Anno accademico 2001 - 2002

Laurea in Informatica

Appunti di Programmazione 2

Carlo Tomas 1500 MA

Documento scritto con LaTeX 2_ε

Capitolo 1

Richiami di C++ di base

1.1 Le regole d'oro della programmazione

- Leggibilità del codice
- Buon utilizzo delle risorse a disposizione

1.2 Funzioni

Possono essere di due tipi fondamentali:

- senza argomenti: `int main()`
- con argomenti: `int f(int x, int y)`

Inoltre i parametri possono essere passati per:

- per valore, passando semplicemente il nome della variabile
- per indirizzo, passando quindi l'indirizzo della variabile

1.2.1 Funzioni ricorsive

Sono funzioni che richiamano se stesse. Sono formate da due parti:

- *parte di controllo* che permette che la funzione termini
- *chiamata ricorsiva* di se stessa

Di solito sono più compatte della loro versione iterativa, ma costringono a compiere un numero maggiore di calcoli.

1.3 Modello di memoria

stack	memoria allocata *staticamente* dalle funzioni (variabili *automatiche*)
	spazio vuoto che permette a *heap* e *stack* di crescere per ottimizzare lo spazio vuoto
heap	memoria allocata *dinamicamente* da programmatore
unitialized data	
initialized read / write data	variabili *globali* e *statiche*
initialized read only data	
text	codice eseguibile

1.4 Scope delle variabili

globali	sono visibili da tutte le funzioni del programma, ma vanno evitate perché altrimenti possono causare conflitti con le variabili locali.
automatiche	sono visibili solo all'interno della funzione o del blocco in cui sono state definite.
statiche interne	ovvero statiche locali. Il loro valore resta in memoria fino al termine del programma.
statiche esterne	ovvero variabili globali. Sono visibili solo alle funzioni che si trovano nel loro stesso file *al momento della compilazione*.

1.4.1 Principio del NEED TO KNOW

Ognuno deve avere *tutte e sole* le informazioni che servono a svolgere il compito affidato.

1.4.2 Principi di Parna

Il *committente* di una funzione deve dare all' *implementatore* tutte le informazioni necessarie a *realizzare* la funzione, e *nulla di più*.
L'*implementatore* di una funzione deve dare all'*utente* tutte le informazioni necessarie ad *usare* la funzione, e *nulla di più*.

1.5 Tipi di dato

Ci sono vari tipi di dato:
primitivi

```
int a;
float b;
char c;
```

composti

```
struct punto {
  float x;
  float y;
  };
punto origine;
origine.x = 0.0;
origine.y = 0.0;
```

personalizzati

```
typedef float coordinata;
coordinata x, y;
```

1.6 Puntatori

Ci sono due operatori fondamentali per i puntatori:

operatore indirizzo $\&a$ fornisce l'indirizzo della variabile a

operatore dereferenziazione $*p$ interpreta p come un puntatore e fornisce il valore contenuto nella cella di memoria puntata

I puntatori sono molto utili come argomenti delle funzioni. Per passare un parametro x per indirizzo, passo un puntatore che punta ad esso (o più direttamente passo $\&x$, ovvero il suo indirizzo).
Un altro utilizzo dei puntatori viene fatto se si necessita di far ritornare più valori ad una funzione, infatti basta passare gli indirizzi dei parametri, anziché la variabile stessa (o equivalentemente dei puntatori ad esse).[1] Vediamolo con un esempio:

```
int* f(int* x)
{
  (*x)++;
  return x;
}
```

[1] se passo l'indirizzo delle variabili nella funzione attraverso l'operatore `&`, nella funzione devo mettere asterischi dappertutto. Se invece lavoro con le reference, nella funzione devo inserire l'operatore &, ma *solo nei parametri formali della funzione.*

Le operazioni che si compiono sono

1. dereferenziazione
2. incremento
3. restituzione del parametro

Il risultato della funzione è l'indirizzo della variabile x. Se invece consideriamo la seguente funzione:

```
int& g(int& x)
{
  x++;
  return x;
}
```

In questo caso le operazioni che si compiono sono

1. incremento
2. restituzione del parametro

Il risultato di questa funzione è invece il valore della variabile x.
Un altra proprietà dei puntatori è il fatto che un puntatore a struttura non utilizza il classico punto per la dereferenziazione, ma l'operatore "freccia".

```
struct punto {
  float x;
  float y;
  };
punto origine, *p;
origine.x = 0.0;
origine.y = 0.0;
p = &origine;
cout << p->x << ‘‘ ’’ << p->y << endl;
```

1.7 Compilatore, librerie e precompilatore

La creazione di un programma eseguibile avviene secondo questi passi:

1. scrittura del programma come file di testo
2. creazione del file sorgente
3. traduzione del programma da parte del compilatore
4. lista degli errori di sintassi (ma anche no...)
5. eventuale correzione degli errori e ripetizione dal passo 3.
6. creazione del file oggetto (*.o)
7. collegamento tra i vari file oggetto ed eventuali librerie (linker)
8. creazione del file eseguibile su disco
9. caricamento dell'eseguibile in memoria (loader)[2]
10. file eseguibile in memoria
11. esecuzione del programma

1.7.1 Compilazione

Consideriamo ora la prima fase del processo di creazione del file eseguibile. Vedremo alcune righe di comando, con la relativa spiegazione.

```
g++ pippo.c
```

Questo comando mappa il file sorgente `pippo.c`, scritto in un linguaggio ad alto livello, in linguaggio macchina (basso livello), l'unico linguaggio che può essere compreso dal computer[3]. L'esecuzione del programma sarà *sequenziale*, ovvero a partire dalla prima riga in avanti.

```
g++ prog.C -o prog
```

Questa riga di comando crea il file eseguibile, che anziché chiamarsi `a.out`, come di default, viene chiamato `prog`.

```
g++ prog.C -o prog -Ldir -lpippo
```

Questa stringa permette di compilare il programma, attivando i vari collegamenti fra l'eseguibile e le librerie standard, più eventuali altre librerie specificate dagli switch `-L` (seguito dalla path contenente la libreria) e `-l` (seguito dalla parola chiave della libreria desiderata).[4] In questo caso, la parola chiave è `pippo`, quindi si riferirà alla libreria `libpippo.a`, presente alla path `dir`.

[2] anche detto *processo*. Per ogni sorgente sono possibili più processi contemporaneamente
[3] vedi sezione B.3, pag.81
[4] per una lista delle librerie, vedi sezione A.5, pag.78

```
g++ prog.C -c prog.o
```

Di solito il file oggetto viene creato per il processo di link e poi cancellato. Questa riga di comando permette la creazione del file oggetto come unico risultato, ovvero la traduzione del file sorgente in linguaggio macchina[5].
Per proseguire con la compilazione, è sufficiente digitare la seguente riga di comando:

```
ld prog.o -o prog -Ldir -lpippo
```

che chiama il *loader*, che collega il file oggetto con tutte le librerie necessarie e crea l'eseguibile.

```
g++ prog.C -E > prog.baubau
```

Questo comando forza il compilatore a fermarsi subito dopo il primo passo di traduzione del file sorgente, ovvero dopo l'azione del preprocessore. Quindi la correttezza della sintassi non viene controllata. Se inoltre si vuole dare un nome a tale file, basta inserire una parentesi angolare chiusa, seguita dal nuovo nome del file.

1.7.2 Preprocessore

All'interno di un programma, ci sono dei costrutti che non fanno parte del linguaggio `C++`, ma sono costrutti *esterni*: `#define, #ifdef` sono alcuni esempi. Il preprocessore sostituisce questi costrutti con il relativo codice, agendo come un text editor, ma in modo *intelligente*, perché sostituisce soltanto i *token* del linguaggio, ovvero i singoli elementi[6].
I suoi compiti sono:

- sostituire gli `#include` con il codice dei files inclusi. Se i files sono indicati con le parentesi angolari $< \ldots >$, il preprocessore va a cercare il file in `/usr/include` oppure in `/include`, altrimenti nella directory corrente o in quella eventualmente specificata.
- sostituire tutte le occorrenze delle costanti definite dai `#define` presenti nel file sorgente, con il corrispondente valore. È consigliabile però non usare il `#define`, ma il corrispondente `const`, ricordandosi di mettere il punto e virgola in fondo.

Compilando con lo switch `-E`, si blocca la traduzione a questo punto.

[5] L'utilità è spiegata alla sezione B.7, pag.84
[6] Se definisco una costante N, non andrà a sostituire in "Nome"

Compilazione condizionale

La compilazione può essere anche condizionata dall'esterno, attraverso le parole chiave `#ifdef` e `#ifndef`:

```
#define DEBUG
  ...
#ifndef DEBUG
  cout << ''Versione ufficiale'';
#else
  cout << ''Versione di prova'';
#endif
  ...
#ifdef DEBUG
  cout << i << endl;
#endif
```

In questo caso il preprocessore controlla se la costante `DEBUG` è stata definita (non serve dare un valore, basta scrivere la riga di `#define`); a seconda dell'esito del controllo, verranno inserite o meno alcune righe di codice. Questo è un buon metodo per escludere i messaggi di debugging (vedi sezione B.6, pag.83) quando il programma gira in versione ufficiale.
Un'altra scelta possibile è commentare tali pezzi di codice, ma si va incontro a due fondamentali difficoltà:

- se ho commenti annidati, si provocano errori;
- vai tu a commentare centinaia di messaggi di debug!

1.8 L'operatore `sizeof`

L'operatore `sizeof(tipo di dato)` restituisce il numero di byte necessari ad immagazzinare il tipo specificato in memoria.
Allo stesso modo, scrivendo `sizeof(x)` si ottiene il numero di byte necessari per memorizzare la variabile `x`[7].

1.9 Array (ovvero i Vettori!)

Gli array sono collezioni di elementi omogenei. In particolare un array di k elementi di tipo `T` è un blocco contiguo di memoria di grandezza `k*sizeof(T)`. Per indicare gli elementi del vettore viene utilizzato un indice, messo fra parentesi quadre. In tal modo, un array può essere utilizzato esattamente come una variabile, in cui l'indice stabilisce quale posizione considerare all'interno del vettore.

[7] un puntatore ha sempre la stessa grandezza, dipendente dall'hardware, in generale 4 byte, perché il suo contenuto è un indirizzo, quindi un intero.

Attenzione! Gli array hanno dei limiti:

- gli indici spaziano sempre[8] da 0 a $k-1$
- il numero di elementi è **fisso**, deciso a livello di compilazione, ovvero a **compile-time**;
- il numero di elementi non può variare durante l'esecuzione del programma, ovvero a **run-time**;
- non c'è alcun controllo sugli indici durante l'esecuzione. Quindi se considero `int a[10]`, il compilatore **non dà alcun errore** se scrivo `a[50]=4`.

Perciò utilizzando i vettori devo decidere a priori le risorse da mettere a disposizione dell'utente del programma.
I vettori possono essere inizializzati in vari modi:

```
int primi[] = {2, 3, 5, 7, 11, 13, 17};

int a[10];
for(int i = 0; i < 10; i++)
  a[i] = i*2;

int i, n = 100;
int *p;
for(p = a; p < a+n; p++)
  *p = 0;
```

Nel primo modo non serve specificare la dimensione, in quanto il compilatore conta il numero di elementi e alloca la memoria in modo corretto. Il secondo metodo è utile se gli elementi seguono una regola matematica precisa. Nel terzo caso, copio l'indirizzo, dereferenzio, assegno il valore e poi incremento[9].

1.9.1 Vettori e puntatori

Si assomigliano, ma non sono proprio la stessa cosa...
La differenza principale fra queste due scritture sta nel fatto che, mentre un vettore viene allocato staticamente a compile-time e la sua locazione di memoria è in stack, un puntatore viene allocato dinamicamente e quindi la sua locazione di memoria è in heap. `v[0]` è equivalente a `*v`, ma se scrivo `int[100]`, per trovare un'espressione equivalente utilizzando i puntatori devo scrivere:

```
int *v;
v = new int[100];
```

In altre parole devo dichiarare il puntatore, ma poi devo anche allocare dinamicamente la memoria (vedi sezione 1.10, pag.12).
I vettori possono essere visti come zuccheri sintattici per sostituire i puntatori, cioè

[8] per sapere perché proprio da zero, vedi 1.9.1
[9] Ma è un modo molto fumoso, lasciamolo perdere

`v[i]` si può vedere come abbreviazione di `*(v+i)`

Per questo motivo, gli indici dei vettori cominciano da zero e non da uno. Infatti, se cominciassero da uno, avrei che l'equivalenza non varrebbe più, perché il corrispettivo di `v[1]` è `*(v+1)`, che non è il primo elemento, ma il secondo. Per ristabilire l'equivalenza dovrei scrivere `v[i] = *(v+i-1)`[10]

1.9.2 Vettori multidimensionali

I vettori possono essere a più dimensioni. Ad esempio, scrivendo `int v[2][3]`, viene creato un vettore bidimensionale, ovvero una matrice della forma

$$\mathcal{M}_{2\times 3}(\mathbb{N}) = \begin{pmatrix} v_{11} & v_{12} & v_{13} \\ v_{21} & v_{22} & v_{23} \end{pmatrix}$$

Un vettore bidimensionale `v[n][k]` è un vettore di k elementi, i quali sono vettori di n elementi di tipo intero! Quindi

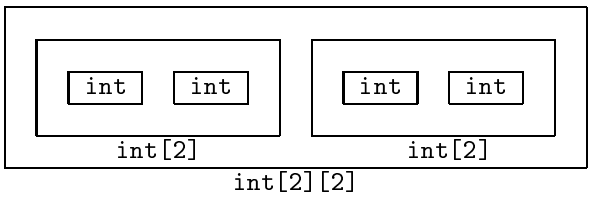
`int v[n][k]` è equivalente a `int (v[n]) [k]` ma anche a `**v`

Anche nel caso dei vettori bidimensionali, come si vede, c'è una versione che utilizza i puntatori, che però è provvista di due asterischi. Dovremo quindi dereferenziare due volte per avere il valore, ma dovremo utilizzare delle parentesi:

`*(*v+i)`

Nel caso `int v[2][2]` avremo quindi la seguente tabella, in cui su ogni riga ci sono varie forme equivalenti per indicare la locazione di memoria posta sulla sinistra.

0012FF70	`v`	`v[0]`	`&v[0][0]`
0012FF74			`&v[0][1]`
0012FF78		`v[1]`	`&v[1][0]`
0012FF7C			`&v[1][1]`



Una caratteristica dei vettori è il fatto che, quando vengono passati alle funzioni, il loro contenuto viene modificato dalla funzione. È quindi come se venissero passati per indirizzo alla funzione[11]. Inoltre quando vengono passati ad una funzione, non è necessario specificare la loro dimensione, proprio perchè viene passato soltanto l'indirizzo del vettore, non il suo contenuto.

[10] che comporta due digitazioni e un'operazione in più...

[11] il che avviene realmente, in quanto vengono trattati come puntatori dal compilatore

1.10 Allocazione dinamica della memoria

Per evitare di dover prevedere le risorse da mettere a disposizione già in fase di compilazione, è possibile allocare la memoria dinamicamente, attraverso il comando

```
new tipo
```

che alloca `sizeof(tipo)` byte nella memoria heap (vedi sezione 1.3, pag. 4) e restituisce un puntatore alla base della memoria, allo stesso modo della funzione del C `malloc()`.
Per liberare un settore di memoria allocato dinamicamente si usa invece la funzione

```
delete(*p)
```

che libera la memoria puntata da p, allo stesso modo della funzione del C `free()`.
Attenzione! Siccome la memoria non è infinita, è bene deallocare lo spazio che non serve più. Perciò per ogni `new` che alloca, è bene che ci sia una `delete()` che dealloca quando la memoria non serve più. Se non si procede in questo modo, si può incorrere nel cosiddetto errore di **memory leak**.
Esempio:

```
void f()
{
  int *v;
  v = new int[10];
}
```

Alla fine dell'esecuzione della funzione `f` il puntatore che era stato creato viene cancellato, ma la memoria non viene deallocata, e quindi viene persa ed insorge il memory leak. Per correggere l'errore ci sono due possibilità:

- deallocare il puntatore tramite il comando `delete(v);`
- ritornare il puntatore allocato attraverso `return v;`

I memory leak sono molto difficili da trovare (infatti i programmi di debugging che servono per trovarli, costano i loro bei soldini...). Un altro errore che si presenta è lo **stack overflow**, dovuto al fatto che la memoria disponibile è stata tutta riempita. Un errore di questo genere avviene ad esempio quando si chiama una funzione ricorsiva che non può mai terminare, come `fattoriale(-1)`.

1.11 Stringhe

Non vengono trattate come primitive dal `C++`, ma come vettori di caratteri (vedi appendice A.3, pag.78). Tuttavia le funzioni di libreria di I/O trattano in modo speciale le regioni di memoria contenenti dei char.
Sono considerati *stringhe* i vettori di caratteri, che sono terminati da un elemento contenente il carattere non stampabile `\0`, indicato anche come `NULL`.
Si avrà quindi che un array di lunghezza N potrà contenere una stringa di lunghezza massima $N-1$, in quanto l'N-esimo carattere serve per il carattere `NULL`.

1.11.1 Operazioni con stringhe

Per scrivere stringhe, si può utilizzare la `cout`, che continua a scrivere caratteri finché non incontra il carattere terminatore `NULL`. Bisogna però fare in modo che esista questo carattere, ovvero bisogna stare attenti a non sovrascrivere un altro valore al suo posto, altrimenti la funzione prosegue fino al prossimo carattere di terminazione, causando un errore.

Per leggere le strighe è invece possibile utilizzare la funzione `cin`, ma tale funzione ha il limite di terminare la lettura non appena trova uno spazio. Perciò se la stringa è composta da più parole, ne viene letta soltanto una, mentre le altre restano nel *buffer* (vedi sezione B.1). Per evitare questo inconveniente, si può utilizzare la funzione `cin.getline(s, dim)`, che legge (`dim - 1`) caratteri dalla stringa `s`[12].
Un'altra scrittura comoda è `cin >> ws`, che sta per *white spaces*, che permette di leggere la stringa tralasciando tutti gli spazi, che vengono "mangiati" dal comando `ws`.

Altre funzioni che operano su stringhe si possono trovare nel file di libreria `string.h`:

- `char *strcpy(a,b)`, che copia la stringa b sulla stringa a;
- `int strcmp(a,b)`, che confronta le due stringhe dal punto di vista lessicografico e che ritorna
  - un numero negativo se a $<$ b;
  - zero, se le due stringhe sono uguali;
  - un numero positivo se a $>$ b;
- `char *strcat(a,b)`, che appende la stringa b alla stringa a, partendo dal terminatore di b;
- `size_t strlen(a)`, che restituisce la lunghezza della stringa a.

