default che effettua la copia superficiale dell'oggetto("shallow copy").
Consiste nell'applicare ordinatamente il costruttore di copia ad ogni dato membro (per i dati relativi la copia avviene con l'operazione sui primi byte relativi)

Quando tra i dati membri dell'oggetto vi son puntatori, non e' opportuno usare la copia superficiale ma bensi' una copia profonda ("deep copy").
In questo caso il costruttore di copia deve essere esplicitamente programmato.

Nel caso dello shallow copy:
I caratteri di due stringhe sono condivisi ("sharing")
- modificando il carattere di una stringa si modifica anche l'altra stringa
- se una delle due stringhe e' deallocata, il distruttore dealloca anche l'array di caratteri, lasciando cosi' la'tra stringa in uno stato scorretto, con un puntatore "dangling".

Nel caso del deep copy:
- Ogni stringa ha la sua copia di caratteri
- qualunque operazione su una stringa non modifichera' l'altra.

L'operatore di assegnazione

L'operatore di assegnazione di una classe C ha la seguente forma:

```
C& operator=(const C&);
```

La sua implementazione differisce da quella del costruttore di copia:
- deve verificare che non si tratti di un autoassegnazione (x=x)
- deve distruggere l'oggetto valore attuale della variabile a cui si assegna
- deve restituire il riferimento alla variabile stessa per permettere catene di assegnazione (x=y=z).

Puntatori in C++

Il C++ prevede puntatori a funzioni e puntatori a dati di tutti i tipi, semplici o strutturati.

Tipo puntatore

E' un tipo che puo' assumere come insieme di valori tutti gli indirizzi di memoria.
Un puntatore ad una variabile di tipo T identifica l'indirizzo di memoria della variabile.

Percio' una variabile di tipo puntatore e' una variabile che contiene l'indirizzo di memoria della variabile puntata.

```
typedef int * pint;
 pint pcount; //variabile di tipo puntatore
 int count=10;
 pcount=&count; // al puntatore viene assegnato l'indirizzo della
variabile count
 *pint //denota la variabile puntata da pint (deferenziazione)
```

Esempio:

```
int main()
 {
     int i=10;
     int* p=&i; //P contiene l'indirizzo di n
     *p=20;
     return 0;
 }
```

La dimensione del puntatore (sizeof()) e' costante ed e' indipendente dal tipo puntato, nell'architettura a 32 bit e' di 4 byte.

Inizializzazione dei Puntatori

Un puntatore non inizializzato, come qualsiasi variabile, ha come valore iniziale uno aleatorio (valore del registro di memoria all'atto della sua definizione).
E' imporatente dunque inizializzare la variabile. Un puntatore che viene inizializzato a 0 (zero) o a NULL non ha nessun valore, non punta ad nessun oggetto
e quindi non indirizza a nessun dato presente in memoria ed e' detto puntatore nullo.

Esempio:

```
char *p=0; //oppure
 char *p=NULL;
 //si pu_ testare il valore del puntatore
 if (p==0) ...
 //oppure
 if (p==NULL)
```

Puntatore a puntatore

```
int i=100;
int *ptr1=&i;
int **ptr2=&ptr1
```

ptr1 e ptr2 sono due puntatori diversi. Il primo Þ un puntatore ad un intero. Il secondo, invece, Þ un puntatore a puntatore.

```
i= 95;
*ptr1=95;
**ptr2=95;
```

Puntatori costanti

Un puntatore costante ha un valore che non puo' essere cambiato:

```
T* const p= <indirizzo var>;
```

esempio:

```
int *const p1=&x;
```

p1 e' un puntatore constante che punta a x, p1 e' constante e quindi non puo' essere modificato il suo riferimento, ma *p cioe' x e' una variabile a cui possiamo invece modificare il valore.

```
*p1=10;//OK
p1=&y;//ERRORE
```

Puntatori a costanti

```
const T* p= <indirizzo const o stringa>;
```

E' un puntatore ad una variabile constante, const. Il valore del puntatore si puo' modificare mentre il valore puntato no.

```
const int x=25;
cont int *p1=&x;

*p1= &e; //OK, qualunque tentativo di modificare il valore di e
comportera un errore di compilazione
*p1=15; //ERRORE
```

Puntatori constanti a costanti

```
const T* const p= <indirizzo const o stringa>;

const int x=25;
const int * const p1=&x;
```

Qualsiasi tentativo di modificare p1 o *p1 produrra' un errore di compilazione.

I puntatori: Operazioni

Il c++ tratta esplicitamente le espressioni di tipo puntatore.

Sono previste le seguenti operazioni:

- ASSEGNAZIONE: tra puntatori che punatano allo stesso tipo T*:
- Operatori unari unitari (forma prefissa e postfissa)
- INCREMENTO: "++"
- DECREMENTO: "--"
- SOMME E DIFFERENZA: tra un puntatore ed un intero

Puntatori a dati strutturati

Un puntatore puo' puntare a tipo di dati strutturati (array o record).

Si analizzeranno i seguenti casi:

- Puntatore ad array; //pu_ utilmente essere sostituito dalla notazione a [i] tipica degli array
- Puntatore a record;

Puntatore ad array

Il puntatore ad array e' una variabile che punta alla prima locazione dell'array (a[0] in C++);
Considerando pero' che gli elementi sono tutti in posizioni contigue, il puntatore puntera' a tutti gli elemetni dell'array.

Un puntatore ad array e' dunque di per se un puntatore costante al tipo T degli elementi dell'array.

Esempio:

```
const int dim=100;
 float a [dim]; // a Þ un array di dim elementi di tipo float
 float* p ; // p Þ un puntatore a float
 //azzeramento di tutti gli elementi di un array di 100 elementi
 for (p = a, int i=0; i<dim; i++, p++)
 *p= 0;
```

Puntatore a record

Un puntatore a record e' una variabile che punta all'indirizzo di memoria ove e' allocato il record, questo e' motlo utile nella realizzazione di tipi dinamici.