Bisogna stare attenti con le stringhe, perché in fin dei conti sono vettori (vedi sezione 1.9, pag.9). Ad esempio il seguente codice è errato:

```
char parola[10];
char *altraparola;
strcpy(altraparola, parola);
```

Il problema sta nel fatto che il puntatore `*altraparola` non è stato inizializzato e quindi non si a cosa punta. Se in particolare nella locazione di memoria a cui punta c'è il valore zero, esso verrà interpretato dal compilatore come un `NULL`-pointer, ovvero un puntatore a `NULL`. Ma poiché sulla locazione di memoria zero non si può scrivere, si avrà un errore, che verrà segnalato come **Bus error**, in quanto il compilatore (che si fida del programmatore[13]) penserà che sia avvenuto un errore nella trasmissione del dato da memoria a CPU, ovvero un problema di BUS[14].

[12] In genere termina anche se trova un `\n`.
[13] ma alle volte non dovrebbe...
[14] uno di quei famosi raggi gamma che mi cambiano i bit...

Per correggere l'errore basta inserire `char altraparola[10]` al posto del puntatore nel codice, oppure allocare dinamicamente la memoria in base alla lunghezza della stringa `parola`. (il secondo metodo è un po' più lento rispetto al primo.)
Attenzione quindi: il comando `char *frase` definisce un puntatore, ma *non alloca memoria!!!*.
L'ultima cosa da vedere sulle stringhe è un metodo per copiare zone di memoria, char per char, attraverso i puntatori:

```
int n;
char *p, *q;
...
while(n--)
  *p++ = *q++;
```

In questo modo copio `n` char dalla locazione di memoria puntata da `p` a quella puntata da `q`.
Nota: `*p++` sta per `*(p++)`, cioè prima incremento il puntatore e poi dereferenzio. Se avessi avuto `(*p)++`, avrei incrementato il valore della variabile puntata da `p`.

Capitolo 2

La pila

2.1 Caratteristiche del codice perfetto

Un buon codice, ovvero un codice fatto per durare nel tempo, deve essere

- *riutilizzabile*, ovvero deve essere possibile usarlo in più contesti
- *mantenibile*, cioè facile da modificare e da comprendere
- *generale*, cioè indipendente dal programma in cui viene inserito

In questo corso, impareremo *come* creare codice che rispetti tali caratteristiche.
Ed ora partiamo!

2.2 La Pila

Una pila è una struttura per immagazzinare dati, in cui i dati vengono estratti in ordine inverso rispetto all'inserimento[1].
Consideriamo il seguente codice che definisce un pila di 10 interi:

```
struct Pila
{
  int contenuto[10];
  int marker;         // l'indice del vettore
};
```

Questo pezzo di codice ha due grossi vincoli:

- se voglio definire una pila di floating point, non posso riutilizzare questo codice, ma devo riscriverlo;
- se devo immagazzinare 15 interi, non posso utilizzare questa pila, in quanto il vettore è allocato staticamente e quindi non è possibile modificarlo.

[1] È un *LIFO-stack*, ovvero Last In, First Out. Invece una *coda* è un *FIFO-stack*, ovvero First In, First Out (detto anche *queue*).

Il primo problema è il più difficile da risolvere e lo vedremo più avanti[2]. Cerchiamo però di risolvere il secondo. La soluzione più comoda è allocare dinamicamente il vettore contenuto nella pila, in modo da poterlo modificare qualora ce ne sia bisogno.
Alla luce delle caratteristiche che deve avere il nostro codice, modificheremo quindi il codice in questo modo:

```
struct Pila
{
  int size;
  int defaultGrowthSize;
  int marker;
  int * contenuto;
};
```

dove assumeremo che:

- `size` indica la dimensione iniziale del vettore
- `defaultGrowthSize` indica l'incremento della dimensione del vettore, una volta che questo si è riempito
- `*contenuto` è il puntatore alla locazione di memoria da cui parte il vettore

Adesso va già meglio, ma la pila non è ancora utilizzabile, in quanto non è stata allocata alcuna memoria per contenerla.
Costruiamo allora una serie di funzioni, utili alla creazione e alla modifica della nostra pila.

2.3 Creazione di una pila

La prima funzione che scriviamo è quella che permette di creare la pila, allocando la memoria necessaria.

```
Pila *create(int initialsize)
{
  // crea una pila
#ifdef DEBUG
  cout << ``entro in create'' << endl;
#endif
  Pila *s = new Pila;
  s->size = initialsize;
  s->defaultGrowthSize = initialsize;
  s->marker = 0;
  s->contenuto = new int[initialsize];
  return s;
}
```

[2] vedi sez. 4, pag. 55

Commentiamo questa funzione.
Come prime righe di codice, inseriamo gli output per la fase di debug, attraverso il costrutto `#ifdef`.
Dopodiché creiamo una nuova pila, allocando la memoria necessaria[3] e creando i collegamenti fra il puntatore a pila e i membri della stessa. Infine inizializziamo il vettore come formato da `initialsize` interi. Lo facciamo in modo dinamico, come già accennato, in modo che sia possibile in seguito aumentare la sua dimensione.

Notiamo che sarebbe stato possibile creare una pila senza utilizzare il puntatore, scrivendo semplicemente `Pila s; s.size = initialsize`... ma avremmo creato la pila in stack, anziché in heap, come una variabile automatica. In questo modo, non appena ritornavamo il puntatore alla pila (in questo caso si tratta semplicemente del suo indirizzo), la funzione di creazione sarebbe stata terminata e la variabile cancellata, impedendo il suo utilizzo. Avremmo avuto fra le mani una locazione di memoria inutilizzabile.

2.4 Distruzione di una pila

La prossima funzione distrugge la pila, deallocando la memoria utilizzata, in modo da non incorrere nel memory leak e poter riutilizzare lo spazio.

```
void destroy(Pila *s)
{
  // distrugge la pila
#ifdef DEBUG
  cout << ``entro in destroy'' << endl;
#endif
  delete [](s->contenuto);
  delete(s);
}
```

La penultima riga dealloca la memoria puntata da `contenuto`. Le parentesi quadre stanno ad indicare che si tratta di un vettore. L'ultima riga invece dealloca la memoria a cui si riferisce il puntatore a pila `s`.
Notiamo che queste due righe *devono essere eseguite in questo ordine*. Infatti, una volta eseguito `delete(s)`, si perde l'indirizzo del vettore contenuto, che non è più deallocabile e si incappa nel memory leak.

2.5 Crescita della pila

Quando la pila è piena, cioè quando il vettore è stato riempito, è necessario aumentare la sua dimensione, se si vogliono immagazzinare altri dati, senza uscire dal range del vettore. È necessario quindi allocare nuova memoria.

Ma non è detto che le locazioni di memoria contigue al vettore siano libere e nemmeno che siano in numero sufficiente. Ad esempio potrebbe accadere che la memoria contigua al vettore sia occupata da altre variabili definite all'interno del mio codice.

[3] pari a 4 byte per le variabili della pila, più `4*initialsize` byte per il vettore `contenuto`

Una soluzione potrebbe essere lo spostare questi dati in una locazione libera, ma dovrei aggiornare tutti gli eventuali puntatori che puntano ad essi[4]. Questo non risulta quindi un metodo applicabile, in quanto potrei aver copiato il vecchio indirizzo in altre variabili e dovrei aggiornare anche quelle.

La soluzione più comoda risulta quindi la seguente:

- allocare un nuovo vettore in uno spazio di memoria libero
- copiare il contenuto del vecchio vettore nel nuovo spazio
- deallocare il vecchio vettore[5]
- aggiornare l'indirizzo memorizzato in `s->contenuto` con quello della nuova locazione di memoria.

Il risultato è il seguente:

```
void cresci(Pila *s, int incremento)
{
  // aumenta la dimensione della pila
#ifdef DEBUG
  cout << ``entro in cresci'' << endl;
#endif
  s->size+=incremento;          // nuova dimensione
  int *temp = new int[s->size]; // nuovo spazio allocato
  for(int k = 0; k <= s->marker; k++)
    temp[k] = s->contenuto[k];
  delete [](s->contenuto);      // dealloco la memoria
  s->contenuto = temp;          // aggiorno il puntatore
}
```

2.6 Inserimento dati nella pila

Questa funzione inserisce un nuovo valore nella prima cella del vettore che risulta libera. In realtà vado a scrivere sulla cella puntata da `marker`, cioè dall'indice del vettore. Se però il vettore risulta pieno, viene chiamata la funzione cresci, allocando nuova memoria e permettendo così la memorizzazione del dato.

```
void inserisci(Pila *s, int k)
{
  // inserisce un nuovo valore k nella pila
#ifdef DEBUG
  cout << ``entro in inserisci'' << endl;
#endif
  if(s->size == s->marker) // il vettore è pieno
    cresci(s, s->defaultGrowthSize);
  s->contenuto[s->marker] = k;
  s->marker++;
}
```

[4] Questo procedimento si chiama *rilocazione dei dati*

[5] devo farlo *prima* di sovrascrivere l'indirizzo, altrimenti perdo il vecchio dato

2.7 Estrazione di un valore dalla pila

Questa funzione estrae l'ultimo elemento inserito nella pila, se la pila non risulta vuota; in caso contrario, l'esecuzione termina con un errore, attraverso la funzione `assert`.

```
int estrai(Pila *s)
{
  //estrae l'ultimo valore inserito
#ifdef DEBUG
  cout << ``entro in estrai'' << endl;
#endif
  assert(s->marker>0);
  return s->contenuto[--(s->marker)];
}
```

Notiamo che l'ultima istruzione ritorna l'ultimo valore inserito, dopodiché decrementa l'indice[6].

2.7.1 la funzione `assert` e i prerequisiti

La funzione `assert` permette di controllare i dati prima di utilizzarli. In pratica sostituisce gli `if` di controllo, rendendo più leggibile e meno confuso il codice. Come argomento accetta una condizione di controllo. Se la condizione risulta vera, l'esecuzione del codice prosegue senza problemi; se invece la condizione risulta falsa, l'esecuzione viene terminata e appare un output di errore del tipo

```
prog: prog.C:7: int main(): Assertion `k>0' failed.
```

Questa funzione può essere utilizzata in due modi:

- prima di una chiamata di funzione, all'interno della funzione chiamante, per controllare che i parametri che verranno passati siano accettabili;
- all'interno della funzione chiamata, per controllare che i dati che sono stati passati siano accettabili.

La scelta di una o dell'altra opzione, tira in ballo il cosiddetto *problema dei prerequisiti*. Definiamo

prerequisito condizione che deve essere vera prima che la funzione venga chiamata[7]

postrequisito condizione che deve essere vera quando la funzione termina

invariante condizione che mantiene il suo valore di verità prima, durante e dopo l'esecuzione della funzione

[6] il valore restituito non viene cancellato!!

[7] ad esempio la funzione che calcola il fattoriale ha come prerequisito che il parametro sia non negativo

La programmazione basata su questi concetti è detta *programmazione a contratto*. Essa ha il vantaggio che le funzioni non avranno problemi per quanto riguarda i parametri, perché soddisfano i prerequisiti. È quindi una buona regola esplicitare i prerequisiti nella documentazione del codice[8], in modo da non dover utilizzare le asserzioni.

2.8 Stampa dello stato della pila

Questa funzione stampa semplicemente tutte le caratteristiche attuali della pila. Non serve quindi il messaggio di debug, in quanto la chiamata presuppone già un output.

```
void stampaStato(Pila *s)
{
  // stampa lo stato della pila
  cout << ``======================'' << endl;
  cout << ``size = '' << s->size << endl;
  cout << ``defautlGrowthSize = ''
       << s->defautlGrowthSize << endl;
  cout << ``marker = '' << s->marker << endl;
  for(int k = 0; k < s->marker; k++)
    cout << ``[``<<(s->contenuto[k]) <<''];
  cout << endl;
  cout << ``======================'' << endl;
}
```

2.9 Copia di una pila

La funzione produce una copia della pila passata come parametro, ritornando un puntatore alla locazione di memoria della nuova pila.
È proprio necessario creare una funzione che copia la pila?
Non basterebbe scrivere queste due righe?

```
Pila *s = crea(5);
Pila *w = s;
```

Il problema sta nel fatto che con queste due righe *non creo* una nuova pila che sia la copia della vecchia, ma *copio semplicemente l'indirizzo* della vecchia pila in un nuovo puntatore. Il risultato è che se modifico la vecchia pila, modifico anche la nuova, perché entrambi i puntatori si riferiscono *alla stessa pila*.

[8] ... nella speranza che qualcuno la legga

La funzione giusta è quindi la seguente:

```
Pila *copia(Pila *from)
{
  // copia la pila
#ifdef DEBUG
  cout << ``entro in copia'' << endl;
#endif
  Pila *to = crea(from->size);
 /* crea una pila della stessa dimensione della vecchia */
  to->defaultGrowthSize = from->defaultGrowthSize;
  for(int k = 0; k < from->marker; k++)
    to->contenuto[k] = from->contenuto[k];
  to->marker = from->marker;
  return to;
}
```

Notiamo che l'assegnazione del valore di `defaultGrowthSize` serve, perché a priori non so se la dimensione della vecchia pila è stata modificata in precedenza. Inoltre il vettore `contenuto` esiste, perché è stato creato dalla funzione `crea`.

2.10 Problemi di questa versione

```
#ifndef ASSERT
  #define ASSERT
  #include <assert.h>
#endif
#ifndef IOSTREAM
  #define IOSTREAM
  #include <iostream.h>
#endif
// *********** STRUTTURE DATI *********
struct Pila
{
  int * contenuto;
  int size;
  int marker;
  int defaultGrowthSize;
};
// *********** FUNZIONI ***************
Pila * crea(int initialSize) ;
void distruggi(Pila * s) ;
void cresci(Pila *s, int increment);
void aggiungi(Pila *s, int k);
int estrai(Pila *s);
void stampaStato(Pila *s);
Pila *copia(Pila *from);
```

A questo punto le funzioni sono state create tutte. Basterà quindi costruire un file header `Pila.h` in cui inserire la definizione della struttura e i prototipi delle funzioni, mentre il corpo di ogni funzione sarà incluso in un file di libreria.

Questa versione però è insoddisfacente, perché è poco leggibile. Se per esempio avessi una decina di strutture, con dieci funzioni a testa, il file header risulta veramente incomprensibile.
Se inoltre volessi implementare una struttura `coda` sullo stesso modello, dovrei cambiare tutti i nomi delle funzioni, altrimenti avrei funzioni diverse con lo stesso nome.
Per questo motivo in `C++` è stato introdotto il *polimorfismo* per le funzioni, ovvero la possibilità di avere più funzioni con lo stesso nome, ma con parametri di tipo diverso.
Si definisce *firma* di una funzione la coppia

(nome della funzione, tipi dei parametri formali)

I primi compilatori di `C++` non facevano altro che sostituire i nomi delle funzioni omonime con una stringa formata dalla firma delle stesse, compilando poi come se il programma fosse stato scritto in `C`. In tal modo, se per esempio avessimo una funzione `cresci` sia per la pila che per la coda, avremmo che i nomi delle funzioni diventerebbero:

`cresci_pila_int` e `cresci_coda_int`

Se invece volessi chiamare la funzione `cresci`, passando come parametro, anziché un intero, un dato di tipo `short`? Potrei farlo, perché il compilatore eseguirebbe un cast, convertendo il dato in intero prima di chiamare la funzione, perché non ha il prototipo di tipo short.

E se invece volessi definire una funzione

```
int cresci(coda *s, int k);
```

ovvero una funzione con lo stesso nome e gli stessi parametri di quella già esistente, ma con tipo di dato ritornato diverso?
Non potrei farlo, perché in `C` e in `C++` è possibile ignorare il valore di ritorno di una funzione, guardando soltanto il *side effect*. Non sarebbe quindi possibile distinguere le due funzioni, anche perché il tipo di dato ritornato *non entra nella firma della funzione*.

Inoltre non in questo modo non sarebbe molto chiaro il legame di appartenenza delle funzioni alla pila.
Ricordiamo che nelle definizioni dei tipi di dato abbiamo:

- informazioni sulla memoria utilizzata;
- informazioni sulla rappresentazione del tipo[9];
- definizione delle operazioni disponibili.

Finora, definendo la struttura, abbiamo definito soltanto il tipo del dato e la sua rappresentazione, ma non abbiamo definito alcuna operazione.
Definiamo quindi l'*interfaccia*, ovvero l'insieme dei modi in cui posso interagire

[9] ad esempio il tipo `char` è rappresentato come carattere, il tipo `float` attraverso mantissa ed esponente...

con la struttura, inserendo nella struttura i nomi delle operazioni permesse, con la loro firma, ma *senza* specificare il codice.
In `C++` posso fare una cosa simile, attraverso l'uso degli header file, ma non ho un costrutto apposito, come invece accade in `Java`.

2.11 Come lego le funzioni alla pila?

Una prima soluzione è portare le funzioni all'interno della definizione della struttura. Avrò quindi due parti:

- definizione dei dati;
- definizione delle funzioni.

Poiché la funzione si riferisce alla struttura stessa, il parametro formale che le passo non serve più e quindi lo tolgo. Dovrò quindi modificare anche il corpo della funzione, sostituendo ogni occorrenza del parametro passato con il *parametro formale implicito* `this`, che risulta quindi essere un puntatore alla struttura.
Però scrivere il corpo della funzione all'interno della struttura comporta alcuni svantaggi:

- il codice sorgente è visibile all'utente della struttura, il che va contro il principio di Parna;
- l'interfaccia risulta meno comprensibile, perché non introduco soltanto la firma, ma anche il codice stesso;
- il file header contiene il codice delle funzioni, cosa che non salvaguarda il mio lavoro.

Devo cambiare sistema.
Scriverò quindi *soltanto i prototipi* delle funzioni nella struttura, mentre il corpo delle funzioni verrà inserito in un file di libreria. Il problema sta nel mostrare che la funzione appartiene alla struttura `Pila`. La soluzione si ha scrivendo

```
int Pila::estrai()
{
  ...    // corpo della funzione
}
```

In tal modo l'utente capisce subito che si tratta di una funzione della pila.
Si possono fare anche altre modifiche, per rendere il codice meno pesante. Ricordiamo ora la regola generale sullo scope delle variabili all'interno di un codice:

Una variabile è visibile dalla sua definizione fino alla parentesi graffa che racchiude il blocco in cui è definita.

In particolare, una variabile globale è visibile in tutto il programma, in quanto non esiste una parentesi graffa che la include.

Quindi le variabili della struttura[10] sono visibili all'interno delle funzioni che sono state definite nella struttura. Non serve quindi usare il puntatore implicito, nemmeno se la struttura contiene soltanto il prototipo della funzione.
Ma allora a cosa serve il puntatore `this`?
Serve in alcuni casi particolari, ad esempio se avessi una definizione di questo tipo:

```
struct P
{
  ...
};

P* f(P *q)
{
  return q;
}
```

Notiamo che la variabile passata come parametro alla funzione è la stessa che viene restituita. In questo caso non è possibile fare a meno del puntatore implicito, in quanto non sapremmo come far ritornare il valore. Quindi diventerà

```
struct P
{
  ...
  P* f()
  {
    return this;
  }
}
```

È preferibile tenere sempre il parametro implicito, in quanto si aumenta la leggibilità del codice, soprattutto in presenza di programmi molto lunghi.

2.12 La funzione costruttore

La funzione `crea` è diversa dalle altre, in quanto non le viene passato alcun parametro formale di tipo pila[11]. Non sarà quindi possibile applicare il ragionamento visto finora. Inoltre questa funzione mi dà alcuni problemi: se per esempio l'utente non legge la documentazione[12], può voler definire una pila, semplicemente utilizzando il comando `new`, che alloca la memoria per la struttura, ma non per il vettore `contenuto`. Questo crea molti problemi , in quanto:

- `size` non viene inizializzata;
- `marker` non viene inizializzato;
- `contenuto` non viene allocato.

[10] dette anche *campi o istanze* della struttura
[11] la pila viene creata all'interno della funzione...
[12] RTFM, Read That Fucked Manual!!

Il risultato è che quando inserisco dati nella pila, vado a scrivere in locazioni di memoria non predecibili, dipendenti dai valori che ci sono nelle variabili non inizializzate.
Un altro problema: se definisco una variabile pila come automatica (o globale) semplicemente con `Pila s;` questa verrà allocata in stack, ma anche stavolta non verranno inizializzati i campi. Quindi, appena vado ad inserire dati nella pila, ricado nel problema precedente.

Devo rassegnarmi: così come è scritta, la funzione `crea` non va bene. Dovrei scrivere due funzioni: una che crea la pila, l'altra che la inizializza soltanto. Ma se l'utente non legge la documentazione... il mio codice resta comunque fragile. Il `C++` mi viene incontro e mi permette di scrivere una funzione che ha *lo stesso nome* della struttura, che ha il compito di inizializzare (e basta!!!) i campi della struttura stessa. Il suo nome è *funzione costruttore*.

```
Pila::Pila(int initialSize)
{
  size = initialSize;
  defaultGrowthSize = initialSize;
  marker = 0;
  contenuto = new int[initialSize];
}
```

In questo modo, appena l'utente definirà una variabile di tipo Pila, il compilatore chiamerà *implicitamente* (ovvero senza che nessuno glielo chieda) la funzione costruttore, che inizializzerà i campi in modo corretto.
Notiamo però che il costruttore richiede come parametro la dimensione iniziale. Dovremo quindi modificare la sintassi: avremo

```
Pila *s = new int Pila(10); // definizione dinamica
Pila t(10);                 // definizione statica
```

Ricordiamo che ora le funzioni sono state dichiarate internamente alla struttura. La chiamata di funzione sarà quindi diversa da prima. Chiamerò la funzione come se dovessi riferirmi ad una variabile.
Ad esempio la funzione `estrai` verrà chiamata scrivendo

```
// definizione dinamica
  Pila *s = new Pila(3);
  s->estrai();
// definizione statica
  Pila t(10);
  s.estrai();
```

Notiamo infine che la funzione costruttore non restituisce alcun valore. Ma **attenzione:** non deve essere dichiarata void!! Quando scrivo il prototipo nella struttura, non devo scrivere niente davanti al nome, quando scrivo il corpo della funzione, scriverò semplicemente `Pila::Pila(int initialSize)`.

2.13 La funzione distruttore

Ricordo che scrivendo `Pila s(10);` cioè defininendo la pila staticamente, essa verrà memorizzata

- in stack, se è una variabile locale o automatica;
- nello spazio per le variabili globali, se è una variabile globale o del main.

Quando la funzione termina, se la pila è locale, se non chiamo la funzione `distruggi`, ho un problema di memory leak, in quanto la variabile viene cancellata, ma lo spazio del vettore non viene deallocato.
Il linguaggio ci viene ancora una volta in soccorso, dandoci la possibilità di definire un' altra funzione implicita: la funzione *distruttore*. Questa funzione viene chiamata dal compilatore subito prima di terminare una funzione, in modo che lo spazio venga deallocato in modo corretto, evitando memory leak.

`Pila::~Pila()`

```
{
// distrugge la pila
#ifdef DEBUG
  cout << ``entro nel distruttore'' << endl;
#endif
  delete [] contenuto;
// NON VA delete this;
}
```

In questo modo, quando la funzione termina, la memoria viene deallocata, ma il puntatore implicito non viene cancellato. Scrivendo quindi

```
f()
{
 Pila *s = new Pila(10);  // chiama il costruttore
 ...
 delete s;
}
```

al momento della `delete`, varrà chiamato il distruttore, che deallocherà la memoria, dopodiché il puntatore verrà cancellato dalla funzione stessa. In questo modo si evita il memory leak e si mantiene la regola che per ogni `new` ci deve sempre essere una `delete`.

È buona norma di programmazione creare sempre le funzioni costruttore e distruttore (eventualmente commentandole se non servono), in modo da rendere più leggibile il codice.

Il distruttore non serve soltanto per le strutture, ma anche per qualsiasi altra risorsa che deve essere deallocata. Ad esempio si può chiamare un distruttore per deallocare memoria utilezzata da file temporanei, da variabili allocate dinamicamente...
La cosa importante è

Ricordarsi che esistono chiamate di funzione in modo implicito!

D'ora in poi chiameremo variabili e funzioni di una struttra in modo diverso:

- le variabili campo verranno dette *variabili di istanza* oppure *dati membro*
- le funzioni saranno chiamate *metodi* oppure *funzioni membro*[13]

2.14 Overloading e Tradeoff Space-Time

Abbiamo già parlato di polimorfismo quando abbiamo affrontato il problema di avere più funzioni con lo stesso nome. La soluzione che avevamo adottato era quella di dotare ogni funzione di una firma, formata dal suo nome e dal tipo dei parametri formali, in modo da poter distinguere funzioni con lo stesso nome ma con parametri di tipo diverso.
Questo sistema è detto *overloading dei metodi* e vale anche per il metodo costruttore, ma *non per il distruttore!!* Sarà possibile perciò avere più metodi per lo stesso costruttore, ma un unico distruttore.
Ad esempio, se l'utente non specifica la dimensione della pila, il metodo costruttore che è stato implementato non può inizializzarla. Dovremo quindi creare due metodi diversi, uno con una dimensione di default ed uno con la dimensione passata come parametro. La nuova versione del costruttore sarà quindi

```
Pila::Pila()
{
// costruttore con dimensione di default
  int initialSize = 10;
  this->size = initialSize;
  this->defaultGrowthSize = initialSize;
  this->marker = 0;
  this->contenuto = new int[initialSize];
}
```

Avremo quindi tre modi diversi (sia statici che dinamici) per definire una pila:

- `Pila *s = new Pila(10);`
- `Pila *s = new Pila();`
- `Pila *s = new Pila;`

Gli ultimi due sono equivalenti. Se ho definito soltanto il costruttore con parametro, solo il primo modo è utilizzabile. Se inoltre non viene definito un metodo costruttore, l'unica scrittura esatta è la terza.
Sorge però un problema: implementando due costruttori si creano due punti del programma in cui è inserito lo stesso pezzo di codice. La parte di inizializzazione della struttura infatti è uguale nei due costruttori e questo non è bene, in quanto il codice risulta poco mantenibile[14].
La soluzione è creare una funzione che contenga solo il pezzo di codice replicato

[13] le prime nomenclature sono usate in generale nella programmazione ad oggetti, le seconde soprattutto in `C++`

[14] se devo fare modifiche a un pezzo di codice, devo ripetere queste modifiche a tutte le repliche del codice, ma potrebbero essere tante...

e chiamarla al momento opportuno. Questa funzione non sarà chiamata dall'utente e viene detta *funzione di servizio*. Questa soluzione però mi porta lo svantaggio che le variabili vengono allocate in stack, così pago l'ottimizzazione del codice in termini di velocità. Questo è un problema non facile da risolvere, perché mantenibilità e velocità dei programmi non vanno d'accordo.

In questo caso abbiamo una soluzione: inserire davanti al prototipo della funzione che deve essere replicata il comando `inline`.
Questo comando è una specie di macro, che dice al compilatore di espandere la chiamata della funzione con corpo completo della stessa[15].
Nota: le funzioni implementate all'interno della struttura sono tutte trattate come `inline`.
Il processo di sostituzione non è però di semplice realizzazione, in quanto è necessario sostituire tutti i parametri formali con i rispettivi valori passati come argomento alla funzione. Anche in questo caso si ha un conflitto fra velocità di esecuzione e ottimizzazione del codice.
In generale questo è un problema di *Tradeoff space-time*, ovvero compromesso fra spazio occupato dal programma e tempo impiegato per l'esecuzione.
Posso rendere il codice più veloce, pagando in termini di spazio su disco. Ad esempio se ho bisogno di calcolare molte volte una funzione che ha come parametro un intero e restituisce un intero molto grande, posso evitare di fare ogni volta tutti i conti (e quindi risparimare tempo) calcolando una volta per tutte la funzione per ogni intero possible (a 16 bit sono 65535), inserendo il risultato in un vettore. Questo riduce di molto il tempo di esecuzione, perché non è più necessario eseguire i calcoli ma basta andare a leggere il risultato sul vettore. D'altra parte occupo più spazio in memoria, in quanto prima il vettore non c'era. Nel caso della macro `inline`, risparmio tempo durante l'esecuzione, ma aumento lo spazio occupato, in quanto il codice viene espanso e l'eseguibile risulta più grande.

Possiamo però risolvere il problema della duplicazione del codice e anche della definizione di due costruttori anche in un altro modo: definiamo un unico costruttore che unisce entrambe le caratteristiche:

```
Pila::Pila(int initialSize = 10)
{
  ...
}
```

In questo modo se l'utente specifica una dimensione per la pila, quella sarà passata al costruttore, altrimenti verrà utilizzato il *valore di default*.
Inoltre la sintassi permette che i parametri che hanno un valore di default possano essere omessi nella chiamata di funzone.
Attenzione: ci sono però dei vincoli in merito:

- le variabili a cui è stato assegnato un valore di default devono per forza essere *gli ultimi parametri formali* della funzione. Non sarà possibile quindi una definizione del tipo `int f(int x = 3, int y)`
- se specifico il valore di un parametro che ha un valore di default, devo farlo per tutti i seguenti parametri con default della funzione chiamata, ovvero

[15] un po' come i `#define` con il preprocessore

appena ometto un parametro, devo omettere tutti i successivi, altrimenti si creano problemi di ambiguità!

2.15 Variabili pubbliche e private

L'utente della mia struttura potrebbe voler fare gli inserimenti a mano, senza chiamare la funzione appropriata[16].
Ad esempio:

```
...
int x,y;
Pila s;
x = s.estrai();
y = s.contenuto[--marker];
```

Ma se la pila è vuota? Causo un errore!!
Vorrei quindi impedire all'utente di accedere alle variabili della mia struttra, in modo da costringerlo ad utilizzare le mie funzioni.
Il processo si dice *privatizzazione delle variabili.*
Quello che si fa è inserire all'interno della struttura la parola chiave `private:` (attenzione ai due punti!!!), che rende invisibili all'esterno tutte le variabili che vengono definite dopo di essa[17]. In tal modo solo le funzioni proprie della struttura possono accedere a tali informazioni.

Posso anche permettere all'utente di accedere alle istanze, ma attraverso particolari funzioni che predispongo io, dette funzioni *getter* e *setter.*
Se ad esempio volessi leggere o scrivere il valore dell'istanza `x`:

```
void setX(int k)
{
  x = k;
}


int getX()
{
  return x;
}
```

Il vantaggio sta anche nel fatto che attraverso le funzioni getter e setter posso tener traccia di quello che l'utente fa.

[16] pensando che così risparmia memoria e aumenta la velocità...
[17] Di solito i dati che si dichiarano privati sono le variabili di istanza e le funzioni di servizio

2.16 Le classi

A questo punto, mi sono talmente allontanato dal vecchio concetto di struttura, che mi sento di cambiare il nome a questo nuovo costrutto che ho creato. Con uno slancio di fantasia, lo chiamerò **classe**.
Una differenza fra i due costrutti sta nel fatto che

- nelle `struct` tutto quello che non viene definito è *pubblico*[18];
- nelle `class` tutto quello che non viene definito è *privato*.

Se per entrambi i costrutti specifichiamo se le istanze sono pubbliche o private, essi si equivalgono.

```
struct Pila
{
  private:
    int size;
    int defaultGrowthSize;
    int marker;
    int *contenuto;
    void cresci(int incremento);
  public:
    Pila(int initialSize);
    Pila();
    ~Pila();
    void copy(Pila *to);
    void inserisci(int k);
    int estrai();
    void stampaStato();
};

class Pila
{
  private:
    int size;
    int defaultGrowthSize;
    int marker;
    int *contenuto;
    void cresci(int incremento);
  public:
    Pila(int initialSize);
    Pila();
    ~Pila();
    void copy(Pila *to);
    void inserisci(int k);
    int estrai();
    void stampaStato();
};
```

[18] scriveremo `public:`

Capitolo 3

Overloading di metodi

3.1 Riepilogo su classi e strutture

Abbiamo introdotto le classi a partire dalle strutture. Abbiamo notato alcune differenze, ma in fin dei conti sembra tutto come prima.
Dove sta la grande differenza? Perché cambio addirittura nome?
Perché questi nuovi costrutti sono diversi dalle strutture, pur derivando da esse. In particolare sono più ricchi. Inoltre anche in altri linguaggi di programmazione ad oggetti tipi di dati complessi come questo vengono chiamati classi.

Ok, ma perché sono importanti?
Le classi sono molto utili nella gestione di programmi grossi. Ma la caratteristica fondamentale sta nel fatto che attraverso l'uso delle classi *cambia l'approccio al programma*! Nella programmazione tradizionale si utilizza il metodo basato sulla *decomposizione funzionale*: quando scrivo un programma, la prima domanda che mi pongo è

cosa voglio che il mio programma faccia?

Parto quindi dalla funzione `main` e decompongo il problema in componenti più semplici.
In pratica, scrivo la funzione principale, nella quale inserisco (ad alto livello) le operazioni che voglio implementare, passando poi al dettaglio una volta finito. Alla fine del processo, che viene detto *processo top-down*, mi pongo la domanda

come sono fatti i dati?

In tal modo, adeguo i dati al problema, ovvero la *struttura dati* è dettata dall'implementazione delle funzioni.
Lo svantaggio è proprio questa dipendenza, in quanto il codice scritto in questo modo risulta difficilmente matenibile e riutilizzabile in contesti diversi.

La progammazione ad oggetti invece si pone fin da subito la domanda sulla struttura dei dati, mentre la stesura della funzione principale è solo l'ultimo passo.

Definiremo quindi come **oggetto** una variabile di tipo `class`.
A volte gli oggetti vengono paragonati ad entità antropomorfe; in questo caso si parla di *attori*[1].

[1] si stanno facendo studi sulla programmazione orientata agli attori

A queste entità vengono mandati dei *messaggi* (ovvero viene passato un dato o viene dato un comando da eseguire) e vengono definiti per esse dei *comportamenti* (vengono implementate delle funzioni che essi possono eseguire).

3.2 Pile in stack o in heap

Come abbiamo già visto è possibile definire una Pila sia allocandola dinamicamente che in modo automatico:

```
Pila a;
```

Questo è un oggetto con le seguenti proprietà:

- allocato in stack;
- la locazione in cui è memorizzato ha un proprio nome associato: `a`;
- è quindi presente nella *reference table*, ovvero la tabella che il compilatore crea, contenente tutte le variabili dichiarate all'interno del programma.

```
Pila *z = new Pila;
```

Questo non è un oggetto, ma un puntatore a oggetto, quindi:

- allocato in heap;
- la locazione in cui è memorizzato non ha un proprio nome associato, ma solo un puntatore ad essa;
- non è quindi presente nella *reference table*.

Inoltre ho anche altre differenze se alloco in stack:

- non devo dichiarare utilizzando la `new`;
- uso l'operatore "punto" anziché l'operatore "freccia";
- non devo cancellare la variabile alla fine della funzione tramite la `delete`, perché essendo allocata in stack, viene cancellata automaticamente dal compilatore.

3.3 Rivediamo la funzione copia

Riprendiamo un attimo la funzione di copia della Pila, così come era stata definita all'inizio:

```
Pila *copia(Pila *from)
{
  // copia la pila versione iniziale
#ifdef DEBUG
  cout << ''entro in copia'' << endl;
#endif
```

```
    Pila *to = crea(from->size);
/* crea una pila della stessa dimensione della vecchia */
    to->defaultGrowthSize = from->defaultGrowthSize;
    for(int k = 0; k < from->marker; k++)
      to->contenuto[k] = from->contenuto[k];
    to->marker = from->marker;
    return to;
  }
```

Vediamo ora alcune varianti, per ripassare un attimo quanto visto finora.

3.3.1 Versione metodo

Riportiamo la versione che viene implementata all'interno della classe Pila.

```
    Pila * Pila::copia()
    {
      // copia la pila versione metodo
    #ifdef DEBUG
      cout << ``entro in copia'' << endl;
    #endif
      Pila *to = new Pila(size);
  /* crea una pila della stessa dimensione della vecchia */
      to->defaultGrowthSize = defaultGrowthSize;
      for(int k = 0; k < marker; k++)
        to->contenuto[k] = contenuto[k];
      to->marker = marker;
      return to;
    }
      // nella funzione chiamante si avrà invece
      Pila a;
      Pila *b = a.copia();
```

Facciamo alcune osservazioni:

- il parametro formale viene sostituito con il puntatore implicito `this`, che si riferisce alla struttura chiamante (in questo caso la pila `a`);
- il riferimento a `from` viene sostituito con `this`, oppure addirittura omesso, per quanto visto sullo scope delle varibili;
- alla definizione della funzione, premetto che il metodo appartiene alla struttura `Pila`, utilizzando l'operatore "::";
- al posto della funzione `crea` chiamo il costruttore, passandogli la dimensione giusta.