Esempio:

```
// Dichiarazioni di un tipo strutturato Ts
struct Ts { // Ts Þ un tipo strutturato
     D1;
     .....;
     Dn ; } ;
Ts r ; // r Þ variabile di tipo Ts
// Dichiarazione di un tipo puntatore a Ts
typedef Ts* Punt;
// Dichiarazione di variabile di tipo puntatore a Ts
Punt p;

Punt p = &r; // p e' una variabile puntatore inizializzata ad r

p puo' essere ridefinito nel corso del programma

p= &r1; //ora p punta alla variabile r1 di tipo Ts
```

Accesso alla singola componente:

```
(*p).Di;

p->Di;
```

Esempio:

```
// Definizione di tipo struttura:
struct Abbonati { // definisce struttura di nome Abbonati
    Stringa nominativo;
    Stringa indirizzo;
    int numTel ; };
typedef Abbonati* Pabb;
// Definizione di variabili
Abbonati a; // a una variabile di tipo Abbonati
Pabb p= &a; // p, di tipo Pabb, e' inizializzato ad a
// Accesso alla componente nominativo mediante puntatore
p->nominativo;
```

Classi di memorizzazioni delle variabili in C++

Dichiarazioni e definizioni

E' imporatante prima di tutto capire la differenza tra dichiarazione e definizione delle "entita' " di un programma.

Una dichiarazione e' una frase che introduce un nome in un programma e ne enuncia le proprieta'.

- Per constani e variabili:
  -- enuncia il tipo, le classe di immagazzinamento
- Per nomi di tipo:
  -- introduce il nome e la categoria
- Per funzioni
  -- prototipo

Una definizione descrive completamente un' entita'.

- Per le variabili:
  -- dichiara il tipo e definisce allocazione in memoria
- Per nomi di tipo:
  -- definisci tutti i dettagli del tipo
- Per le funzioni
  -- definisce il codice della funzione

Una definizione e' anche una dichiarazione.
Variabili, funzioni vanno dichiarate prima di essere utilizzate.
Per ogni entita' esistono piu' dichiarazioni.

Le definizioni non possono essere ripetute.
Le dichiarazioni devono essere consistenti tra loro e con le definizioni.

esempi di dichiarazioni:

```
     extern int a; // dichiara(ma non defin.) la variable a,
definita in altro file.
     struct Persona; // dichiara un nome di tipo per una struct.
     void f(int); // dichiara(ma non defin.) la funzione f
(prototipazione).
```

esempi di definizione:

```
     const int b = 5; // defin. di costante con nome
     int a; // defin. di variabile
     struct Persona { // definisce il tipo Persona
     char Nome[20];
      } ;
     void f(int x) { // definizione di funzione
      } // corpo della funzione
```

Classi di memorizzazione delle variabili nel C++

Per le variabili in C++, la definizione specifica la classe di memorizzazione, cioe':

- la visibilita' spaziale;
- la durata temporale (ciclo di vita);
- l'allocazione in memoria durante l'esecuzione.

Classi di memorizzazione del C++ (storage class):

- variabili automatiche
- variabili esterne
- variabili automatiche statiche
- variabili esterne statiche
- variabili registrer
- variabili dinamiche

Modello di gestione della memoria per l'esecuzione di un processo

- Area programmi e costanti
  - destinata a contenere le instruzioni, in linguaggio macchina, e le costanti del programma
- Area dati statici
  - destinata a contenere variabili allocate staticamente e quelle esterne (globali)
- Area heap
  - destinata a contenere le variabili dinamiche esplicite, di dimensioni non prevedibili a tempo di compilazione
- Area stack
  - destinata a contenere le variabili automatiche quelle definite all'interno delle funzioni

Variabili automatiche

Sono le variabili locali ad un blocco di istruzioni.
Sono definite in un blocco di istruzioni e sono allocate e deallocate con esso.
La visibilita' lessicale e' locale al blocco.
In memoria sono allocate nell'area stack.

Variabili esterne

Sono le variabili che vengono definite all'esterno di ogni blocco.
La visibilita' lessicale si estende a tutto il programma, me per renderle visibili anche alle procedure contenute in file diversi, devono essere ivi dichiarate con la parola chiave extern.
La loro estensione temporale si estende a tutto il programma.
In memoria vengono allocate nell'area dati statici o globale.

Variabili automatiche statiche

Sono variabili automatiche prefissate con la parole chiave static.
La visibilita' lessicale e' locale al blocco di istruzioni dove sono definite.
La loro estensione temporale si estende a tutto il programma.
In memoria vengono allocate nll'area dati globale.

Variabili esterne statiche

Sono variabili esterne prefissate con la paola chiave static.
La visibilita' lessicale e' locale al file dove sono dichiarate.
La loro estensione temporale si estende a tutto il programma.
In memoria vengono allocate nll'area dati globale.

Variabili register

Sono variabili automatiche prefissate con la parola chiave register.
La visibilita' lessicale e' locale al blocco dove sono dichiarate.
Sono allocate e deallocate con il blocco di istruzioni in cui esse sono contenute.
Se possibile vengono allocate in un registro del processore, altrimenti nell'area stack.
Puo' convenire dichiarare register una variabile automatica usata di frequente, in modo che l'allocazione in un registro della macchina ne aumenti la velocita' di accesso.

Variabili dinamiche

Sono definite durante l'esecuzione del programma mediante puntatori.
Sono accessibili ovunque esiste un riferimento ad esse.
L'allocazione e la deallocazione sono controllate dal programmatore.

In memoria vengono allocate nell'area heap.

Variabili e strutture dinamiche

La gestine delle variabili dinamiche e' a cura del programmatore, attraverso dei meccanismi lunguistici che sono:

- variabili di tipo puntatore;
- operatore new;
- operatore delete;

Operatore NEW

In fase di esecuzione, l'operatore new alloca un'area atta ad ospitare un valore di tipo T e restituisce il puntatore a tale area;
Il tipo T definisce implicitamete l'area di memoria occorrente.

esempio:

- `T* p = new T; //alloca area atta a contenere un elemento e restituisce puntatore`
- `T* p1 =new T[n]; //alloca area atta a contenere n elementi e restituisce puntatore`

Operatore DELETE

Produce la deallocazione dell'area di memoria punatata da una variabile p, cioe' annulla l'allocazione, rendendone di nuovo disponibile lo spazio de memoria nell'area heap.

esempio:

- `delete p; //dealloca area puntata da p`
- `delete [] p1; //dealloca tutto l'array precedentemente allocato`
- `delete [10] p1; //dealloca 10 elementi a partire da quello puntato da p1`

Allocazione dinamica di un array

Con l'allocazione dinamica di un array il puntatore restituito e' quello dell'indirizzo del primo elemento dell'array.

```
float* a= new float[n];
```

Il riferimento agli elementi dell'array viene espresso a mezzo della consueta notazione con indice

```
a[ i ] = 10.0            *(a+i)=10.0
```

Allocazione dinamica di un record

L'allocazione dinamica di un record avviene analogamente, attraverso l'uso dell'operatore new.
Indicanro con R un tipo di record e con r un puntatore ad R, si ha:

```
R* r = new R;
```

Allocazione dinamica di un record contenente un puntatore

```
// Definizione di tipo struttura:
struct Abbonati { // definizione del record Abbonati
                    char* nominativo;
                    Stringa30 indirizzo;
                    int numTel ;} ;
//allocazione dinamica di un record di tipo Abbonati
Abbonati* p = new Abbonati;
//legge nominativo da tastiera
char nome[50];
cin.get(nome,50);
p->nominativo = new char[ strlen(nome ) + 1 ];
strcpy(p->nominativo, nome);
cout << p-> nominativo;
// deallocazione
delete []p->nominativo; //necessario !
delete p;
```

Collegamento fra record generati dinamicamente

```
struct Abbonati;
typedef Abbonati* Pabb;
// Definizione di tipo struttura:
struct Abbonati { // definizione del record Abbonati
                        char * nominativo;
                        Stringa30 indirizzo;
                        int numTel ;
                        Pabb succ;//punta al successivo record } ;

//Definizione di due variabili di tipo puntatore ad Abbonati
Pabb p, p1;
//Allocazione di due variabili di tipo Abbonati
p = new Abbonati;
p1= new Abbonati;
//Collegamento del primo record con il secondo
p->succ=p1;
```

```
//Per generare dinamicamente più record collegati a mezzo di una
catena di puntatori, è sufficiente inizializzare solo il primo
record,con le istruzioni
p = new Abbonati;
q = p; // il valore di p non deve essere alterato, è la testa
della catena ed effettuare quindi in ciclo le operazioni
q ->succ = new Abbonati
q = q->succ;//avanza lungo la struttura
```

Errori comuni con puntatori e variabili dinamiche

Tipici errori quando si usano variabili dinamiche sono:

- Dimenticare di allocare un dato nello heap e usare il puntatore come se lo stesse riferendo
- Dimenticare di restituire la memoria utilizzata allo heap manager (memory leak)
- Tentare di usare dati dinamici dopo che sono stati restituiti allo heap manager
- Invocare delete piu' di una volta sullo stesso dato

Conversioni di tipo (casting) in C++

Trasforma in valore v1 di T1 in un valore v2 di T2.
I tipi fra i quali viene eseguita la conversione, sono in generale, tipi che sono uniti tra loro attraverso un percorso di conversione applicato del compilatore.

Il tipo T2 di destinazione e' noto, in generale, sempre al tempo di compilazione. Si parla di:

Casting statico: se anche T1 e' noto a tempo di compilazione;
Casting dinamico: se T1 e' noto a tempo di esecuzione;

Le operazione di conversione possono essere espresse in forma esplicita o applicate implicitamente dal processore. Sono da preferirsi le conversoni esplicite.

Operatori per la conversione di tipo

- `static_cast<T> (x)`
  - converte x al tipo T, per tipi collegati tra loro.
- `reinterpret_cast<T> (x)`
  - converte x nel tipo T, per tipi NON collegati tra loro
- `dynamic_cast<T*> (p)`
  - converte p al tipo T*, per classi di una gerarchia
- `const_cast<T> (x)`
  - consente di alterare l'oggetto costante x di tipo T

static_cast

Si utilizza per tipo collegato tra loro, la verifica e' eseguita a tempo di compilazione.

esempio:

```
double z=3.14544
int i= static_cast<int>(z);
```

Si utilizza spesso per la conversione di tipi aritmetici, tra puntatori a void e puntatori al tipo T e in generale per tutte quelle operazioni di conversioni lecite
cioe' che esiste almeno un cammino di conversione tra il tipo di x ed il tipo T.

reinterpret_cast

Si utilizza per tipi NON collegati tra loro.

esempio:

```
char* str="abc";
int i= reinterpret_cast<int>(str);
```

Il risultato cui si perviene attraverso questo casting non e' tipicamente portabile da un sistema all'altro.

dynamic_cast

Si utilizza per eseguire cats dinamica tra puntatori.

```
dynamic_cast<T*> (p);
```

- restituisce 0 se p non punta ad un oggetto di tipo T
- restituisce p se p punta ad un oggetto di tipo T, oppure ad una classe accessibile da T

L'operatore in esame e' utilizzato in genere per una conversione di tipo a tempo di esecuzione per il down-casting di puntatori (da puntatore di una classe base a uno di una classe derivata)

const_cast

Si utilizza per convertire un oggetto costante in un oggetto non costante.

```
const_cast<T> (x);
```

- altera la constness dell'oggetto costane x di tipo T

Conversione di tipo per oggetti di tipo classe

La conversione di tipo per variabili di tipo classe e' eseguita nei seguenti casi:
- nell'operatore di assegnazione con oggetti di tipo diverso
- nella chiamata delle funzioni, se parametro effettivo e parametro formale sono scambiati per valore e sono di tipo diverso

Conversione mediante costruttori

Una funzione costruttore della classe X del tipo:
`X(C&);`
Puo' essere riguardata come un operatore di conversione C->X. Esse e' richiamata implicitamente dal compilatore per eseguire l'operazione di conversione.
Il tipo C puo' essere di tipo standard oppure definito dall'utente.

Qualora si voglia inibire l'utilizzo del costruttore per le operazioni implicite di conversione di tipo bisogna premettere l'attributo explicit X(C&).

La conversione di tipo C->X effettuata mediante costruttore non e' possibile:
- se X e' un tipo predefinito
- se X e' una classe gia' esistente che non dispone del costruttore X(C&)

Per tale motivo il C++ introduce gli operatori di conversione:
- operator X();//converte oggetto proprio alla classe X

esempio:

```
Stringa{
           operator int();
};
Stringa::operator int(){
//trasforma oggetto proprio di tipo Stringa in un intero
     return i;
}
Stringa s("123");
int i=s; //richiama s.operator int();
```

Ridefinizione degli operatori

Funzioni operatore

Il C++ consente di definire funzioni operatore applicabili a variabili di tipo classe. Si perviene per tale via alla possibilita' di costruire espressioni che utilizzano gli operatori inizialmente definiti nel linguaggio come applicabili alle variabili ordinarie di tipo aritmetico o altro tipo predefinito.
Seguendo questo approccio possono essere utilizzate espressioni operanti su oggetti:

```
c4= (c1+c2)*c3
```

con c1, c2, c3, c4 variabili di tipo numero complesso.
In realta', un'operazione su tipo predefinito, come ad esempio:

```
i1+i2 //somma di due interi
```

puo' essere riguardata come eseguita a mezzo di richiamo di una funzione che esegue l'operazione richiesta, sviluppano un algoritmo noto al compilatore.
La funzione somma e' inoltre ridefinita nel linguaggio C++ perche' e' anche prevista la somma di due reali:

```
r1+r2 //somma di due reali ( ridefinizione della funziona somma)
```