Notiamo che non avremmo potuto usare una variabile locale al posto del puntatore `*to`, in quanto al momento di restituire l'indirizzo, avremmo passato alla funzione chiamante un puntatore ad una locazione di memoria non più utilizzabile, in quanto la variabile era stata già deallocata.

3.3.2 Versione con references

Vediamo ora la versione che utilizza le references, anziché i puntatori

```
Pila& copia(Pila& from)
{
  // copia la pila versione references
#ifdef DEBUG
  cout << ``entro in copia'' << endl;
#endif
  Pila *to = crea(from.size)
/* crea una pila della stessa dimensione della vecchia */
  to->defaultGrowthSize = from.defaultGrowthSize;
  for(int k = 0; k < from.marker; k++)
    to->contenuto[k] = from.contenuto[k];
  to->marker = from.marker;
  return *to;
}
```

Cosa succede? Tratto il parametro formale come una semplice variabie, non come puntatore e quindi utilizzo l'operatore "punto"; in pratica utilizzo lo stesso principio dei puntatori.
Vediamolo meglio:

3.4 Parliamo di references

Partiamo con un esempio facile[2].

3.4.1 prima versione, non fa niente

```
void f(int z)
{
  z = 3;
}

int main()
{
  int k = 2;
  f(k);
}
```

In questo caso il risultato è il seguente:

tipo	nome	valore	indirizzo	scope
int	k	2	100	main
int	z	3	104	f
int	k	**2**	100	main

Come si può notare, la funzione `f` non ha nessun effetto sul `main`.

[2] per altri esempi, vedi le note del corso di Sebastiani

3.4.2 seconda versione, con i puntatori

```
void f(int* z)
{
  *z = 3;
}

int main()
{
  int k = 2;
  f(&k);
}
```

Stavolta il risultato è il seguente:

tipo	nome	valore	indirizzo	scope
int	k	2	100	main
int*	z	**100**	104	f
int	k	**3**	100	main

Nel caso dei puntatori, alla funzione viene passato l'indirizzo della variabile da modificare. Quindi le variazioni al valore del parametro formale della funzione chiamata si ripercuotono sul valore della variabile passata.

3.4.3 terza versione, con le references

```
void f(int& z)
{
  z = 3;
}

int main()
{
  int k = 2;
  f(k);
}
```

Alla funzione viene passata la variabile, ma il paramtero formale appare con la `&`, che sta a significare che l'indirizzo del parametro *coincide con l'indirizzo della variabile passata come parametro alla funzione*!!
Non viene quindi allocato nuovo spazio, ma semplicemente viene dato un nuovo nome alla variabile `k`, in modo che la cella sia accessibile con due nomi diversi, a seconda della funzione in cui ci si trova[3].

tipo	nome	valore	indirizzo	scope
int	k nel main z nella f	2	100	main
int&	z		**100**	f
int	k	**3**	100	main

[3] `k` nella funzione `main`, `z` nella funzione `f`

Tornando quindi alla funzione `copia`, notiamo che la variabile viene passata per reference, e quindi non viene copiata, ma viene dato un alias alla sua cella di memoria. Inoltre alla fine restituisco `*to`, perché la funzione richiede in uscita una variabile, non un puntatore. Devo quindi dereferenziare[4].

3.5 Copy constructor

Consideriamo il seguente codice:

```
void f(Pila s)
{
  s.estrai();
  s.inserisci();
  cout << ''s'';
  s.stampaStato();
}

int main()
{
  Pila w(5); // questo è un oggetto!!
  w.inserisci(1);
  cout << ''w'';
  w.stampaStato();
  f(w);
  w.stampaStato();
}
```

In fase di compilazione questo codice genera un errore, in quanto passando la pila `w` alla funzione, di essa viene fatta una copia "sbagliata". Il problema sta nel fatto che non viene chiamato il costruttore all'interno della funzione chiamata per inizializzare la pila, ma viene creata comunque un'istanza nuova. Avviene cioè una copia di una pila senza utilizzare il metodo che abbiamo implementato, e quindi il puntatore che viene creato punta alla stessa locazione di memoria di quello passato e mi genera un conflitto.
Infatti, poiché all'interno della funzione `s` e `w` *non sono distinte*, il programma va a scrivere sempre sullo stesso vettore, che però alla fine della funzione viene deallocato (viene chiamato il distruttore). Quindi se si torna nella funzione principale e si chiede di stampare lo stato della pila, si trovano valori estranei, dovuti proprio alla deallocazione.
Notiamo che questo è lo stesso problema che si aveva con il metodo `crea`, nel quale avevamo due operazioni distinte (allocazione e inizializzazione) compiute dalla stessa funzione. Anche in questo caso quindi separeremo le due operazioni, lasciando al sistema il compito di allocare la memoria e rivolgendo le nostre attenzioni soltanto alla copia della pila.
Il nuovo metodo è un metodo speciale, che va sotto il nome di *copy constructor*.

[4] l'*aliasing* delle variabili può essere fatto anche all'interno della funzione `main`, ma è meglio evitarlo, per non causare ambiguità.

```
Pila::Pila(Pila &from)
{
#ifdef DEBUG
  cout << ``entro in copy constructor''<< endl;
#endif
  if(this==&from) return;
  if(size!=from.size)
    contenuto=new int[from.size];
  size=from.size;
  defaultGrowthSize=from.defaultGrowthSize;
  for(int i=0; i<marker; i++)
    contenuto[i] = from.contenuto[i];
  marker = from.marker;
}
```

Anche questo è un esempio di overloading del costruttore, ma ha firma diversa rispetto ai precedenti. Anche questo non ritorna alcun dato, ma non deve essere dichiarato `void`, perché ha un side effect sulla struttura che viene creata. Come per costruttore e distruttore, è sempre bene creare anche questo metodo speciale, eventualmente commentandolo quando non serve.
Notiamo ora che in questo metodo speciale si opera sulla struttura di *destinazione*, non su quella di *origine*. Infatti, come nel caso del copy constructor, il puntatore implicito `this` si riferisce alla *nuova* struttura. C'è quindi un rovesciamento rispetto a prima: mentre prima avevo

- `this` riferito all'origine dei dati;
- `nome` riferito alla destinazione dei dati.

Adesso invece avremo

- `from` riferito all'origine dei dati;
- `this` riferito alla destinazione dei dati.

Inoltre il parametro `from` è passato per referenza. Sarà quindi necessario utilizzare l'operatore "punto".
Notiamo che come facevamo con il costruttore, anche in questo caso il metodo *deve* essere implementato, per evitare errori di allocazione. In presenza invece di strutture semplici non serve, perché il compilatore utilizza il copy constructor che ha di default, che copia le celle di memoria bit per bit.

È possibile evitare che l'utente della mia pila utilizzi il copy constructor, semplicemente dichiarandolo privato all'interno della classe. In questo modo `f(w);` non può essere chiamato, se non dalle istanze della classe. In questo modo non serve nemmeno inserire il codice del copy constructor nel file di libreria, in quanto la funzione non può essere chiamata nel programma.

3.5.1 Shallow copy VS deep copy

Il copy constructor è una soluzione al problema della scelta di *che tipo di copia* serve in un determinato punto del programma:

shallow copy , ovvero superficiale, che copia bit per bit da una locazione di memoria all'altra[5]

deep copy , ovvero copia profonda, che copia le istanze della classe e gli eventuali riferimenti esterni che sono puntati da membri della classe stessa

Se ad esempio creo una classe `Libro`, vorrò creare un'istanza `Indice`, che altro non è che un vettore di puntatori, ognuno dei quali punta ad esempio ad un `Capitolo`, visto come vettore di stringhe. Se però volessi creare anche un indice analitico (ovvero volessi fare una copia dei puntatori), non dovrei copiare anche tutti i capitoli, ma soltanto i riferimenti ad essi.
In questo caso utilizzerò una shallow copy.
In generale, utilizzerò

shallow copy per copiare strutture contenenti solo variabili di tipo primitivo;

deep copy per copiare strutture a più livelli[6].

3.6 Operator overloading

Nel nostro cammino nella programmazione vorremmo portare il linguaggio sempre più vicino al linguaggio corrente, raggiungendo un livello più astratto possibile. Negli anni sono stati fatti molti progressi:

Persona		
⋮		
OOP		
`C++`, Pascal	→	Tipi personalizzati
Fortran	→	Non implementa le strutture
Assembly Language	→	Più astratto del linguaggio macchina, linguaggio simbolico
Linguaggio macchina	→	Lavora con numeri e registri fisici
Macchina		

I tipi di dato primitivi sono dotati di operazioni (somma, prodotto...) che vengono definiti all'interno delle librerie standard del linguaggio. In tutti i linguaggi di programmazione orientata ad oggetti è possibile definire nuovi tipi di dato, ma non in tutti è possibile anche ridefinire i vecchi operatori per adattarli alle nuove strutture di dato. In `C++` è concesso il polimorfismo delle funzioni e quindi in particolare è possibile il polimorfismo degli operatori. In altre parole è possibile caricare su un simbolo che ha già un significato prestabilito, un nuovo significato, che viene disambiguato dal primo grazie alla firma.

[5] è quello di default
[6] composte cioè da variabili primitive, puntatori, ma anche da vettori, vettori di vettori...

In generale la sintassi per l'overloading di un operatore è

```
ClassName & ClassName::operator□(ClassName & from)
```

dove □ sta ad indicare un operatore predefinito[7].
In questo modo avremo che

`a.operator□(b)` diventa `a □ b`

Si noti che il parametro del metodo viene passato *per reference* e viene restituito un oggetto appartenete alla *stessa classe* del metodo chiamato. Questo vale anche per l'operatore di assegnazione, per due motivi:

- per omogeneità con la sintassi dell'operator overloading;
- in generale, in `C` una assegnazione ritorna un valore, in particolare ritorna il valore assegnato. Procederemo quindi allo stesso modo, per omogeneità con le funzioni di assegnazione per i tipi di dato primitivi.

3.6.1 Overloading di = (assegnazione)

Cominciamo con l'operatore di assegnazione. Il codice sarebbe infatti molto più leggibile, se potessimo scrivere una riga del tipo

```
Pila a, b;
...
a = b;
```

Andremo in pratica ad operare allo stesso modo del copy constructor e in particolare andremo incontro agli stessi problemi trovati con il copy constructor.
Cominciamo con una implementazione ingenua:

```
Pila & Pila::operator=(Pila &from)
{
  size = from.size;
  defaultGrowthSize = from.defaultGrowthSize;
  marker = from.marker;
  contenuto = from.contenuto;
}
```

Questo metodo va bene, ma *solo se ho soltanto istanze primitive*. Quindi in questo caso non è corretto, in quanto copia soltanto l'indirizzo a cui il puntatore è riferito, ma non copia il contenuto del vettore puntato. In tal modo se modifico la pila `from`, modifico anche la pila di destinazione.

Un altro problema apparente è il fatto che questo metodo cerca di accedere a variabili di istanza che non appartengono all'oggetto chiamante. Ma questa operazione è veramente proibita? Se così fosse, dovrei definire per ogni variabile di istanza un metodo setter e uno getter.
In realtà le istanze dichiarate nella parte privata della classe sono tali *solo* per oggetti di *altre classi*.

[7] operatori come +, -, *, /, ++...

Questo significa che due oggetti appartenenti alla stessa classe possono *vedere le rispettive parti private*[8], semplicemente per il fatto che io programmatore so come è fatta la classe e quindi è assurdo che mi protegga da me stesso,[9]. Al contrario l'utente del mio codice non sa come è fatto quest'ultimo e quindi non gli dò la possibilità di accedervi.

Proviamo allora con un'implementazione diversa:

```
Pila & Pila::operator=(Pila &from)
{
  if(this==&from) return *this;
  size=from.size;
  defaultGrowthSize=from.defaultGrowthSize;
  delete []contenuto;
  contenuto = new int[from.size];
  for(int i=0; i<marker; i++)
    contenuto[i] = from.contenuto[i];
  marker = from.marker;
  return *this;
}
```

Questa versione è corretta e presenta alcuni miglioramenti rispetto al copy constructor:

- la prima riga di codice controlla se si sta verificando una *autoassegnazione*, ovvero se si sta assegnando una pila a se stessa: ad esempio

  ```
  Pila a;
  Pila &b;
  b = a;
  a = b;
  ```

- `delete []contenuto` è necessario, perché il vettore era già stato inizializzato in precedenza, ma a priori non so se la sua dimensione va bene per il nuovo vettore che devo inserire. Inoltre se questa riga di codice non viene eseguita, si verifica un memory leak;

- il metodo ritorna una reference, quindi è necessario dereferenziare il parametro implicito, aggiungendo l'asterisco.

Una possibile modifica potrebbe essere l'inserimento di un controllo sulla dimensione del vettore destinazione. Se questa coincide con la dimensione del vettore da copiare, non viene eseguita la deallocazione e riallocazione della memoria.

```
if(size!=from.size) {
  delete []contenuto;
  contenuto = new int[from.size];
  size = from.size;
}
```

[8] e non ci dilunghiamo sui doppi sensi...

[9] anche se a volte dovrei...

Possiamo ora chiederci se quest'ultima versione considerata (con il controllo sulla dimensione) sia più o meno efficiente rispetto alla precedente. La classica risposta è *dipende*! Dipende dalla frequenza con cui chiamo questa funzione e dalla frequenza con cui capita che chiamante e parametro passato abbiano la stessa dimensione: se hanno la stessa dimensione molto di frequente, il secondo codice è più efficiente, perché mi permette di fare meno calcoli; se invece avviene più spesso che le dimensini siano diverse, questa versione è meno efficiente, in quanto faccio un controllo in più per niente.

Ritorniamo un attimo al problema dell'autoassegnazione. Se non effettuo il test all'interno del metodo, ho possibile perdita di dati, in quanto se le due pile coincidono, quando dealloco il vettore `contenuto`, perdo il collegamento con i valori che volevo copiare, generando un errore. È quindi sempre bene inserire un test per l'autoassegnazione in questo tipo di metodi. Ma devo stare attento a come scrivo il test. Esiste infatti una notevole differenza fra questi due test:

```
if(this == &from)
  ...

if(*this == from)
  ...
```

Il primo test controlla soltanto gli indirizzi di base delle due pile, mentre il secondo ha come operandi *due pile*, ma se non dispongo di un overloading per l'operatore di confronto, ho errore.

3.6.2 Overloading di ==

Un altro concetto che è bene riguardare è il concetto di *uguaglianza*. Due pile sono uguali se coincidono campo per campo e se hanno soltanto alcuni campi uguali? Al momento dell'implementazione della funzione è bene porsi queste domande, documentando le decisioni prese in merito, perché queste possono influenzare eventuali scelte future.
Ad esempio una scelta possibile è la seguente: due pile sono uguali se hanno gli stessi elementi al loro interno (e quindi se `marker` è lo stesso per entrambi e gli elementi di `contenuto[]` sono gli stessi due a due). Se implementiamo questa scelta, non faremo il controllo sulle dimensioni.
Se invece decidiamo che due pile sono uguali solo se coincidono campo per campo[10], una possibile implementazione sarà la seguente:

```
bool Pila::equals(Pila &s)
{
#ifdef DEBUG
  cout<<"entro in equals"<<endl;
#endif
  if(size!=s.size) return false;
  if(marker!=s.marker) return false;
  if(defaultGrowthSize!=s.defaultGrowthSize)
    return false;
```

[10] ovviamente gli indirizzi a cui puntano i puntatori saranno diversi, a meno che le due pile non siano proprio la stessa

```
  for (int k=0;k<marker;k++)
    if (contenuto[k]!=s.contenuto[k]) return false;
  return true;
}

bool Pila::operator==(Pila &from)
{
  return equals(from);
}
```

Notiamo che `equals` è una *funzione*, non un metodo, che viene chiamata dal metodo che fa l'overloading dell'operatore di uguaglianza. Questo tipo di implementazione è però svantaggioso se la funzione viene utilizzata di frequente, in quanto deve essere chiamata e deve essere allocato lo spazio per le sue variabili, con una perdita di tempo notevole. La soluzione è dichiararla `inline`, o inserire l'implementazione della funzione all'interno della definizione della classe.
Altra caratteristica è il tipo di dato ritornato: `bool` o `boolean`. Alcuni compilatori danno questi tipi come predefiniti, altri no. Un metodo rapido per definirli è mapparli sugli interi attraverso la definizione di una `enum`, che assegni alla costante `false` il valore zero e alla costante `true` il valore uno[11]:

```
enum bool={false, true}
```

3.6.3 Overloading di <<

Vorrei poter scrivere qualcosa del tipo

```
Pila a;
cout << a;
```

Il problema sta nel fatto che non posso definire un overloading per l'operatore <<, perché questo è un metodo di una classe diversa[12]. Quindi il nuovo metodo apparterrebbe alla classe di `cout` e non alla classe `Pila`. Ma per poterlo fare, dovrei avere a disposizione il codice sorgente della classe `ostream`, cosa che va contro al principio di incapsulazione dell'informazione.
Adotto allora un'altra soluzione: non includo la funzione nella classe e scrivo

```
ostream & operator<<(ostream &sx, const Pila &dx)
{
#ifdef DEBUG
  cout << ``entro in stampa'' << endl;
#endif
  sx << ``======================'' << endl;
  sx << ``size = '' << dx.size << endl;
  sx << ``defautlGrowthSize = ''
     << dx.defautlGrowthSize << endl;
```

[11] in generale, qualsiasi valore non nullo va bene
[12] appartiene alla classe `ostream` di output

```
    sx << ``marker = '' << dx.marker << endl;
    for(int k = 0; k < dx.marker; k++)
      sx << ``[``<<(dx.contenuto[k]) <<'']'';
    sx << endl;
    sx << ``======================'' << endl;
    return sx;
  }
```

Al posto di `cout` compare `sx`, che ha lo stesso comportamento, poiché è dichiarato come appartenente alla stessa classe di `cout`[13].
La funzione ritorna `sx`, ovvero una variabile della classe `ostream`. Questa riga è necessaria, in quanto se concateno gli output (ad esempio `cout << a << b;`) ogni chiamata della funzione deve restituire un ostream, per permettere la chiamata della funzione successiva (ovvero per avere a sinistra una ostream).
La Pila passata viene preceduta da `const`, quindi non viene modificata all'interno della funzione. Questo è anche un modo per rassicurare l'utente, poiché egli vede il prototipo della funzione all'interno del file header e capisce che la pila che viene passata non subirà alcuna modifica all'interno. Notiamo però che ci sono due scritture possibili:

- `Pila * const dx` mantiene costante l'indirizzo della pila
- `const Pila &dx` mantiene costante il contenuto della pila

Quindi le reference sono preferibili ai puntatori in termini di sicurezza dei dati.
In questa versione della funzione c'è però un problema: non posso accedere ai campi interni della pila, perché la funzione è esterna alla pila. Posso però accedere ai valori contenuti nel vettore, in quanto questo *non fa parte della pila*, ma è soltanto puntato da un membro della stessa.
Per salvare capra e cavoli, basta mettere davanti alla definizione della funzione la parola chiave `friend`, che dice al compilatore che la funzione è "amica" della classe e quindi può accedere anche ai campi privati.
Inoltre notiamo che, pur essendo all'interno della pila, la funzione è dotata di due parametri. Questo significa che *non c'è il parametro implicito*!!