L'operazione in esame e' realizzata con un algoritmo diverso, anch'esso noto al compilatore.
La funzione somma puo' essere ulteriormente ridefinita, con riferimento ad operandi appartenenti a una classe C:

```
c1+c2 //somma di due oggetti della classe C
```

L'operazione viene in questo caso eseguita secondo un algoritmo definito dal progettista della classe C.
Dal punto di vista tecnico, la redifinizione di un generico operatore OP viene realizzata mediante definizione di una funzione di nome operator op. L'occorrenza dell'operatore op in un'istruzione produce l'attivazione della funzione operator op in precedenza definita.
Cosi' applicato la funzione:

```
c1+c2
```

viene trasformata in:

```
c1.operator op(c2)
```

che produce l'attivazione della funzione operator op() della classe C.
Si tratta della definizioni di piu' funzioni operator op() che costituiscono dunque un insieme di funzioni di egual nome sovraccaricate (operator overloading). Il compilatore, come per il sovraccarico dei nome, scegliera' la funzione che maggiormente si adatta al tipo degli operandi.

Una tassonomia degli operatori

Gli operatori posso essere classificati cosi':

- unari (accettano un solo operando)
  - prefissi(l'operatore precede l'operando)
    - esempio new e delete
  - postfissi(l'operatore segue l'operando)

- binari(accettano due operandi)
  - ad esempio +,-

Alcuni operatori possono essere sia unari che binari
- ad esempio * (dereferenziazione, moltiplicazione)

Alcuni operatori possono essere sia prefissi che postfissi
- ad esempio ++ e --

Le regole da seguire

Per introdurre un operatore bisogna seguire delle regole:

- e' possibile modificare il significato di un operatore esistente, non e' possibile creare nuovi operatori, non e' opportuno cambiare la semantica di operatori applicati a tipi definiti;
- non e' possibile cambiare precedenza, associativita' e "arity"(numermo di operandi);
- non e' possibile usare argomenti di default.

L'overloading di un operatore puo' realizzarsi un uno dei due modi:

- come funzione membro di una classe
  - un operatore binario ha un solo parametro, mentre un operatore unario non ha parametri(l'altro operando e' l'oggetto a cui si applica l'operatore)
- come funzione non membro
  - questa soluzione e' obbligatoria quando la classe non e' disponibile.

Come si comporta il compilatore

Il compilatore quando incontra un'espressione della forma:

```
x op y
```

verifica nell'ordine:

- se nella classe X dell'oggetto x vi e' una funzione membro della forma operator op(Y), dove Y e' la classe dell'oggetto y
- se vi e' una funzione non membro della forma "operator op(X,Y)"

e provvede ad invocare la funzione individuata.

Gli operatori "sovraccaricabili"

Il C++ consente di ridefinire quasi tutti gli operatori previsti del linguaggio:

```
+   -   *   /   %   ^   &   |
```

```
~  !  ,  =  <  >  <= >=
++ -- << >> == != && ||
+= -= *= /= %= ^= &= |=
<<= >>= [] () -> ->* new delete
new[] delete[]
```

Non possono essere sovraccaricati:

```
.  .*  ::  ?:  sizeof
```

Alcuni operatori posso essere sovraccaricati solo come funzioni membro:

```
= [] () ->
```

Per altri gli altri seguiamo la seguente forma:

Operatore	Funzione Raccomandata
Operatre unario	Funzione membro
+= –= /= *= ^= &= |= %= >>= <<=	Funzione membro
Ogni altro operatore binario	Funzione non membro

Ridefinizione di operatori specifici

Ora andremo a vedere piu' nello specifico la ridefinizione dei vari operatori:

- operatore di assegnazione;
- operatori di incremento e decremento;
- operatori di indicizzazione;
- operatore &;
- operatore virgola;
- operatore di chiamata di funzione;
- operatore !;
- operatore di shift << e >>;
- operatore ->;
- operatore [] con dimensioni multiple;
- operatori new e delete;
- operatori di conversione.

Operatore di assegnazione

Prendiamo come esempio una classe C e con c e c1 due oggetti a caso di tale classe, l'operatore di assegnazione realizza operazioni della forma c=c1 ed e' dunque un operatore binario che ha il compito di produrre l'assegnazione delle componenti di c1 alle analoghe componenti di c.
Questo operatore deve essere ridefinito nel caso in cui gli oggetti si presentano con estensione dinamica, andremo ad operare cosi':

- ricopiare la parte base dell'oggetto c1 nella parte base dell'oggetto c, a eccezione del puntatore all'estensione dinamica;
- deallocare l'estensione di c in area heap;
- allocare una nuova estensione di c in area heap;
- ricopiare l'estensione di c1 nell'estensione di c.

Dal punto di vista sintattico il prototipo ha la forma:

```
C& operator=(const C& c1);
```

Il valore restituito e' un riferimento all'oggetto proprio, e in questo modo vengono consentite le assegnazioni multiple, nella forma c2=c=c1.

Operatori di incremento e decremento

L'operatore ++ puo' assumere come e' noto, forma prefissa e post fissa. Per distingure la funzione operator ++() nei due casi, il compilatore consente di definire due distinte funzioni sovrapposte:

++ prefisso: prototipo
```
C operator ++(); // chiamata ++c
```

++ postfisso: prototipo
```
C operator ++(int); //chiamata c++
```

Il compilatore trasforma la notazione c++ in c.operator++(0).
Analogamente posso essere definiti gli operatori -- postfissi e prefissi.

Operatori di indicizzazione

Un impiego tipico dell'operatore di indicizzazione e' quello legato alla classe Vettore, che prevede l'accesso agli elementi di un vettore con controllo dell'indice, allo scopo di verificare che esso non sia esterno all'intervallo [0,dim-1].
Ciascun oggetto a di tipo Vettore e' costituito dalle componenti dim, dimensione del vettore, e v, puntatore all'area contenente il vettore. La funzione operator[] viene definita:

```
T& operator[](int i){
//controlla i rispetto ai limiti del vettore
     return v[i];//restituisce riferimento a elemento i-esimo
}
```

La funzione in esame e' richiamata con notazioni della forma a[i], utilizzabili sia per accesso ad a[i] sia per aggiornamento.
Qualora occore far riferimento agli elementi di a[i] di un vettore costante non e' necessario definire una seconda funzione operator[].
La funzione avra' prototipo cosi':

```
T& operator[](int i)const;
```

In altre parole, nel caso siano definite entrambi le funzioni, quella con oggetto proprio non const verra' richiamata per vettori non const e consentira' operazioni di aggiornamento, quella con oggetto proprio const verra' richiata per vettori const.