Ma a cosa serve creare tutti questi metodi e cambiare il proprio approccio alla programmazione...?

- vengono eliminati i problemi derivanti dalla suddivisione del lavoro;
- l'uso dei metodi è preferibile rispetto all'uso di funzioni;
- vengono ridotti i costi per progetti grossi;
- vengono ridotti costi del mantenimento del codice;
- si aumenta la riusabilità del codice.

[13] a priori non agisco su `cout`, ma su un'istanza che appartiene comunque a `ostream`, come un file o una socket

3.6.4 È sempre bene utilizzare l'overloading?

No. L'operator overloading è possibile[14] *soltanto* per operatori predefiniti, a causa di possibili conflitti nella precedenza degli operatori.
È quindi consigliabile utilizzare l'operator overloading solo quando è strettamente necessario[15]. Nel nostro caso è utile ai fini della leggibilità del codice.

Come ultima cosa introduciamo un sistema per testare la consistenza dei metodi implementati. È sufficiente inserire una breve funzione di test, che combini gli effetti degli operatori, per vedere se danno i risultati sperati:

```
Pila a;          // costruttore
a.inserisci(7);
Pila b;          // costruttore
Pila c = a;      // copy constructor
b = a;           // overloading di =
if(!(c == b))    // overloading di ==
  cout << ``errore!!'' << endl;
```

Questo frammento di codice chiama tutti gli operatori introdotti[16]. Se le implementazioni sono tutte esatte, non dovrebbe stampare alcun messaggio di errore. Ricordiamo però che questa *non è* una dimostrazione che il codice funziona, poiché esiste un bel metateorema di informatica teorica che afferma che

Non è possibile dimostrare che un programma sia corretto.

Notiamo infine che se abbiamo definito l'overloading di `==`, questo *non significa* che sia definito anche l'overloading di `!=`

3.7 Cosa stampa questo codice?

In questa sezione consideriamo alcuni pezzi di codice e cerchiamo di capire quale sia l'output, supponendo di aver attivato la modalità di debug.

1. Consideriamo il seguente codice:

   ```
   Pila k;
   Pila z = k;
   ```

   La prima riga è chiara. Chiama il costruttore.
   La seconda, letta da sinistra a destra sempbrerebbe chiamare due volte il costruttore (una volta per la definizione di `z` e una volta nel metodo `operator==`) e una volta il distruttore[17].
   In realtà non è così. Infatti il compilatore si accorge che in questo modo spreca soltanto tempo e memoria e quindi taglia la testa al toro, chiamando direttamente il copy constructor.
   Quindi in questo caso, le due scritture seguenti

[14] ma non sempre consigliabile...
[15] ad esempio per strutture matematiche
[16] Per capire perché la quarta riga invoca soltanto il copy contructor, vedi la sezione 3.7
[17] sempre nel metodo `operator==`

```
Pila z = k;   oppure   Pila z;
                       z = k;
```

non sono equivalenti!.

2. Consideriamo il seguente codice

```
Pila z[2];
```

Questo è un vettore di pile. In totale occupa 32 bytes (16 per ogni pila) più lo spazio allocato per i due vettori.
L'output di questo codice sarà formato da 2 chiamate al costruttore (uno per ogni pila). In sequenza le operazioni che i compilatore farà saranno:

- riservare uno spazio per la prima pila;
- chiamare il costruttore void sulla prima pila;
- riservare uno spazio per la seconda pila;
- chiamare il costruttore void sulla seconda pila;

Notiamo inoltre che *non è possibile* scrivere

```
Pila z[2](3);
```

ovvero non è possibile passare a parametro la dimensione della pila, che sarà perciò inizializzata con la dimensione di default. Quindi se voglio costruire vettori di pile *è necessario implementare un costruttore void*[18].

3. Consideriamo il seguente codice

```
Pila z[];
```

Questo codice non stampa **NIENTE**, è come se scrivessi `Pila *z;`
Non definisco quindi una pila, ma soltanto un puntatore a pila. Perché diventi una vera pila devo inizializzarlo, scrivendo `z = new Pila[2];` che ha lo stesso output del caso precedente e restituisce l'indirizzo di base di un vettore di due pile.

4. Come ultimo caso consideriamo

```
Pila z[];
z = new Pila[3];
delete z;
delete []z;
```

Le operazioni che vengono compiute sono le seguenti:

- la prima riga definisce soltanto un puntatore a `Pila`;
- la seconda riga chiama 3 volte il costruttore, allocando quindi 48 byte più lo spazio per tre vettori;

[18] oppure un costruttore provvisto di valore di default

- la terza libera 48 byte ma non i tre vettori, generando così un memory leak, in quanto non viene chiamato il distruttore!
- la quarta fa la cosa giusta, liberando 48 byte e deallocando i tre vettori (chiama quindi 3 volte il distruttore).

Da questo esempio possiamo quindi concludere che il simbolo di vettore nella `delete` non serve se si tratta di un vettore di dati di tipo primitivo, mentre *è necessario* se è un vettore di dati di tipo complesso, perché altrimenti si genera un memory leak.

3.8 La classe stringa

Cerchiamo di definire una classe `String`, partendo dalla definizione di stringa presente nel `C`, ovvero da un vettore di caratteri.
La prima versione sarà

```
#include<iostream>
using namespace std;

class String
{
private:
// variabili di istanza
  char *base;
  int length;

// metodi di servizio
  void strcpy(char *to, const char *from);
  int strlen(const char *s);
  bool strcmp(char *s, char *w);
  void append(char *s, char *w);

// metodi essenziali
  String();
  String(char *s);
  String(String &s);
  ~String();
  String & operator=(const String &s);
  String & operator=(const char *s);
  friend ostream& operator<<(ostream&sx, const String&dx);

// overloading di +
  String & operator+(String &s);
  String & operator+(char *s);
  friend String & operator+(char *s, String &t);
  bool String::equals(String &s);
}
```

Alcune osservazioni su questo codice:

- la seconda riga imposta lo spazio dei nomi, ovvero permette di omettere l'estensione dei nomi dei file da includere;
- la stringa è definita da un puntatore a caratteri e da una lunghezza. È però necessario stabilire se la lunghezza comprende o meno il terminatore della stringa;
- i metodi aggiuntivi hanno un parametro implicito, che però non viene utilizzato. Li definisco in questo modo, così da poter scrivere ad esempio

```
String a;
char z[10];
char w[10];
a.strcpy(z,w);
```

  in questo modo ho la possibilità di utilizzare questi metodi anche con vettori di caratteri, oltre che con le nuove stringhe;
- definisco più overloading per l'operatore `=`, in modo da poter assegnare sia le stringhe nuove che quelle vecchie;
- gli overloading dell'operatore `+` sono tre e sostituiscono in parte la funzione della libreria standard `append`, in modo da poter concatenare stringhe nuove con stringhe vecchie. Vedremo in seguito che l'ultima implementazione non è possibile. Inoltre l'operatore `+` che concatena una stringa vecchia con una nuova è dichiarato amico, perché non ha un parametro del tipo della classe come primo parametro, ma deve poter accedere ai campi privati della classe.

3.8.1 Costruttori e Distruttore

Ci possono essere varie implementazioni, una possibile è la seguente, che inizializza il vettore di caratteri con dimensione uno:

```
String::String()
{
  length = 0;
  base = new char[1];
  *base = 0;
}
```

Oltre che il costruttore vuoto, implementiamo anche il costruttore a cui si passa una stringa:

```
String::String(char *s)
{
  length = strlen(s);
  base = new char[length + 1];
  strcpy(base,s);
}
```

Poiché la dimensione non viene passata a parametro, posso calcolarla attraverso le funzioni di servizio.
Sullo stesso principio si basa anche il copy constructor:

```
String::String(String &s)
{
  length = s.length;
  base = new char[length + 1];
  strcpy(base,s);
}
```

Per quanto riguarda il distruttore, è invece sufficiente deallocare il vettore:

```
String::~String()
{
  delete []base;
}
```

3.8.2 overloading di =

Seguo lo stesso schema utilizzato per le pile: poiché passo come parametro una stringa, prima di tutto controllo che non si verifichi una autoassegnazione.

```
String::String & operator=(const String &s)
{
  if(&s == this) return *this;
  length = s.length;
  delete []base;
  base = new char[lenght + 1];
  strcpy(base, s.base);
  return *this;
}
```

Aggiungiamo anche un altro overloading per questo operatore, utile se vogliamo assegnare stringhe vecchie:

```
String::String & operator=(const char *s)
{
  length = strlen(s);
  delete []base;
  base = new char[lenght + 1];
  strcpy(base, s);
  return *this;
}
```

Nota: non potrei scrivere `String t = '' ''`, perché questa sintassi chiama il copy constructor, che però richiede come parametro una String: infatti il parametro passato non è una stringa, ma un vettore di caratteri e questo provoca un errore di compilazione.

3.8.3 overloading di << e funzione `equals`

L'operatore di output non fa altro che stampare la stringa:

```
ostream & operator<<(ostream&sx, const String&s)
{
  sx << s.base;
  return sx;
}
```

La funzione equals confronta le due stringhe. **Attenzione:** la funzione `strcmp` ritorna zero se le stringhe sono uguali! È quindi necessario negare la quantità ritornata per avere il risultato richiesto:

```
bool String::equals(String &s)
{
  return !strcmp(base,s.base);
}
```

3.8.4 overloading di +

Questo metodo ci darà da penare...
Partiamo da una implementazione ingenua:

```
String::String & operator+(String & right)
{
  char *temp = new char[length+right.length+1];
  char *temp1 = temp+length;
  strcpy(temp, base);
  strcpy(temp1, right.base);
  return new String(temp);
}
```

Le operazione compiute da questo metodo sono:

- creazione di una stringa di caratteri temporanea della lunghezza richiesta;
- creazione di un puntatore che punta alla cella di memoria che contiene il terminatore della prima stringa;
- copia della prima stringa nella nuova locazione;
- copia della seconda string nella nuova locazione[19];
- ritornare una nuova stringa inizializzata con la striga definita prima.

Ma questa versione è sbagliata, perché al termine della funzione, viene restituito un puntatore, non una reference; quindi si perde il riferimento a `temp` e si ha memory leak.

[19] notiamo che queste due ultime operazioni non possono essere invertite, perché altrimenti il terminatore della prima stringa *non* viene sovrascritto

Proviamo quindi a migliorare il codice:

```
String::String & operator+(String & right)
{
  char *temp = new char[length+right.length+1];
  char *temp1 = temp+length;
  strcpy(temp, base);
  strcpy(temp1, right.base);
  String *r = new String(temp);
  delete temp;
  return *r;
}
```

Le operazioni aggiunte in questa versione sono:

- salvare il puntatore;
- deallocare la variabile temporanea, in modo da non avere memory leak;
- restituire il puntatore salvato.

Funziona? Lo vedremo dopo (vedi 3.8.5). Intanto vediamo gli altri metodi.
La cosa che sembrerebbe più ovvia è utilizzare il metodo appena implementato per le altre funzioni, e così facciamo:

```
String::String & operator+(char *s)
{
  String w(s);
  return (*this)+w;
}
```

Notiamo che questa funzione pùo essere scritta come metodo, perché il primo parametro della funzione è del tipo della classe e si può rendere implicito.

```
String::String & operator+(char *s, String &t)
{
  String w(s);
  return w+t;
}
```

Osserviamo in questo caso che questa funzione non è una member function, perché il primo parametro passato non è del tipo della classe, ma non è nemmeno dichiarata `friend`, poiché non è necessario. Infatti la funzione non accede a nessun campo privato della classe.
Se però invece di utilizzare un metodo già fatto avessi voluto implementare daccapo la funzione, avrei dovuto accedere a campi privati e quindi avrei dovuto dichiararla amica.

Vediamo ora l'ultimo metodo di overloading per la somma, ovvero quello che prende come parametri due vettori di caratteri:

```
String & operator+(char *s, char *t)
{
  String w(s);
  String v(t);
  return w+v;
}
```

Questo *non va bene*!
Infatti un overloading di operatori deve avere un parametro formale di tipo classe esplicitato, oppure deve essere una member function[20]. In questo caso invece ho soltanto due puntatori a carattere; in pratica la firma della funzione sarà `operator+(int, int)`, in quanto i puntatori in fin dei conti sono interi.
Una soluzione potrebbe quindi essere quella di rendere la funzione una member function, ma anche in questo caso si genera un errore, in quanto la somma è un operatore binario, mantre se rendo la funzione membro della classe, vengono passati tre parametri, di cui uno implicito!
Morale della favola: **non posso implementare questa funzione!**

3.8.5 operatore+ giusto

Dove sta il problema dell'operatore somma?
Il problema sta nel fatto che quando vengono concatenate le operazioni, ad esempio scrivendo `a+b+c+d`, per ogni somma viene creato un oggetto temporaneo, che poi non viene deallocato. Questo causa una serie di memory leak. Infatti:

$$\begin{aligned} e &= \underbrace{a+b}_{t_1}+c+d \\ &= \underbrace{t_1+c}_{t_2}+d \\ &= \underbrace{t_2+d}_{t_3} \\ &= t_3 \end{aligned}$$

In questo caso ho ben tre strighe temporanee che si sono perse.
Come risolvo il problema? Ritorno alla definizione della classe, aggiungendo una variabile di istanza di tipo booleano, che dica se l'oggetto è temporaneo.

```
class String
{
private:
  char *base;
  int length;
  bool temporary;
  ...
}
```

[20] in modo da avere automaticamente un parametro implicito

Nel metodo somma invece, faccio un controllo per vedere se la stringa su cui sto operando sia temporanea; in caso affermativo la cancello prima del termine della funzione, in modo da non generare memory leak.

```
String::String & operator+(String & right)
{
  char *temp = new char[length+right.length+1];
  char *temp1 = temp+length;
  strcpy(temp, base);
  strcpy(temp1, right.base);
  String *r = new String(temp);
  r->temporary = true;
  delete temp;
  if(this->temporary) delete this;
  return *r;
}
```

In questo modo elimino gli oggetti temporanei `t1` e `t2` e anche il parametro implicito, se questo punta a qualcosa di temporaneo. Ma lo elimino subito prima di ritornare il puntatore alla stringa, perché da quel momento in poi non devo più accedere ai suoi campi.
Allo stesso modo farò anche nell'assegnazione:

```
String::String & operator=(const String &s)
{
  length = strlen(s);
  delete []base;
  base = new char[lenght + 1];
  strcpy(base, s);
  if(s.temporary) delete &s;
  return *this;
}
```

Così facendo, elimino anche l'ultimo oggetto temporaneo genrato dalla concatenazione vista sopra, ovvero `t3`.
A questo punto posso vedere se la mia classe funziona, attraverso l'utilizzo di *variabili statiche*.
Dichiaro semplicemente come statica una variabile di istanza della classe, in modo che essa venga *condivisa* da tutti gli oggetti di quella classe[21]. Ora quello che posso fare è ad esempio incrementare tale variabile ogni volta che chiamo il costruttore e decrementarla ogni volta che chiamo il distruttore, in modo da avere sempre sott'occhio il numero di oggetti vivi presente nel mio programma. Quando però nel `main()` devo inizializzare questa variabile, ho una sintassi speciale:

```
...
int String::n = 0;
int main() { ... }
```

[21] questo è un ottimo meccanismo di intercomunicazione fra gli oggetti.

La variabile statica deve essere inizializzata *fuori dal main*, come una variabile globale, anteponendo il nome della classe di appartenenza, come se fosse la dichiarazione di un metodo. Inoltre, ogni volta che verrà utilizzata, dovrà essere sempre accompagnata dal nome della sua classe, perché sia chiaro che ci si riferisce proprio a quella.

Capitolo 4

I template

All'inizio del corso avevamo affrontato il problema di come rendere la pila più generale possibile, in modo da poterla riutilizzare in qualsiasi contesto. Siamo riusciti a renderla abbastanza flessibile, capace di adattarsi al numero di dati da inserire, ma allo stato attuale è ancora fortemente limitata dal un fattore: il tipo di dato immagazzinato *deve* essere un intero. Sarebbe utile riuscire a rendere la pila indipendente dal tipo di dato immagazzinato. Questo implica ovviamente che il tipo di dato andrà modificato ovunque necessario; ad esempio anche nel parametro passato a tutte le funzioni che operano sulla pila.
La nostra Pila può quindi essere pensata come una classe parametrica, nella quale il parametro definisce il tipo di dato che caratterizza la Pila stessa.

4.1 Prima versione: il `#define`

Una prima modifica che si potrebbe fare è dare la possibilità di cambiare il tipo di dato inserito, definendolo attraverso il costrutto `#define`:

```
#define TIPO char
class Pila
{
  int size;
  int defaultGrowthSize;
  int marker;
  TIPO * contenuto;
  void cresci(int increment);
public:
   Pila(int initialSize);
   Pila(Pila & to);
   Pila & Pila::operator=(Pila & from);
   ~Pila();
   void inserisci(TIPO k);
   TIPO estrai();
   int equals(Pila &s);
   int operator==(Pila & from);
   friend ostream & operator<<(ostream &sx,const Pila &dx);
};
```

Questa prima soluzione funziona, ma ha alcuni limiti: il più evidente è che ad esempio non possiamo avere all'interno dello stesso programma una pila che gestisca interi ed una che gestisca floating point, in quanto l'uso del costrutto di definizione vincola il tipo di dato una volta per tutte (almeno all'interno dello stesso file).