Operatore &

L'operatore & e' un operatore predefinito per le variabili di tipo classe: &c fornisce l'indirizzo dell'oggetto c.
Nel caso di oggetto a di tipo Vettore l'operatore & puo' essere ridefinito, allo scopo di ottenere non l'indirizzo dell'oggetto, ma piuttosto l'indirizzo iniziale dell'area v:

```
const T* operator&()const{//restituisce indirizzo iniziale del
vettore
     return v;
}
```

Operatore virgola

L'operatore virgola e', in generale, usato per espressioni della forma e,e1 e produce il calcolo di e ed e1 restituendo e1.
Se e ed e1 con gli oggetti c e c1 della classe C, l'operatore virgola puo' essere ridefinito nella forma:

```
C operator,(C& c1);
```

La funzione in esame puo' essere utilizzata per eseguire qualche elaborazione sull'oggetto proprio c e su c1 per restituire il valore c1.

Operatore di chiamata a funzione

L'operatore di chiamata funzione viene definito per classi che abbiano una sola funzione membro: in questo caso si usa anche la denominazione di funzione oggetto. In altri casi l'operatore viene definito con riferimento a una funzione membro predominante di una classe.
Il prototipo della funzione operator() assume la forma:

```
TR operator()(lista dei parametri);
```

mentre l'istruzione di chiamata di questa funzione ha la forma:

```
c(lista argomenti);
```

Nel caso in esame l'oggetto c assume l'apparenza del nome di una funzione; il numero di argomenti della funzione puo' assumere qualsiasi ed e' definito dal progettista della classe.

Operatore !

L'operatore ! viene talora definito per gestire le condizioni di errore che si producono a seguito dell'applicazione di una funzione membro f(), per esempio la funzione di ingresso/ uscita.
E' il caso d osservare che nel C++ non e definito l'operatore di valutazione if(c), con c di tipo non standard, e infatti esse rientra tra le operazioni predefinite per gli oggetti delle classe e non e' ridefinibile tramite operatore. Il progettista della classe puo' pero' rendere disponibile all'utente anche la notazione if(c), attribuendo opportunamente a essa una semantica complementare a quella della notazione if(!c). Percio' si dovra' definire un operatore di conversione:

```
operator void*();
```

La funzione esamina gli indicatori di stato e restituisce 0 se vi sono errori, altrimenti restituisce il puntatore a c nel caso di non errore.

Operatori di shift << e >>

Gli operatori di shift vengono ridefiniti per consentire all'utente di eseguire sugli oggetti di una classe C le operazioni standard di ingresso uscita tipiche delle variabil predefnite. In particolare vengono rese disponibili all'utente le operazioni:

```
cin>>c; //legge e memorizza le componenti di c
cout<<c; //visualizza le componenti di c
```

Vengono allo scopo definite due funzioni operatore nella forma di funzioni ordinaria con due argomenti, il primo di tipo istream (oppure ostream), il secondo di tipo C, con i seguenti prototipi:

```
istream& operator >> (istream& in, C& c); //legge le componenti di
c
ostream& operator << (ostream& out,const C& c); //visualizza c
```

In particolare le funzioni in esame possono essere dichiarate amiche della classe C.

Operatore ->

L'operatore in esame e' detto selettore di membro e consente, nella sua definizione standard, di puntare a una componente di un oggetto.
Indicata con C una classe contenente una variabile memebro pubblica el, e con c un oggetto di C, si puo' accedere alla componente el di c con le notazioni:

```
...c.el... //accede alla componente el di c
C* p=&c; //p punta a c
```

```
p->el // accede alla componente el di c a mezzo di puntatore
ordinario
```

L'accesso alla componente el di c puo' anche essere ottenuto a mezzo di un adeguata ridefinizione dell'operatore -> . In questo caso si ottiene una maggiore flessibilita' perche' la funzione implicitamente richiamata, puo' eseguire una qualsiasi altra operazione che il progettista ritenga conveniente definire (da qui il termine smart point utilizzato per i puntatori in esame).
La ridefinizione dell'operatore -> richiede l'introduzione di una classe ausiliaria PC la quale assume il ruolo di classe puntatore alla classe C. La classe PC esegue la redifinizione dell'operatore -> , nella forma di funzione membro, ed e' dotata di una variabile membro che punta a C.
Questo schema aiuta a capire l'esposto:

Specifica della classe C

```
class C{//contiene elemento el
public:
     //struttura dati pubblica
     T el;
     ...
};//fine della specifica di C
```

Specifica della classe PC

```
class PC{//classe ausiliaria, redifinisce la funzione ->
private:
     C* pc; //variabile membro di tipo C*
public:
     //redifinizione operatore ->
     C* operator->(){... return pc;}
     PC(C* p):pc(p){}  //costruttore, assegna a pc il punatatore a
C
}; // fine della specifica di PC
```

Uso della classe C

```
void main(){
C c(...);//definisce e inizializza oggetto c
PC p(&c); //richiama costruttore di PC;p,pc puntano a C
...p->el... //richiama  p.operator->()
...
}
```

La notazione p->el consente di accedere alla componente el di c.
Piu' in generale il compilatore esegue una chiamata ricorsiva all'oparatore unario postfisso -> fino a che non si accede a una classe che ha la variabile membro di nome el.

Operatore [] con dimensioni multiple

L'operatore di indirizzamento [] puo' essere applicato ripetutamente a un oggetto, purche' esso sia opportunamente ridefinito.
Ad esempio:

```
o[ i ][ j ]
```

prima viene chiamata la funzione:

```
o.operator[](i)
```

Se per ipotesi essa fornisce un oggetto c, a tale argomento viene quindi applicata la funzione:

```
c.operator[](j)
```

Abbiamo percio'

```
o.operator[](i).operator[](j)
```

Consideriamo per esempio una classe Matrice con elemti di tipo T, potremo definire all'interno di essa una prima funzione operator[] con il seguente prototipo:

```
T* operator[](int i); //restituisce puntatore p alla riga i della
matrice.
```

La funzione in esame e' richiamata nella forma m[i][j] e restituisce un puntatore p alla riga i-esima di m. Una seconda funzione operator[] puo' avere il prototipo:

```
T* operator[](int j); //restituisce puntatore a elemento adi riga
i e colonna j
```

La funzione in esame e' richiamata nella forma p.operator[](j) e restituisce l'elemento della matrice appartenete alla riga puntata da p e alla colonna j.

Operatori new e delete

La redifinizione degli operatori new e delete per una specifica Classe C puo' essere legata alla possibilita' di applicate algoritmi di gestione piu' efficienti di quelli previsti dagli operatori standard, in relazione alle specifiche esigenze di allocazione e deallocazione per gli oggetti della classe C.
In particolare, l'utente potra' richiamare gli operatori ridefiniti, utilizzando le usuali notazioni:

```
C*p=new C; //alloca un oggetto della classe C
C*p=new C[n] //alloca un array di oggetti della classe C
delete (p); //delalloca oggetto puntato da p
```

```
delete[](p); //dealloca array di oggetti puntati da p
```

Agli operatori cosi' ridefiniti si applicano le medesime regole viste per gli operatori standard corrispondenti per quanto attiene alla chiamata implicita di costruttori e distruttori e alla inizializzazione degli oggetti.
Le chiamate in esame vengono tradotte dal compilatore nelle seguenti forme:

```
new C          e' trasformato in   operator new(sizeof C)
new C[n]       e' trasformato in   operator new(n*sizeof C)
delete (p)     e' trasformato in   operator delete(p)
delete[](p)    e' trasformato in   operator[] delete (p)
```

Conseguentemente il progettista della classe C dovra' utilizzare i seguenti prototipi di funzione:

```
void* operator new(size_t n); //alloca un oggeto di n byte
void* operator new[](size_t m); //alloca un oggetto di m byte
void operator delete(void* p); //dealloca oggetto
void operator delete[](void*p); //dealloca array di oggetti
```

Il tipo size_t e' un tipo predefinito, atto ad assumere per valore la dimensione in byte di un oggetto o di un insieme di oggetti.

Operatore di conversione

In una generica classe C possono essere definite una o piu' funzioni per la convenzione di un oggetto c di tipo C in un valore di tipo X predefinito o definito dall'utente.
La funzione di conversione assume necessariamente la forma di funzione membro della classe C e ha il seguente prototipo:

```
operator X(); //converte un valore di tipo C in un valore di tipo
X
```

La funzione di conversione viene richiamata esplicitamente con le notazioni:

```
c.operator X() oppure (X)c oppure static_cast <X> (c)
```

Piu' spesso essa e' richiamata implicitamente.

Il sistema di I/O in C++

Il C++ e' un linguaggio senza alcuna primitiva di I/O, ma dotato di una libreria standard che costituisce un'estensione del linguaggio: la libreria FSTREAM, la quale si presenta sotto forma di gerarchia di classi.
Esse consentono di trattare sia file di tipo testo, con le relative operazioni di formattazione, sia file di tipo binario.

Inoltre la libreria introduce il concetto di stream (flusso) che rappresenta un' astrazione dei dispositivi di ingresso/uscita, ha lo scopo di rendere la scrittura dei programmi indipendente dal particolare dispositivo impiegato, e semplificare il problema di portabilita' dei programmi stessi. Lo stream consente di definire un'interfaccia semplice e uniforma verso l'utente.

La gerarchia di classi I/O

- Il meccanismo di ereditarieta', fa si che tutto cio' che e' possibili realizzare con le classi "padri" e' anche possibili realizzare sulle classi derivate;
- L'header file iostream contiene le classi ios,istream,ostream,iostream:
- L'headre file fstream include iostream, e contiene inoltre le classi ifstream, ofstream, fstream.

Inoltre ci sono altre classi, quali la classe STREAMBUF, che gestisce i buffer di I/O ed esegue le operazioni di accesso alle unita' fisiche del sistema. Ulteriori classi sono le classi ISTRSTREAM, OSTRSTREAM e STRSTREAM per le operazioni di letture scrittura su stringhe, invece che su file.

Le classi istream e ostream

Le classi fondamentali per le operazione di I/O primarie sono:

- istream, per le operazioni di input;
- ostream, per le operazioni di output;

Si tratta di due classi predisposte per le operazioni di I/O sequenziali su file di tipo testo, particolarmente adatte per l'I/O primario (tastiera,video,stampante).
In un programma percio' vengono introdotte specifiche variabili su cui operare:

```
istream a;
ostream b;
```

introducono variabili a di tipo istream e b di tipo ostream.
Le variabili di tipo ostream e istream sono variabili strutturate che contenono tutte le informazioni occorrenti per la gestione dello stream:

- sui buffer associati allo stream;
- sulle dimensioni dello stream;
- sugli errori;
- sulla posizione corrente dei puntatori allo stream;
- sul "fine file"

Gli stream standard cin e cout

Le operazioni di I/O primarie avvengono in particolare a mezzo di due variabili standard:

- CIN, di tipo istream, che rappresenta la tastiera;
- COUT, di tipo ostream, che rappresenta il video;

tali variabili, all'atto di esecuzioni di ciascun programma, risultano predefinite e i file corrispondenti risultano aperti.

Alcuni esempi:
Lettura di un carattere CH (compreso lo spazio)

```
cin.get(ch);
int i=cin,get();
```

Lettura di una stringa

```
char s[100];
cin.get(s,100);//il carattere '\n' resta nello stream, non e'
consumato

char s[100];
cin.getline(s,100);//il carattere '\n' viene prelevato ed
eliminato, senza immetterlo in s
```

Altre funzioni di ostream

```
ostream& put(char c);
ostrem& write(char *s, int n);
```

esempio:

```
char ch='a';
cout.put(ch);

char s[1000];
cout.write(s);
```

Operazioni di ingresso e uscita con formato

Funzioni per il controllo del formato

Il controllo del formato avviene a mezzo di opportune funzioni membro della classe ios:

- int width(int) : definisce l'ampiezza del campo;
- int width() : definisce l'ampiezza del campo;
- int precision(int) : definisce la precisione per i numeri reali;
- int precision() : restituisce la precisione precedentemente definita;

esempio:

```
cout.width(4);
cout<<'a';

cout<<3.14159265; //stampa 3.14159, sei cifre significative
cout.precision(4);
cout<<3.14159265;//stampa 3.145, quattro cifre significative
```

Manipolatori senza argomento

Uso dei manipolatori senza argomento

Nella libreria iostream sono definiti numerosi meccanismi per eseguire operazioni sullo stream, detti manipolatori. Alcuni manipolatori sono privi di argomento:

- flush : svuota il buffer di uscita
- endl : invia a cout il carattere '\n'
- ends : invia a cout il carattere '\0\'
- dec : definisce base decimale per gli interi
- hex : definisce base esadecimale per gli interi
- oct : definisce base ottale per gli interi
- ws : produce eliminazione spazi senza input

Sono utilizzati nella forma generale:

```
cout<<manipolatore;
```

esempio:

```
cout<<9<<flush; //produce lo svuotamento del buffer associato al
video
cout<<hex<<x; //produce stampa a video del valore intero x in base
16
```