4.2 Seconda versione: i `templates`

Occorre uno strumento più flessibile e potente, che permetta, per ogni istanza di Pila, di specificare di quale tipo di Pila si tratti. Ci serve insomma un cosiddetto *meta-tipo di dato*[1], ovvero qualcosa che ci permetta di specificare cioè che *tipo di tipo di dato* vogliamo.
Il `C++` fornisce un costrutto che ci risolve il problema: il `template`[2].
Il nostro header file, usando i templates, diviene:

```
template <class TIPO> class Pila {
  TIPO * contenuto;
  int size;
  int marker;
  int defaultGrowthSize;
  void cresci(int increment);
public:
  Pila();
  Pila(int initialSize) ;
  void inizializza(initialSize);
  ~Pila() ;
  void aggiungi(TIPO k);
  TIPO estrai();
  void stampaStato();
  Pila(Pila & from);
  bool Pila::operator==(Pila & from);
  Pila & Pila::operator=(const Pila & destro);
  friend ostream & operator<<(ostream &sx,Pila &dx);
};
```

Lo header file rimane quindi identico a quello che avremmo usato con una `typedef`, ma con una importante differenza: la class non è una classe vera e propria, ma il template (modello) di una classe, con parametro TIPO. (Il nome del parametro è arbitrario, e di parametri ve ne possono essere più d'uno).
L'invocazione di una Pila avverrà ora specificandone il particolare tipo desiderato: potremo quindi scrivere

```
Pila <int> a;
Pila <float> b(10);
Pila <char> a[3];
```

[1] Il prefisso meta, serve per reiterare l'entità di cui si parla: un meta-linguaggio è un linguaggio usato per definire linguaggi, una meta-dato è un dato che descrive i dati...
[2] In altri linguaggi lo stesso concetto è denominato `generic`

In questo modo vengono create una Pila di interi, una Pila di floating point un vettore di tre Pile di caratteri, riutilizzando il codice generico della Pila, ma specializzandolo di volta in volta, inserendo il tipo di dato desiderato.

4.3 Implementazione

L'implementazione del template di una classe deve seguire alcune semplici regole, che presentano due possibilità:

- inserire il codice dell'implementazione all'interno della definizione del template;
- scrivere il codice dell'implementazione all'esterno della definizione del template.

Nel primo caso la sintassi è la stessa che avremmo usato con una `#define`.Ad esempio, il costruttore verrà implementato nel seguente modo:

```
Pila(int initialSize=15)
{
  size=initialSize;
  defaultGrowthSize=initialSize;
  marker=0;
  contenuto=new TIPO[initialSize];
}
```

L'implementazione rimane quindi sostanzialmente invariata rispetto alla versione originale. L'unica differenza infatti sta nel fatto che il vettore `contenuto` viene associato ad un array di `TIPO` anziché ad un array di interi.
Nel secondo caso la scrittura è differente. Invece che specificare che ad esempio il costruttore appartiene alla classe `Pila` (come facevamo di solito scrivendo `Pila::Pila`), dobbiamo specificare che appartiene al *template* di una classe:

```
template <class TIPO> Pila<TIPO>::Pila(int initialSize)
{
  size=initialSize;
  defaultGrowthSize=initialSize;
  marker=0;
  contenuto=new TIPO[initialSize];
}
```

Utilizzando questa versione del codice, dobbiamo fare attenzione che la specifica del parametro `TIPO` appare due volte:

1. per specificare che il template usa il tipo `TIPO` come suo parametro;
2. per specificare che la Pila di cui definisco il costruttore è parametrizzata su `TIPO`.

Gli altri metodi, utilizzando la stessa sintassi, diventeranno:

```
template <class TIPO> Pila< TIPO>::~Pila(){...}
template <class T> void Pila<T>::cresci(int increment){...}
template <class T> void Pila< T >::aggiungi(T k){...}
template <class T > T Pila< T >::estrai(){...}
template <class X > Pila< X >::Pila(Pila< X > & from){...}
template <class X > Pila< X >&
   Pila< X >::operator=(const Pila< X >& dx){...}
```

Notiamo che, così come avviene con i nomi dei parametri delle funzioni, il nome usato per il parametro è puramente formale. Potremo quindi utilizzare, come negli esempi qui sopra, diversi nomi per denotarlo[3].
Nota: La funzione dichiarata `friend` *deve seguire la sintassi interna, e non può essere scritta con sintassi esterna.*

4.4 Ancora una cosa...

E se volessimo usare la nostra Pila per immagazzinare stringhe, sotto forma di vettori di caratteri? Potremmo tentare di dichiarare `Pila <char*>a;`. Il programma *NON SAREBBE CORRETTO*, e girerebbe per puro caso. Il problema sta nel fatto (visto più volte) che dovremmo dichiarare le stringhe come `char*`, ma la pila che abbiamo costruito è fatta per contenere dati, non *reference* ai dati.
Se invece che usare vettori di caratteri usassimo la classe `String` che abbiamo costruito in precedenza, tutto andrebbe a buon fine.

4.5 ...ma questo è solo l'inizio

Abbiamo qui grattato la superficie dell'uso dei templates. Molte altre cose vi sarebbero da aggiungere[4], ma per questi argomenti si rimanda ad altre letture (ad esempio il volume 2 di *Thinking in C++*).
Però non siamo proprio veramente contenti: siamo sì riusciti ad ottenere una Pila adattabile al tipo di dato desiderato, ma ancora non possiamo inserire dati di tipo diverso nella stessa pila...

[3] A patto che nomi diversi stiano in funzioni diverse, ovviamente
[4] templates con parametri costanti, templates con parametri multipli, il costrutto `typename`, templates di templates, istanziazione dei templates

Capitolo 5

Ereditarietà

5.1 Object Based e Object Oriented Programming

Un linguaggio di programmazione che sia *object based*[1] dà la possibilità di definire oggetti, ma è diverso da un linguaggio *object oriented*. La differenza sta nel fatto che il secondo tipo di linguaggio fornisce agli oggetti anche proprietà di *ereditarietà*, cioè la capacità di usare il codice di una classe (*superclasse* o classe padre) in un'altra classe (*sottoclasse* o classe derivata).
Questa proprietà è importante per due motivi fondamentali:

- favorisce il riutilizzo del codice;
- facilita il mantenimento del codice.

Esempio:

```
// CLASSE PADRE
class Punto
{
 public:
   int x; int y;
   void muovi(int dx, int dy)
   {
     x+=dx; y+=dy;
   }
};
// CLASSE FIGLIA
enum Color{red, orange, yellow, green};
class PuntoColorato:Punto
{
 public:
   Color colore;
   void setcolor(Color c) {colore = c;}
};
```

[1] ad esempio Visual Basic

Per sottolineare il fatto che una classe è figlia di un'altra si utilizza la sintassi

```
class ClasseFiglia:ClassePadre
```

Notiamo che le istanze proprie della classe padre diventano anche propire della classe figlia. Si dice che vengono *ereditate*. Allo stesso modo si comporteranno anche i metodi.

```
int main()
{
  PuntoColorato k;
  k.x = 3; k.y = 5; k.colore = red; // istanze ereditate
  k.muovi(2,7);                     // metodo ereditato
  k.setcolor(orange);
  cout << k.x << ‘‘ ’’ << k.y << ‘‘ ’’ << k.colore << endl;
}
```

Se però andiamo a compilare questo programma, il compilatore dà errore, dicendo che non è possibile accedere ai dati della superclasse. Questo perchè il linguaggio protegge automaticamente le variabili di istanza e i metodi delle classi padri. Per permettere alla sottoclasse di accedere ai campi della classe padre è necessario aggiungere la parola chiave `public` subito dopo i due punti nella definizione della classe figlia:

```
class sottoclasse:  public superclass
```

Bisogna però chiarire che in questo modo è possibile raggiungere soltanto i campi *pubblici* della superclasse, mentre non è comunque possibile raggiungere quelli privati. Per farlo è necessario modificare la classe padre, inserendo al posto della parola chiave `private` la parola `protected`. In tal modo i dati saranno

- visibili dentro la classe;
- visibili ai figli di ogni livello;
- invisibili all'esterno.

Esiste una sintassi particolare per potersi riferire alle variabili di istanza della superclasse. Si utilizza l'*operatore di risoluzione di scope*, che vale sia per le istanze che per i metodi su cui è stato fatto overriding (vedi sezione 5.3): si scriverà ad esempio

```
k.Punto::muovi(2,7);
```

5.2 Costruttori e distruttori

Supponendo di aver definito costruttore e distruttore sia per la classe padre che per quella figlia, con il solito messaggio di debug, compilando ed eseguendo il seguente programma

```
int main()
{
  PuntoColorato k;
}
```

si avrà il seguente output

```
creo Punto
creo PuntoColorato
distruggo PuntoColorato
distruggo Punto
```

La sequenza delle operazioni compiute è quindi

- chiamata *implicita* del costruttore della superclasse, perché ne ho bisogno per la costruzione delle variabili proprie della superclasse;
- chiamata del costruttore della sottoclasse, per aggiungere le variabili della sottoclasse.

Una osservazione importante: il costruttore della superclasse viene chiamato *senza alcun parametro*. L'eventuale parametro viene applicato *soltanto alla figlia*.
Il contrario avviene nella chiamata al distruttore:

- prima viene chiamato il distruttore della sottoclasse, per eliminari i dati interni;
- poi viene chiamato il distruttore della classe padre.

5.3 Method overloading e overriding

Anche per le classi derivate si può parlare di overloading dei metodi:

```
class A{
  ...
  void f();
};

class B: public A{
  ...
  void f(int);
};

main(){
  A varA;
  varA.f();
  B varB;
  varB.f();
  varB.f(3);
}
```

Le ultime due righe di codice chiamano due metodi diversi: la prima chiama il metodo della superclasse, mentre la seconda quello della sottoclasse. Il compilatore infatti si accorge che la firma `f(void)` non esiste all'interno della classe figlia e quindi va a cercarla nella superclass.

Se invece anche nella classe `B` definisco un metodo `f(void)`, in questo caso si parla di *method overriding*, cioè di una riscrittura del metodo. Per chiamare il metodo della classe padre quindi dovrò scrivere `varB.A::f()`.
Perciò è possibile definire metodi con le stesso nome, che però fanno cose diverse a seconda del *tipo* che viene passato come parametro.

5.4 Ereditarietà e templates

Con i templates eravamo riusciti a creare pile aventi tipi di dato diversi. Rimaneva però il problema della gestione di pile con dati diversi all'interno dello stesso programma. Una soluzione possibile è data dall'ereditarietà: è sufficiente costruire una pila contenente un dato di tipo generico `Object`, che verrà sottoclassato in seguito, inserendo tutti i tipi di dato che servono.
Però questa sintassi accetta come contenuto della pila soltanto entità di tipo classe. E se fosse necessario inserire un `int`?
La soluzione è semplice: basta costruire una classe `Intwrapper`, che conterrà semplicemente un intero e i metodi getter e setter per accedervi. In questo modo è possibile definire dei *contenitori per tipi di dato primitivi*.
In `C++` sorgono però dei problemi: se ad esempio si acquistano delle classi contenute all'interno di una libreria, non è possibile far diventare tali classi delle figlie della nostra classe generale `Object`, in quanto dovremmo scrivere

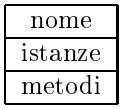
```
NomeClasse :  public Object
```

Ma non avendo a disposizione il codice sorgente, non è possibile[2].
Quindi, in casi semplici come quello presentato... usiamo i `templates`!!

5.5 Un po' di UML

UML, cioè *Unified Modelling Language*, è un linguaggio che permette di progettare programmi ad oggetti in modo compatto. È un modo per avvicinarsi alla nuova concezione della programmazione introdotta con gli oggetti, che mette il fuoco non sulle funzioni da implementare, ma sulla struttura dati su cui queste verranno utilizzate.

Una classe si può rappresentare anche come un diagramma



Ogni oggetto così creato è associabile ad altri oggetti, attraverso due tipi fondamentali di relazioni:

- una relazione di tipo *has-a*, in cui un oggetto viene messo in relazione con suoi componenti o con altri oggetti ad esso correlati: ad esempio la classe Veicolo con la classe Ruota, oppure la classe Studente con la classe Esame;
- una relazione di tipo *is-a*, in cui un oggetto viene messo in relazione con altri, che sono sue *specializzazioni*, o dualmente più oggetti vengono messi

[2] In `Java` si può fare, perché ogni classe eredita per default da una classe `Object`

in relazione con uno solo, che indica una loro *generalizzazione*: ad esempio la classe Veicolo è generalizzazione per Automobile e anche per Motociclo.

A volte si trovano diagrammi in UML aventi diversi colori: viene utilizzato un colore per evidenziare la struttura *logica* del programma e uno per evidenziare quella fisica.
Una precisazione: si possono trovare diagrammi in cui il rombo che indica la relazione di appartenenza è colorato[3]. Questo significa che le due classi sono vincolate una all'altra da una relazione di "Contains": ad esempio una classe `Porta` è vincolata ad una classe `Aula`, poiché non esistono aule senza porte, ma lo stesso legame non c'è fra `Studente` e `Aula`, perché esistono aule vuote, ovvero senza studenti.
Ci sono infine alcune configurazioni, dette *pattern*, che vengono utilizzate frequentemente negli schemi di struttura, in quanto sono costrutti molto potenti e quindi è utile conoscerli. Due esempi sono mostrati in figura 5.1.
Il diagramma sulla destra indica che esistono istanze della classe considerata che sono in relazione con altre istanze della stessa classe (*relazione ricorsiva*).
Il diagramma sulla sinistra mostra una doppia relazione fra due classi distinte: l'interpretazione di questo pattern è la seguente: la classe `Contenitore` è una sottoclasse di `Componente`, ma a sua volta ogni istanza della seconda classe ha un'istanza della prima che la contiene.

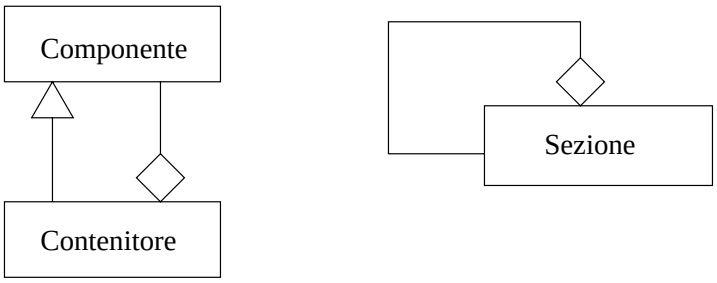


Figura 5.1: Pattern UML

5.5.1 Applicazione

Scrivere un programma che sia in grado di gestire la configurazione di una scacchiera e le mosse che l'utente propone, scartando quelle non corrette.
La struttura dati è visibile nella figura 5.2 Notiamo che sarà preferibile utilizzare puntatori ad oggetto, anziché istanze di oggetti, in quanto il contenuto di ogni casella continuerà a variare. Inoltre avremo molti vantaggi:

- saranno più chiare le relazioni fra le classi;
- per ogni pezzo avrò un unica istanza, che invece si duplicherebbe se non utilizzassimo i puntatori (i passaggi per copia producono copie dei parametri);

[3] vedi figura 5.5, pag. 76

- l'esecuzione sarà più veloce, in quanto si lavorerà con puntatori invece che con strutture complesse.

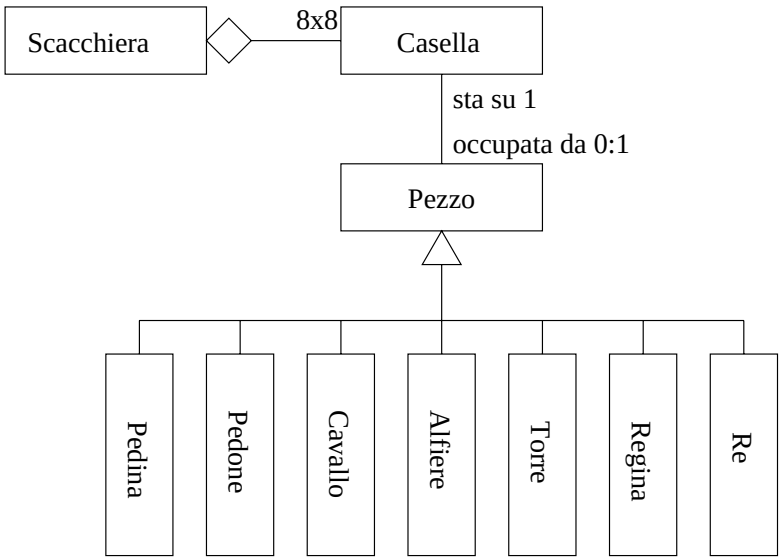


Figura 5.2: UML scacchi

Ma avremo un problema: se inserissimo nella classe `Pezzo` un metodo setter, che inizializza un pezzo, non potremmo utilizzare poi lo stesso metodo ad esempio con la classe `Pedone`, in quanto non è delle stesso tipo. Anche nel caso dei puntatori avremmo lo stesso problema, anche se in maniera minore, in quanto si possono sempre fare assegnazioni fra puntatori.
Questo è un problema di *Casting*.

5.6 Static binding e dynamic binding

Solitamente il legame tra una variabile e il suo tipo viene fatto dal compilatore a comila-time, ovvero *indipendentemente* da cosa fa il programma.

```
Punto *k = new PuntoColorato;
k->muovi(3,2);
```

Sarebbe bello poter utilizzare il metodo `muovi` della classe figlia. Ma il compilatore riconosce che la variabile `k` è di tipo `Punto` e quindi chiama il metodo appartenente alla stessa classe. Questo viene detto *static binding*.
Se invece si vuole forzare la chiamata al metodo della classe derivata, si fa quello che viene definito *dynamic binding*, che ricorda al compilatore di controllare *a run-time* il tipo di dato passato a parametro e di chiamare di conseguenza il metodo appropriato. Ovviamente questo comporta una diminuzione della velocità di esecuzione.

Un esempio in cui si utilizza il dynamic binding è il seguente:

```
Punto *k = new PuntoColorato;
delete k;
```

In questo caso vengono chiamati i due costruttori (quello della classe padre e quello della figlia), ma *soltanto un distruttore*, ovvero quello relativo alla classe padre. In questo ambito non si ha memory leak, in quanto le variabili proprie della classe figlia sono tutte di tipo primitivo e quindi vengono distrutte automaticamente al termine della funzione. In generale però posso avere dei problemi di deallocazione.
Per risolvere il problema si utilizza il binding dinamico e si aggiunge la parola chiave `virtual` davanti alla dichiarazione del distruttore, in modo che il compilatore si ricordi che a run-time deve andare a vedere che tipo di istanza viene passata e chiami anche il distruttore della classe `PuntoColorato`.
Nota importantissima: il dynamic binding viene applicato *soltanto* ai puntatori ad oggetto. Non sarà possibile quindi applicarlo alle istanze di una determinata classe, per le quali si utilizzerà sempre esclusivamente il binding statico.