Manipolatori con un argomento

Uso dei manipolatori con un argomento

Ulteriori manipolatori con un argomento sono definiti nel file iomanip :

- setbase(int) : posizione base per interi
- setprecision(int) : fissa precisione per reali
- setw(int) : fissa ampiezza del campo
- setiosflag(long) : posiziona a 1 i flag di formato
- resetiosflag(int) : posiziona a 0 i flag di formato

I manipolatori elencati sono utilizzati nella forma generale:

```
cout<<manipolatore(arg);
```

esempio:

```
cout.setprecision(10);//produce la definizione di una precisione
pari a 10 ed e' equivalente a cout.precision(10)
```

Gestione dello stato dello stream

Lo stato dello stream e' rappresentato da un insieme di indicatori dello stado dello stream a valori 0, 1 oppure falso, vero.
I bit in esame sono memorizzati in una variabile membro della classe ios di nome state e tipo intero, in posizione opportuna.
In particolare gli indicatori registrano la condizione verificatasi a seguito dell'esecuzione dell'ultima operazione ingresso/uscita sono definiti cosi':

- eofbit : assume valore vero nel caso di fine file;
- failbit : assume valore vero nel caso di un errore di formato che non ha prodotto la perdita di caratteri(errore recuperabile);
- badbit : assume valore vero nel caso di un errore che ha comportato la perdita di dati (errore irrecuperabile);
- goodbit : assume valore vero nel caso in cui non vi sia un errore (eofbit,failbit,badbit sono posizionati a 0)

Inoltre e' definita la funzione membro:

```
void clear(int i=0);
```

Utilizzata per posizionare a good lo stato dello stream, allo scopo di procedere all'operazione di I/O dopo un errore.

Ed ancora lo stato dello stream puo' essere esaminato a mezzo di una funzione membro

```
int rdstate()const;
```

La quale restituisce un valore intero che codifica i valori di eofbit, failbit, badbit.

esempio:

```
//Testa il valore di goodbit
bool.good()const;
cin.good();
//Testa il valore di eofbit
bool eof() const;
cin.eof();
//Testa il valore di failbit
bool.fail()const;
cin.fail();
//Testa il valore di badbit
bool.bad() const;
cin.bad();
//inoltre
void clear (int i=0);
int rdstate() const;
```

Operazioni di I/O verso le memorie di massa

Le operazione di I/O per file qualsiasi costituiscono una generalizzazione di quelle primarie; occorre aggiungere:

- l' aperture del file oggetto dell'operazione e quindi la sua caratterizzazione;
- lo sviluppo delle operazioni per file binari;
- le operazioni relative ai file di accesso diretto;
- il trattamento di eventuali errori.

La libreria I/O del C++ mette disposizione tre classi:

- ifstream : per l'input;
- ofstream : per l'output;
- fstream : per operazioni di aggiornamento.

Le classi ifstream e ofstream sono derivate di istream e ostream e aggiungono alle operazioni da esse definite anche le operazioni su file ad accesso diretto.

Apertura di un file

Richiede un collegamento tra l'unita' fisica in oggetto dell'operazione e la variabile di tipo

stream definita nel programma.
Due distinti formalismi che coincidono pero' nella semantica:

- Attraverso un costruttore:

```
< tipofile > nomeInterno(nomeEsterno[,< specificatori >]);
```

- Attraverso la funzione open:

```
< tipofile > nomeInterno;
nomeInterno.open(nomeEsterno[,< specificatori >]);
```

Chiusura di un file

E' complementare a quella di apertura e pone tramite al collegamento tra la variabile di tipo file definita nel programma e il file disponibile sulle memorie di massa.

- Attraverso al funzione close:

```
< tipofile > nomeInterno;
nomeInterno.close();
```

Specifica delle operazioni da compiere

Tra gli specificatori dell'operazione di apertura e' incluso il parametro mode, attraverso il quale si definiscono le caratteristiche del file:

//informazioni relative alla struttura interna del file
- ios::binary -> apertura in modalita' binaria (testo per default);

//informazioni relative al tipo di operazioni sul file
- ios::in -> apertura in input;
- ios::out -> apertura di output, con sovrascrittura;
- ios::in | ios::out -> apertura di input e output;
- ios::app -> apertura di output con append;
- ios::ate -> dopo open si sposta a fine file;

//informazioni relative alla creazione
- ios::noreplace -> in scrittura apre un file nuovo;
- ios::nocreate -> in scrittura non crea il file se non esiste;
- ios::trunc -> se il file esiste cancella i contenuti.

Regole di default

- per default l'apertura e' in modalita' testo;
- in input, se il file non esiste l'apertura da errore;
- in output, se il file non esiste viene creato, se esiste viene cancellato e sovrascritto;
- per ifstream e' ios::in;
- per ofstream e' ios::out;
- per fstream non vi e' default, il modo va esplicato.

Apertura per sola scrittua

L'operazione di apertura per sola scrittura e' delicata perche' puo' produrre la cancellazione indesiderata di un file esistete, percio' si puo' operare come segue:

- se il file da aprire e' un file nuovo si usa 'ios::noreplace';
- se il file deve essere scritto solo in append, si usa 'ios::out | ios::app';
- se il file da aprire puo' essere cancellato si usa la modalita' 'ios::out';
- se il file da aprire e' un file gia' esistente che deve essere aggiornato, si usa la modalita' 'ios::in | ios::out'.

esempi:

```
//aprire un file esterno1 con nome interno f1
ifstream f1("esterno1");
//apre in output il file esterno1 con nome file interno f2
ofstream f2 ("esterno2");
//apre in aggiornamento il file esterno3 con nome f3
fstream f3("esterno3", ios::in|ios::out);
//apertura in input con open
ifstream f1;
f1.open("esterno1");
//apertura in output di un file esistente, mod. append
ofstream f3;
f3.open("file.out",ios::out|ios::app);
//apertura in aggiornamento di un file esistente
fstream f4("file",ios::in|ios::out);
//scrittura di un blocco di byte in modalita' binaria
ofstream os;
os.open("esterno",ios::binary);
float v[1000];
int n= 1000*sizeof(float);
//scrive n byte su os prelevandoli da v
os.write(reinterpret_cast< unsigned char*>(v),n);
....
//lettura di un blocco di byte in modalita' binaria
ifstream is;
is.open("esterno",ios::binary);
float v[1000];
int n=1000*sizeof(float);
```

```
//legge n byte da  is e li pone in v
is.read(reinterpret_cast< unsigned char*>(v),n);
```

Operazione per file ad accesso diretto

la gestione dei file con accesso diretto avviene a mezzo delle funzioni di posizionamento e interrogazione sulla posizione. Sui file ad accesso diretto tipicamete di opera in aggiornamento, alternando quindi operazione di lettura con operazioni di scrittura. Per motivi di bufferizzazione tra l'operazione di scrittura e quella di lettura deve essere interposta un'operazione di posizionamento.

- istream& seekg(long p) : Posizionamento per lettura;
- ostream& seekp(long p) : Posizionamento per scrittura;
- long tellg() : Fornisce posizione per lettura;
- long tellp() : Fornisce posizione per scrittura.

esempi:

```
//Operazioni per file ad accesso diretto
fstream f("esterno",ios::binary |ios::in|ios::out);
//definisce struttura del record
struct Record{
.....
     };
Record r;
int n=sizeof(Record);
long pos=2000;
...
long g=f.tellg();
...
long p=f.tellp();
f.seekp(pos*n);
f.write(reinterpret_cast<unsigned char*>(&r),n);
```

Funzioni ausiliarire (alcune gia' descritte)

- ostream& flush() : Svuota lo stream;
- int peek() : Copia il carattere succ. deallo stream senza estrarlo;
- istream& putback(char c) : Ripone il carattere c nello stream;
- istream& ignore (int n=1,int delim=EOF) : Ignora n caratteri;
- void eatwhite() : Estrae spazi consecutivi;
- int gcount() : Fornisce il numero di caratteri letti piu' di recente.

Strutture dati astratte

- Lista
- Pila
- Coda
- Tabella

ADT Lista

Lista ordinata/non ordinata
Politica di accesso agli elementi
Realizzazione
- Array
- Lista dinamica a puntatori

Rappresentazione con array

Struttura dati:

Array statico piu' eventualmente una variabile di tipo intero che mantiene il numero corrente degli elementi nella lista ed inoltre variabili che rappresentano
indici sull'array.
La sequenza degli elementi e' data dalla posizione stessa degli elementi dell'array.

Lista non ordinata:

Inizializzazione (nelem=0)
Inserimento in testa (push)
Inserimento in coda
Ricerca sequenziale
Eliminazione in testa (pop)
Eliminazione di un elemento dato
Analisi dell'elemento in testa (top)
Calcolo dei predicati empty/full (nelem==0;nelem==N)

Lista ordinata:

Le operazioni devono garantire l'ordinamento
La gestione e' onerosa perche' richiede lo spostamento fisico degli elementi in caso di inserimento e di eliminazione

Rappresentazione con lista dinamica a puntatori

Struttura dati

Richiede la dichiarazione di un tipo Record

E' costituita da un puntatore al tipo Record che rappresenta la testa della lista

Gli elementi della lista vengono allocati dinamicamente

Lista non ordinata

Inizializzazione (I=0)
Inserimento in testa (push)
Inserimento in coda
Ricerca sequenziale
Eliminazione in testa (pop)
Eliminazione di un elemento dato
Analisi dell'elemento in testa (top)
Calcolo dei predicati empty/full (I==0/FALSE)

Lista ordinata

Le operazioni devono garantire l'ordinamento
La gestione e' molto vantaggiosa in quanto non ce bisogno dello spostamento fisico degli elementi in caso di inserimento o eliminazione

In generale un ulteriore vantaggio e' dato dalla occupazione della memoria commisurata all'effettiva necessita' e alla possibilita' di rilasciare le aree non piu'
utilizzate in seguito alla cancellazione di un elemento.

ADT Pila

Pila con politica di gestione LIFO: Last in First out

Start
Push
Pop
Top
Empty
Full

La struttura dati prevede un indice-puntatore alla testa della pila.

```
Start(t=0)
Push(p[t]=e;t++)
Pop(t--;e=p[t])
Top(e=p[t-1])
Empty(t==0)
Full(t==N)
```

ADT Coda

Coda con politica di gestione FIFO: First in First out

Start
Append
Pop
Empty
Full

La struttura dati prevede una variabile contenente il numero degli elementi in coda e sia un indice-puntatore alla testa che un indice-puntatore alla coda.

Una variante molto utilizzata e' la coda circolare.

L'inserimento avviene in coda; Il prelievo avviene in testa.

```
Start (neleme=0;t=c=-1)
Append ((c++)%N;nelem++;q[c]=e;)
Pop (e=q[t];(t++)%N;nelem--;)
Empty (nelem==0)
Full (nelem==N)
```

L'incremento degli indici puntatori e' effettuato dal modulo N

Rappresentazione degli ADT Pila e Coda con lista dinamica a puntatori

E' del tutto simile a quanto gia' visto

La struttura dati della Pila prevede un puntatore alla testa.

La struttura dati della Coda prevede un puntatore alla testa ed uno alla coda, cosi' da facilitare alcune operazioni, ad esempio l'inserimento in coda(senza scorrere tutta la struttura)

ADT Tabella

Una tabella e' una struttura dati definita come un insieme finito di coppie:

```
C=(x,y)          x appartiene X, y appartiene Y          dove y=f(x)
```

x e' detta chiave della tabella;

I tipi X e Y possono essere tipi semplici o strutturati.

Rappresentazione

- In forma statica mediante array

- In forma dinamica mediante lista a puntatori

Le coppie possono essere ordinate per chiave oppure non ordinate per chiave

Rappresentazione statica con array di record

Il record rappresenta la coppia chiave-informazione

Un unico array monodimensionale i cui elementi sono i record

Rappresentazione con matrice

Un unico array bidimensionale

La prima colonna contiene le chiavi

La seconda colonna i valori corrispondenti alle chiavi

Limitazione forte: le chiavi e le informazioni devono essere dello stesso tipo

Rappresentazione statica con due array monodimensionali

Due array monodimensionali

Il primo contiene le chiavi

Il secondo contiene i valori corrispondenti alle chiavi

Rappresentazione mediante lista dinamica

Ciascun elemento della lista e' un record contenente tre campi: chiave, informazione, puntatore al prossimo.

#Conclusioni

Con questo ultimo capitolo termina la mia Guida.

Spero che i concetti sopra esposti siano chiari, e perdonatemi ove ci fossero errori.

Per qualsiasi info non esitatemi a contattarmi al mio indirizzo E-mail.

Siete pregati di riportare l'autore della guida se riportate parte dei essa.

Grazie Mille

Saluti Cordiali

#Name: Guida C++: Aspetti Avanzati
#Author: Giuseppe_N3mes1s
#Licence: GNU/GPL
#E-Mail: dottorjekill_mister@tiscali.it
#Development: 30/12/2008