5.7 Upcasting e downcasting

Il dynamic binding è utile soprattutto quando si fanno assegnazioni fra puntatori di tipo non omogeneo:

```
Punto *k = new PuntoColorato;
```

Si tratta di un'operazione di *upcasting*, ovvero un'assegnazione ad un puntatore ad oggetto di un'istanza di tipo derivato. Questa sintassi è corretta e quindi viene *sempre* permessa. In questo caso infatti, viene allocato qualcosa di più grande e quindi tutti i campi necessarri alla buona definizione di un'istanza della superclass sono presenti.
Notiamo però che non sarebbe possibile scrivere `k->colore`, perché `k` è di tipo `Punto` e quindi non ha il campo `colore`.
Non è invece lecito assegnare ad un puntatore ad una classe una istanza di una sua superclasse, cioè

```
PuntoColorato *w = new Punto;
```

In questo caso infatti il campo `colore` esiste, ma non è stato allocato. Quindi se vado a leggere troverò porcheria!
Quindi in generale avremo che

Non posso associare ad un puntatore di una classe derivata un'istanza della sua superclass

Questo è un problema di *type checking*, ovvero di controllo di tipo. Infatti in una classe posso metterci qualcosa che è dichiarato come "is-a" (ovvero come specializzazione) della classe stessa, mentre non posso fare il contrario.

Inoltre, se voglio referenziare un metodo della classe derivata, devo stare attento che:

- il metodo esista nella superclass;
- il metodo sia dichiarato `virtual` nella superclass.

Un tipo particolare di metodo virtuale è il metodo *virtuale puro* o metodo *astratto*, che viene scritto come

```
virtual void muovi()=0;
```

Questa soluzione è utile quando con lo stesso nome si vogliono indicare metodi di classi diverse (tutti con la stessa firma!!) che fanno cose diverse. Ad esempio, negli scacchi avremo per ogni figlia della classe `Pezzo` un metodo `controllaMossa()`, ma in ogni classe verrà implementato in modo diverso. Per poter fare questo, dichiareremo all'interno della classe padre un metodo virtuale puro, avente lo stesso nome.
Attenzione: se in una classe ho un metodo virtuale puro, non potrò avere istanze (ovvero oggetti) di quella classe, ma soltanto delle sue sottoclassi. Inoltre ogni sottoclasse *dovrà avere* un overriding del metodo dicharato virtuale puro. Le classi che contengono metodi virtuali puri vengono dette *classi astratte*, e servono soltanto come *generalizzazione* di altre classi.
Notiamo che non sarebbe possibile definire un metodo

```
virtual inline int value()...
```

Infatti le due parole chiave sono in conflitto fra di loro:

`inline` dice al preprocessore di prendere il corpo della funzione e inserirlo al posto della sua chiamata;

`virtual` dice al compilatore di controllare a run-time quale parametro verrà passato alla funzione.

Osserviamo infine che è buona norma dichiarare virtuali tutti i distruttori delle classi, in modo da favorire la riusabilità del codice, in caso di ereditarietà. Si potrebbe definire tutto virtuale, ma questo porterebbe un notevole allungamento nei tempi di esecuzione nel programma (vedi sez. 2.14, pag. 27).

5.8 Classi interne

Invece di avere classi e funzione `main` separati, è possibile inserire l'implementazione delle classi direttamente all'interno della funzione principale. In questo caso si parla di *classi interne*. La differenza sostanziale sta nello scope che le classi hanno. Infatti, se queste vengono definite all'*esterno* del `main`, vengono trattate come *globali* e quindi sono visibili ovunque all'interno del programma. Se invece le classi vengono dichiarate all'*interno* della funzione principale, avranno scope *locale*, saranno quindi visibili solo all'interno del main. Non sarà quindi possibile utilizzare tali classi all'interno di una funzione esterna.

vantaggio aumento la leggibilità e la manutenzione del codice;

svantaggio le classi non sono visibili dall'esterno.

Attenzione: Supponiamo di avere una classe `A`, nella quale sia definita una classe interna e supponiamo inoltre che ci sia anche una classe figlia di `A`. Un esempio di codice potrebbe essere il seguente:

```
class A
{
  ...
  class A1
  {
    ...
  };
};
class B: public A
{
  ...
};
```

Si avranno le seguenti relazioni:

- la classe `A` non ha accesso alle variabili della propria classe interna (nel rispetto delle regole di scope delle variabili);
- la classe interna ha accesso alle variabili della classe in cui è contenuta;
- la classe figlia non ha accesso alle variabili della classe interna al padre.

Un programma di grandi dimensioni è formato da tante classi, aventi molte relazioni fra di loro. È quindi possibile accorpare le classi che hanno elementi in comune o forti relazioni fra di loro in un unico blocco. A tale proposito in `C++` si usa la parola chiave `namespace`, che raggruppa assieme pezzi di codice omogenei[4].

5.9 Static e dynamic cast

Abbiamo visto che è possibile dichiarare un oggetto, inserendo al suo interno un oggetto appartenente ad una sua sottoclasse, e che questo è sempre permesso, grazie alle relazioni di ereditarietà viste in precedenza. È quindi possibile operare delle *conversioni di tipo*, ma con alcune restrizioni:

upcasting non dà alcun problema upcasting. Risulta anzi comodo se è necessario associare istanze di classi figlie ad una certa entità. È inoltre un'operazione univoca, in quanto da una classe figlia si può univocamente risalire alla classe padre[5], cosa che invece non si può sempre fare con certezza nel caso del downcasting, che quindi viene vietato.

downcasting crea alcuni problemi, poiché le istanze di una sottoclasse dipendono sia da se stesse che dalla superclass. Risulta quindi difficile disambiguare. Inoltre all'interno delle classi figlie posso avere variabili in più rispetto alla classe padre. Operando attraverso downcasting, tali variabili *non* verranno inizializzate, creando poi errori.

[4] in Java si usa la parola chiave `package`.

[5] ... a meno di ereditarietà multipla, vedi sez. 5.10, pagina 70

È però possibile fare anche dei cast *espliciti*:

```
A *p1;
p = (B) p1;
```

In questo modo informiamo il compilatore noi sappiamo che questa conversione si può fare senza problemi e che ci impegnamo ad usare quel puntatore in modo corretto.
Le conversioni di tipo esplicito sono comunque *pericolose*. Infatti, se non ci fosse alcun vincolo sulle possibilità di cast, potremmmo operare downcasting anche in modo sconsiderato, senza che il compilatore ci avverta dei possibili errori.
Ad esempio, se avessimo

```
class Forma {...};
class Cerchio: public Forma {...};
class Quadrato: public Forma {...};
class Altro {...}
```

Sarebbe possibile convertire un oggetto di classe `Forma` in un cerchio o un quadrato, ma questo porterebbe agli errori visti in precedenza. Sarebbe bello che il compilatore controllasse se il tipo passato al cast è *compatibile* con il puntatore e che segnalasse eventuali errori. Il linguaggio ci viene in aiuto e fornisce due funzioni particolari: `static_cast` e `dynamic_cast`.

5.9.1 `static_cast`

Viene valutato a compile time, quindi indipendentemente dal tipo di dato che viene passato come parametro. Abbiamo già visto che questo tipo di cast non è ancora perfetto, perchè in fase di compilazione a volte non è possibile decidere come comportarsi[6].
Nel caso presentato, sarà possibile compiere un cast a `Quadrato`, ma non ad un puntatore ad `Altro`. Questa funzione permette quindi il cast *soltanto* all'interno della propria gerarchia. Non sarà perciò possibile eseguire una conversione fra due classi non legate fra loro da una relazione di parentela, a quelunque livello essa sia.
Avremo ad esempio

```
Cerchio c;
Forma *s = &c; // Upcast normale

Cerchio *cp = NULL;
Altro *sp = NULL;

cp = static_cast<Cerchio *>(s); // permesso
sp = static_cast<Altro *>(s); // non permesso
```

[6] vedi ad esempio il problema degli scacchi, sez. 5.5.1, pag. 63

5.9.2 dynamic_cast

Si valuta a run time, quindi dipende dal tipo di dato che viene passato come parametro. Avremo ad esempio

```
Cerchio c;
Forma *s = &c; // Upcast normale
Cerchio *cp = NULL;
Altro *sp = NULL;

cp = dynamic_cast<Cerchio *>(s);
sp = dynamic_cast<Altro *>(s);
```

Compilando il codice, viene segnalato un errore, e precisamente

```
Type 'Forma' is not a defined class with virtual functions
```

Il compilatore si accorge che la classe padre non ha metodi virtuali e quindi non permette di utilizzare il cast dinamico. Infatti il compilatore cerca di ottimizzare il tempo di esecuzione. Poiché la classe padre non ha metodi virtuali, ogni sua istanza è completamente determinata a compile time. Questa funzione quindi opera il cast dinamico all'interno della gerarchia, come nel caso del cast statico, ma in più lo permette *soltanto* se la classe passata come parametro ha almeno un metodo virtuale (basta anche soltanto il distruttore). In questo modo vengono compiute tutte e sole le conversioni di tipo che "hanno senso".

Una applicazione del cast dinamico si ha ancora una volta con l'esercizio degli scacchi, e precisamente nella dama. Supponendo di voler mangiare un pezzo con una pedina, bisogna assicurarsi che questo sia anch'esso una pedina[7]. Una soluzione potrebbe essere quella di dotare la classe `Pezzo` di una variabile booleana che controlli se esso sia o meno una pedina, ma questo complicherebbe la situazione, in quanto ci sarebbero variabili superflue nella classe padre[8]. Dovrei quindi sottoclassare `Pezzo` in `PezzoDiDama` e `PezzoDiScacchi`, complicando la gerarchia e inserendo poi nella nuova sottoclasse la variabile booleana che serve. La soluzione migliore sta nell'utilizzo del dynamic cast: basta tentare un cast dinamico sul pezzo che si intende mangiare, convertendolo a `Pedina`. Solo se la conversione ha successo, è possibile mangiare il pezzo, perché significa che esso è proprio una pedina.

[7] ricordiamo che una pedina semplice non può mangiare una dama!
[8] negli scacchi infatti questa variabile non servirebbe

5.10 Multiple Inheritance

A volte si rende necessario far derivare una certa classe da più di una classe padre. Un esempio potrebbe essere il seguente: supponiamo di aver acquistato un set di classi, che abbia una propria gerarchia interna (ad esempio per la gestione di una contabilità). Supponiamo che ad un certo punto della gerarchia ci sia una certa classe di nome `Fattura`. Vorremmo utilizzare il codice già costruito per le liste linkate per gestire la nostra contabilità, grazie a queste classi che abbiamo comprato. Vorremmo utilizzare la classe `Payload` come padre per la classe acquistata `Fattura`. Poiché non disponiamo del sorgente di tale classe, dobbiamo aggirare il problema: definiamo quindi una nuova classe `MyFattura`, che accodiamo alla gerarchia, come figlia di `Fattura`. La rendiamo inoltre figlia anche di `Payload`, in modo che possano utilizzare sia i metodi della prima classe (e quindi tutte le classi che abbiamo comprato), che i metodi della seconda (e quindi il codice delle liste linkate).

Con questo semplice esempio abbiamo visto l'utilizzo fondamentale della eredità multipla, ovvero la possibilità di accedere ai metodi di classi non fanno parte della propria gerarchia. La sintassi è la seguente:

```
class A1
{
public:
  A1() {cout << ``costruttore di A1'' << endl;}
  void f1() {cout << ``f1 di A1'' << endl;}
  void f2() {cout << ``f2 di A1'' << endl;}
};
class A2
{
public:
  A2() {cout << ``costruttore di A2'' << endl;}
  void f1() {cout << ``f1 di A2'' << endl;}
  void f2() {cout << ``f2 di A2'' << endl;}
};
class A3: public A1, public A2 {...}; // due padri!
```

Notiamo che l'ordine con cui le classi padre sono riportate nella definizione della classe figlia *è importante*, in quanto determina i metodi che verranno invocati in seguito.
Ad esempio, consideriamo il seguente `main`:

```
int main()
{
  A1 *a1 = new A1();
  a1 -> f1();         // chiama f1 di A1
  A2 *a2 = new A2();
  a2 -> f2();         // chiama f2 di A2
  A3 *a3 = new A3();
  a3 -> f1();         // chiama f1 di A1
  a3 -> f2();         // chiama f2 di A1
}
```

Come si può notare dai commenti nel codice, il compilatore chiama i metodi con la seguente politica:

- prima di tutto cerca nella classe corrente;
- poi cerca nella *prima* classe padre dichiarata;
- poi cerca nelle altre...

Inoltre, come nel caso dell'ereditarietà semplice, alla creazione di una nuova istanza della classe figlia, verranno chiamati tutti i costruttori delle classi padre, *nell'ordine in cui sono dichiarate.*
Se però volessimo chiamare la funzione `f2` di `A2`, dovremmo utilizzare l'operatore di risoluzione di scope "`::`" e quindi scriveremo `a3.A2::f2()`. In questo modo è possibile raggiungere qualsiasi metodo appartenente ad una classe inserita nella gerarchia.

Consideriamo invece il seguente caso:

```
class A
{
public:
  A() {cout << ``costruttore di A'' << endl;}
  void f() {cout << ``f di A'' << endl;}
};
class B: public A
{
public:
  B() {cout << ``costruttore di B'' << endl;}
  void f() {cout << ``f di B'' << endl;}
};
class C
{
public:
  C() {cout << ``costruttore di C'' << endl;}
  void f() {cout << ``f di C'' << endl;}
};

int main()
{
  B b;
  b.f();     // chiama f di B
  b.A::f();  // chiama f di A
  b.C::f();  // errore!!!
  return 0;
}
```

L'ultima chiamata dà errore, perché la classe `C` non fa parte della gerarchia della classe chiamante. La soluzione è utilizzare la parola chiave `static`. Ripassiamone un attimo l'utilizzo:

- una variabile *locale* dichiarata statica è visibile ad ogni chiamata della funzione in cui si trova e sopravvive al termine della funzione stessa;
- una variabile *globale* dichiarata statica è visibile soltanto nel file in cui è dichiarata;
- una variabile di *istanza* dichiarata statica viene utilizzata per far interagire gli oggetti della stessa classe fra di loro, ne esiste una sola per tutti gli oggetti della classe. Inoltre esiste *indipendentemente* dall'esistenza di istanze della classe

A questi utilizzi se ne aggiunge ora un altro: un *metodo* che venga dichiarato statico è utilizzabile indipendentemente dall'esistenza di istanze della classe di appartenenza. Inoltre lo si può vedere come facente parte di una libreria di funzioni. A volte infatti si costruiscono set di classi che contengono soltanto metodi. Tali classi non avranno quindi un metodo costruttore, in quanto di esse non verranno create istanze, ma ne verranno utilizzati soltanto i metodi.
Nota: i metodi dichiarati statici possono agire *soltanto* su variabili statiche[9].
Quindi nel codice precedente sarà possibile scrivere `C::f()` *SE E SOLO SE* il metodo `f()` è dichiarato statico all'interno della classe `C`.

Consideriamo ora la figura 5.3. Essa corrisponde al seguente codice:

```
class A {
  int x;
  void f() {cout << ''f'' << endl;}
};
class A1: public A {...};
class A2: public A {...};
class A3: public A1, public A2 {...};
```

Se dalla classe `A3` volessimo chiamare il metodo `f()`, avremmo un'ambiguità, poiché in questa situazione in memoria si hanno due copie della classe `A`, una per ogni classe figlia `A1` ed `A2`, le quali a loro volta stanno nella memoria allocata per `A3`. La chiamata ad un metodo della classe padre crea quindi un problema, in quanto il compilatore non sa a quale delle due copie accedere.
Anche in questo caso esiste la cura: basta inserire la parola chiave `virtual`[10] davanti alla dichiarazione di parentela della classe `A`:

```
class A1: virtual public A {...};
class A2: virtual public A {...};
```

In questo modo in memoria verrà creata *una sola* copia della classe padre e non ci saranno ambiguità.
Nota: la chiamata ai metodi della classe `A` diventa ambigua solo se viene effettuata da classi "nipoti". Non lo è quindi se effettuata da classi che sono "figlie dirette di `A`". Infine notiamo che quando viene creata un'istanza della classe `A3`, in presenza del `virtual`, il costruttore della classe `A` verrà chiamato *soltanto una volta*!

[9] in quanto entrambi sono indipendenti da istanze di una classe
[10] Attenzione! Non è lo stesso uso di `virtual` che avevamo visto in precedenza!

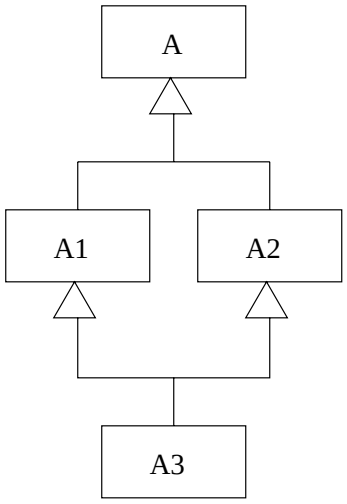


Figura 5.3: Ereditarietà multipla

In assenza della parola chiave invece in memoria si creano *2 copie della classe* `A`, come si può notare dalla figura 5.4 e da questo semplice codice:

```
A3 *a3 = new A3();
a3 -> f1(3);
a3 -> f2(6);
cout << a3 -> A2::x << endl;
cout << a3 -> A1::x << endl;
```

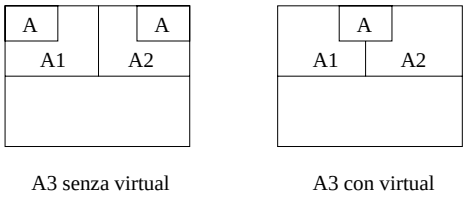


Figura 5.4: Ereditarietà multipla 2

Eseguendo questo codice si nota che i due output danno due risultati differenti, pur accedendo alla stessa variabile di istanza. Quindi mettendo la parola chiave `virtual` si evitano le copie, ma **attenzione:** è necessario metterla *in tutte le classi che ereditano da* `A`.
Come ultima cosa, ricordiamo che se la classe padre ha un proprio metodo ma non viene dichiarata virtuale, le classi appartenenti alla stessa gerarchia che non sono sue figlie dirette *non possono chiamare quel metodo.*

5.10.1 Nota sui costruttori

In caso di ereditarietà abbiamo visto che i costruttori vengono chiamati a cascata. Ma se dovessimo chiamare il coy constructor?

```
class A
{
  A(int i) {cout << ``costruttore di A(int)'' << endl;}
  A(A & a) {cout << ``copy constructor di A'' << endl;}
};
...
A a=1; // chiama A(int)
A b=a; // chiama A(A & a)
```

In precedenza infatti avevamo visto che in caso di ereditarietà, nella superclass veniva chiamato il costruttore *vuoto*!
Supponiamo di avere la situazione mostrata in figura 5.3 e che inoltre la classe `A` contenga puntatori a strutture esterne[11]. Chiamando il copy constructor della classe `A3`, abbiamo bisogno di chiamarlo anche per la classe padre. Ma il compilatore chiama il costruttore vuoto, creando gli errori che abbiamo visto. Notiamo però che commentando il copy constructor che abbiamo creato noi nella classe figlia, viene chiamato il copy constructor della classe padre! Questo quindi significa che

Il copy constructor di default chiama a catena copy constructor delle classi padre, per qualsiasi livello.

Ma se mi serve il copy constructor anche nella classe figlia, torno al problema di prima, che è quindi dovuto ad una scelta degli inventori del `C++`.
Il motivo di questa scelta sta nel fatto che se eredito da una classe di cui non ho il codice sorgente, non so come scrivere il copy constructor per tale classe, e quindi non so farlo nemmeno per le classi figlie. Se invece fosse stata implementata la possibilità di chiamare automaticamente il copy constructor, ci sarebbero stati dei problemi di perdita di dati: supponiamo che la classe padre contenga un puntatore ad un vettore di caratteri, di lunghezza n e che il copy constructor della classe figlia possa aumentare tale dimensione ad m. Le operazioni che verranno compiute saranno:

- copia del vettore lungo n su quello lungo m;
- eliminazione del vecchio vettore;
- aggiornamento del puntatore al vettore.

Se però viene chiamato il copy constructor della classe padre, queste modifiche verranno ignorate e la dimensione ritornerà n, con possibile perdita di dati. Per questo si è deciso che il sistema chiami il copy constructor di default per le classi padre, perché a priori non sa come io ho implementato il mio copy constructor. Se però a me servono copy constructor per deep copy sia nella classe figlia che nella classe padre, come posso fare?

[11] la classe necessita quindi di un copy constructor per deep copy

Basta usare l'operatore di ereditarietà:

```
B(int i):A(i);
B(B & a):A(a);
```

Questa sintassi chiama il costruttore della classe figlia, il quale a sua volta chiama il costruttore della classe padre *specificato* nel codice[12].
Questo risolve tutti i problemi che avevamo.

5.11 Delegation

Nei linguaggi object based[13] non esiste il concetto di ereditarietà mutipla. Come sarà quindi possibile definire i concetti visti nelle precedenti sezioni? La soluzione è data dalla *delegation*. In pratica non si fa altro che sostituire la relazione di *is-a* con una di *contains*: la classe figlia, anziché dipendere dalla classe padre, ha con essa una relazione, che mostra che la prima è *parte* della seconda. In UML si rappresenta con il rombo pieno (vedi figura 5.5), mentre il codice corrispondente sarà

```
class A {...};
class B {
  A a;
};
```

Si tratta quindi la classe padre come un'istanza della classe figlia, cosa che *non* si può fare usando l'ereditarietà, perché causa dei loop.
Ma se la classe padre ha un metodo `setX`, la classe figlia non potrà usarlo direttamente, poiché manca la relazione di ereditarietà: bisogna quindi modificare la sintassi, utilizzando ancora una volta una specie di macro[14]:

```
class A
{
  ...
};

class B
{
  A delegate;
  void setX(int k) {delegate.setX(k);}
};
```

In questo modo si dice al compilatore che ogni volta che trova la funzioe `setX`, invocata dalla classe `B`, lui la sostituisca con una chiamata alla funzione corrispondente della classe padre. In generale quindi per ogni metodo della classe padre, scriveremo un metodo *identico* per la classe figlia, che faccia "rimbalzare"

[12] ovviamente è possibile scrivere codice misto, ad esempio `B(B & a):A(i)...`
[13] vedi sezione 5.1, pag. 59
[14] come per `inline`

la chiamata alla classe delegata.
Ma se ho un metodo `setX` anche nella classe `B`? Avrò semplicemente ottenuto un overriding di un metodo, e la chiamata a tale metodo rispetterà le regole di risoluzione viste per il method overriding: se esiste un metodo nella classe con tale nome, verrà chiamato quello, altrimenti si risalirà nella gerarchia oppure si controlleranno tutte le classe delegate.
Quanto presentato nell'ultima sezione è soltanto un esempio di come si possano aggiungere o sostituire costrutti con forme ad essi equivalenti, con relativamente poco supporto da parte del linguaggio.

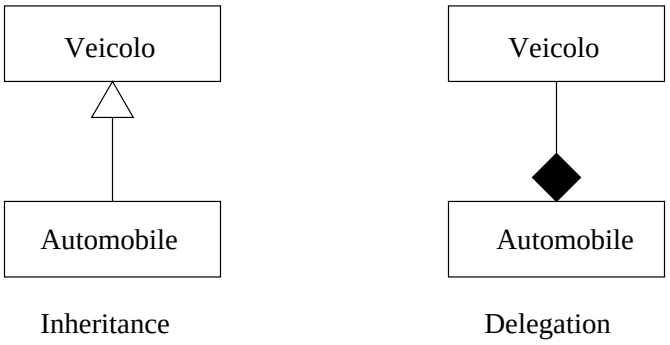


Figura 5.5: Esempio di delegation

Appendice A

All'interno del C++

A.1 Il comando `return 0;`

Questo comando, inserito nella main funztion, permette al sistema operativo di sapere se il programma è andato a buon fine o se ci sono stati problemi. Tipicamente si fa ritornare zero se non ci sono stati problemi, un valore diverso da zero altrimenti. Esiste anche un modo per vedere quale valore è stato ritornato all'ambiente: dalla shell, dopo aver fatto girare il programma, digitare `echo $?`. Apparirà il valore ritornato.

A.2 Assegnazione condizionale

È possibile assegnare il valore ad una variabile in base al risultato di una certa condizione. Il codice può essere di questo tipo:

```
int a, b, c;
if(b > c)
  a = b;
else
  a = c;
```

Ma esiste anche una forma più compatta:

```
a = (b > c) ? b : c;
```

In pratica (poiché l'assegnazione procede da destra verso sinistra) le operazioni sono:

- esecuzione del controllo;
- se risulta vero, si assegna il valore posto a destra del punto di domanda;
- se risulta falso, si assegna il valore posto dopo i due punti

È quindi in tutto e per tutto equivalente alla forma precedente. Ma la prima è preferibile, in quanto è più chiara, anche se meno compatta.

A.3 Perchè solo queste primitive?

I programmatori del linguaggio C++ hanno deciso di predisporre soltanto un numero limitato di primitive, permettendo la costruzione di tipi aggregati (vedi sezione 1.5, pag.5) al programmatore.
Ad esempio sarebbe stata comoda la definizione di una primitiva per il tipo puntatore, ma utilizzare il simbolo asterisco accanto al tipo di dato puntato porta alcuni vantaggi importanti:

- risulta più chiaro il contenuto della variabile;
- è possibile effettuare operazioni di type checking, molto utili in caso di cast dinamici.

Allo stesso modo, non esiste un tipo stringa, il perché... non lo so.
In generale, il tipo del dato serve per:

- sapere quanta memoria viene occupata dalle singole variabili
- sapere come fare le operazioni con quella variabile[1]

A.4 Puntatori a void

A cosa serve un puntatore che non si sa a cosa punta? Beh... serve alla macchina, per le sue buone ragioni interne.
Ad esempio, se si guarda l'indirizzo di una variabile di tipo intero, si noterà che esso è divisibile sempre per 4, mentre quello di una variabile di tipo long sarà divisibile per 8. Ad un puntatore a void verrà assegnato un indirizzo che vada bene per ogni tipo di dato.

A.5 Librerie

Le librerie più utilizzate nella programmazione in C e C++ sono:

- `limit.h, float.h`, funzioni che danno informazioni sui limiti della macchina (grandezze delle word, overflow...);
- `stdio.h`, funzioni standard del C [2];
- `iostream.h`, funzioni standard del C++;
- `math.h`, funzioni matematiche;
- `ctype.h`, funzioni che manipolano caratteri
- `string.h`, funzioni di manipolazione stringhe;
- `time.h`, funzioni temporali;
- `stdlib.h`, funzioni varie.

[1] ogni operazione è definita soltanto per un tipo di dato. Ci sarà quindi una somma per interi, una per reali...

[2] può accadere di dover utilizzare funzioni scritte in linguaggi precedenti a quello in uso. È quindi necessario utilizzare le librerie che contengono le loro definizioni. Si tratta di un problema di **legacy**, ovvero di eredità

A.5.1 Utilizzo delle librerie

Ovvero, come creare una libreria fai-da-te, in pochi passi rapidi ed indolori.
In questo esempio creeremo una libreria geometrica, contenente le funzioni necessarie a disegnare un quadrato e un triangolo, fatti di asterischi.
Utilizzeremo due files:

- square.c, che contiene la funzione per creare il quadrato;
- triangle.c, che contiene la funzione per creare il triangolo;

I passi da compiere sono i seguenti:

1. creare il file `square.c`;
2. `cc -c square.c`, che produce soltanto il file oggetto square.o[3];
3. `ar q libgeo.a square.o`, che aggiunge il file oggetto alla libreria. Se la libreria non esiste, la crea; (`ar` sta per *archive*, `q` sta per *to queue*, cioè accodare[4])
4. creare il file `triangle.c`;
5. creare il file oggetto e accodarlo alla libreria;
6. `ar t libgeo.a`, che fornisce la lista dei file oggetto contenuti nella libreria;
7. `ranlib libgeo.a` permette di leggere il file di libreria partendo da un punto qualsiasi, ovvero di leggere soltanto le parti che interessano[5];
8. creare il file header `geo.h`, che contiene i prototipi delle funzioni di libreria;
9. creare il file in cui serve tale libreria, includendo l'header fra virgolette (`#include``geo.h''`);
10. compilare il file sorgente:

```
g++ prog.C -o prog -L. -lgeo
```

    che dice al compilatore di andare a cercare la libreria con parola chiave "geo" nella directory corrente.

Osserviamo che il file di libreria serve subito al compilatore, in quanto già in fase di controllo di sintassi, deve essere sicuro che il programmatore abbia dato i parametri giusti alle funzioni di libreria, ecc...

[3] D'altra parte non potrebbe andare avanti, perché manca la funzione main()...
[4] che in inglese si legge "chiu", come la lettera "q"
[5] "ranlib" sta per random library, ovvero accesso al file in qualsiasi punto

A.6 Note sulla compilazione

Conviene compilare separatamente ogni classe, inserendo ognuna di esse in files diversi. In questo modo il codice risulta più portabile, in quanto si possono esportare in altri programmi solo le classi che servono, diminuendo lo spazio occupato dall'eseguibile e rispettando inoltre il principio di Parna.
Supponiamo inoltre di aver costruito un programma, contenente due classi `A` e `B`, una figlia dell'altra, implementate in due files separati. Ci saranno diverse dipendenze:

- il file `a.cpp` dipende dal suo header `a.h`;
- il file `b.cpp` dipende dal suo header `b.h`;
- il file `b.cpp` dipende anche dall'header `a.h` e dal file di implementazione `a.cpp`, in quanto la classe `B` è classe derivata da `A`, e ne condivide quindi i metodi;
- il file `main.cpp`, contenente la funzione principale, dipende da entrambi gli header.

Si corre perciò il rischio di includere più volte lo stesso file all'interno del programma. Per evitare questo è bene fare dei controlli, attraverso il costrutto `#ifdef`, ad esempio scriveremo

```
#ifndef A_H
  #define A_H
  #include ``a.h''
#endif
```

Appendice B

Varie ed eventuali

B.1 Accesso in memoria

I tempi di accesso alla memoria disco o al buffer sono molto differenti:

- per accedere al disco occorrono 10^{-2} secondi;
- per accedere al buffer occorrono 10^{-8} secondi.

L'accesso alla memoria temporanea è quindi un milione di volte più rapido di quello alla memoria permanente. Per questo i dati che vengono utilizzati dai programmi vengono caricati nel buffer.

B.2 Come viene caricato un programma?

Per vedere le fasi di costituzione di un programma, vedere la sezione 1.7, pag.7. Quando si esegue un programma, prima di tutto parte una funzione che

- prepara l'ambiente chiamante
- decide le risorse da predisporre

Questa *funzione colla* aggiunge al codice sorgente le varie librerie da includere e chiama la funzione `main()`

B.3 Linguaggio macchina

Il linguaggio macchina è l'unico linguaggio comprensibile al calcolatore ed è composto sostanzialmente dalle seguenti operazioni:

- sposta un dato dalla memoria alla CPU;
- copia un dato in memoria;
- operazioni elementari:
  - modifica di un carattere mediante shift dei suoi bit
  - somma e differenza di bit

- gestione periferiche;

Inoltre ad ogni operazione corrisponde un certo codice. Quindi un programma per essere compreso dal computer deve essere tradotto mediante questi codici.

B.4 Switch opzionali

Ogni eseguibile di ogni ambiente ha le proprie opzioni, che sono attivabili attraverso delle stringhe dette *switch*[1], che vanno inserite dopo il nome dell'eseguibile. In ambiente UNIX sono preceduti dal trattino "-", mentre in WINDOWS dalla barra "/". Per conoscere gli switch corretti da inserire basta consultare le *man pages*, attivabili digitando `man` dal prompt, seguito dal nome del comando.
Per ragioni storiche, la libreria matematica `libmath.a` si inserisce aggiungendo lo switch `-lm`.

B.5 Caratteri

Di solito i caratteri vengono codificati attraverso dei numeri, che vanno da zero a 255, in base alla tabella `ASCII`. I caratteri non sono però tutti stampabili; ne esistono infatti alcuni, detti *caratteri di controllo* che non hanno un output standard (escono cose del tipo ♢, ♡, ♣, ♠, ©, ...). Questi caratteri hanno codice basso, più precisamente da zero a 40.
I caratteri con codice superiore a 128 invece vengono utilizzati per rappresentare i *caratteri nazionali*, ovvero i caratteri come è, à, ì, ò, ù, é, ä, ö, ü, â, ñ...
Per uniformare il codice, si è quindi esteso lo standard `ASCII` all'`UNICODE`, che codifica i caratteri con interi a 16 bit. In tal modo è quindi possibile stampare fino a $2^{16} - 1 = 65535$ caratteri diversi.

B.5.1 Una curiosità

I caratteri maiuscoli hanno codice `ASCII` che parte da 65, mentre i minuscoli partono da 97. Però non sono consecutivi, ovvero i minuscoli non vengono subito dopo i maiuscoli, ma ci sono in mezzo alcuni caratteri come parentesi e apici. Perché? Perché in tal modo la distanza fra una lettera maiuscola e la corrispondente minuscola è 32, ovvero $(00100000)_2$ e questo facilita molto le operazioni con i caratteri, perché per passare da maiuscola a minuscola basta accendere o spegnere un bit.

B.5.2 Caratteri di "A capo"

I caratteri di "a capo" esistenti sono due:

CR carriage return, codice `ASCII` 13;

NL new line, codice `ASCII` 10.

Sono due, perché si rifanno alla vecchia macchina da scrivere, che aveva una leva per andare a capo e una per il ritorno del carrello.
Ogni sistema operativo utilizza uno dei due caratteri:

[1] ovvero interruttori, che decidono se accendere o spegnere l'opzione.

- WINDOWS li utilizza entrambi;
- UNIX utilizza soltanto NL;
- MAC utilizza soltanto CR[2].

Questo significa che il codice `cout << (char) 10` non risulta essere portabile. Per tagliare la testa al toro, quando si scrive un programma, basta scrivere `\n`, oppure `endl`, così ci pensa il compilatore.

B.6 Debug di un programma

Uno dei metodi più utilizzati per *debuggare*, ovvero per trovare gli errori in un file sorgente è il *Bracketing*. Esso consiste nel dividere il codice in blocchi mediante parentesi (in inglese, *bracket*), inserendo per ogni blocco un messaggio di output della forma:

```
cout << ''Sono arrivato qui'' << endl;
```

Eseguendo il programma, appariranno una serie di messaggi di questo tipo. È bene differenziarli tra loro, per capire subito in che punto il programma si blocca.

A volte però questo metodo fallisce: ad esempio consideriamo il codice:

```
int i = 0;
int k = 100;
int z;
...
cout << ''Sono qui'';
...
cout << ''Sono qua'';
z = k / i;
```

L'errore sta palesemente nel fatto che non posso dividere per `i`, in quanto dividerei per zero. Utilizzando il Bracketing così come è riportato nel codice, il programma si fermerà prima dell'assegnazione a `z`, ma non stamperà il messaggio "Sono qua", in quanto il programma trova un errore e termina, senza svuotare il buffer, che contiene ancora la stringa di output.
Ci sono tre possibili soluzioni a questo problema:

- inserire dopo il messaggio un "A capo". Questo metodo è abbastanza empirico, non è riportato da nessuna parte, ma funziona!
- inserire dopo ogni messaggio il comando `cout.flush()`, che svuota il buffer prima di terminare il programma. Questo funziona sempre!
- evitare di fare errori di questo tipo...

Attenzione: non mischiare funzioni di I/O del `C++` con funzioni del `C`, perché hanno due buffer diversi e i messaggi di debug potrebbero comparire nella sequenza sbagliata.

[2]Come si può notare, Mac rende sempre le cose facile per quanto riguarda l'uniformazione del codice

B.7 Software pubblico o privato

La Free Software Foundation è la fondazione che inneggia al software libero, ovvero alla distribuzione del codice sorgente a tutti gli interessati, senza alcun costo. Ad essa sono dovute quasi tutte le applicazioni per linux e UNIX.
I punti di forza di questa visione sono sostanzialmente due:

- ognuno può leggere il codice e capire cosa fa il programma;
- più gente legge il codice, più facilmente vengono trovati eventuali errori o bachi, quindi si migliora la qualità del software.

D'altra parte, questi poveri programmatori non vivono per la gloria e dovranno pur mangiare... Il trucco sta nel fatto che i soldi si fanno non sul programma, ma sull'assistenza al programma!

Ci sono poi le software house che invece hanno deciso di far pagare caro il proprio lavoro. Queste ditte, anziché distribuire il codice sorgente, mettono a disposizione dell'utente il codice *precompilato* (i file `*.o`) e il file header da includere nel programma. In questo modo il loro lavoro non risulta visibile a tutti. Ad esempio, la libreria `iostream.h` non contiene il codice della funzione `cin`, che è contenuto in un file precompilato di libreria.

B.8 Pronuncia

Le parole inglesi si pronunciano "all'inglese". Quindi avremo che

- `C++` si pronuncia "Ci più più", oppure (meglio) "Si plas plas" (come dev'essere...)
- `ASCII`, ovvero lo standard di codifica per i caratteri, essendo un acronimo inglese, si pronuncia "ASKII"

Elenco delle figure

Indice