Appunti fondamenti di programmazione

Giacomo Sansone

11 novembre 2019

Indice

1 21 Ottobre 2019

Operatore ternario L'unico operatore che opera su tre operandi è $'? :'$. Vediamone la grammatica

$$condizione \quad ? \quad espressione1 \quad : \quad espressione2$$

L'operatore valuta se la condizione è vera o falsa (la condizione deve quindi ritornare un tipo booleano, true o false). Se è vera, si ha l'espressione 1, altrimenti l'espressione 2.

```
int main(){
        int a=3; int b=5; int max;
        max = a>b ? a : b;
}
```

Le parentesi non sono necessarie, dato che l'operatore di assegnamento ha priorità minore rispetto all'operatore ternario.

Operatore unario di incremento postfisso Esso è applicato ad una variabile, è unario in quanto richiede un solo operando. E' un modo più sintetico per esprimere $var = var + 1;$, sebbene abbia delle caratteristiche particolari.

```
int main(){
        int i = 5
        int j= i++;
```

```
        cout << i; //6
        cout << j; //5
}
```

Le cose non vanno come ci si aspetta perché è vero che l'operatore incrementa la variabile i, ma restituisce il valore destro della variabile prima che questa venisse modificata. La variabile j prima prende il valore di i, poi questa viene incrementata.

```
Int main() {
        int i=5;
        int j=++i;
        cout << i << endl; //6
        cout << j << endl; //6
}
```

In questo caso, le cose vanno come ci si aspetta. Prima viene incrementato i, poi l'operatore restituisce il valore sinistro della variabile già incrementata. In questo valore sinistro, il risultato dell'operatore di incremento prefisso può essere utilizzato anche in operazioni di assegnamento, a differenza di quello postfisso.

```
Int main() {
        int i=9;
        cout << ++++i; //11
}
```

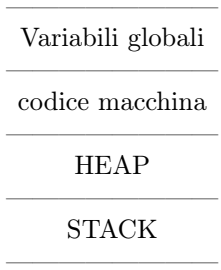
L'operatore di incremento prefisso è associativo da destra verso sinistra, quindi la scrittura è equivalente a $++(++i)$. All'operatore di $shift$ ridefinito sulla classe di uscita viene restituito il valore sinistra della variabile dopo il secondo incremento, e quindi ne viene stampato il valore. L'operatore di incremento postfisso non si può concatenare.

```
Int main() {
        int i=9;
        cout << i++++; //expression is not assignable
}
```

Il primo operatore restituisce un valore destro, mentre il secondo operatore necessita di un valore sinistro (l'operazione può essere accostata a qualcosa del tipo $10++$). Tutti gli operandi che riguardano l'assegnamento necessitano di un valore sinistro.

Visibilità e tempo di vita delle variabili La variabile j può essere utilizzata solo dentro al blocco in cui è stata definita. Se provassi ad usarla al di fuori da tale blocco, il compilatore mi dirà che la variabile non è definita. Infatti, la variabile j è stata distrutta. non appena si è usciti dal blocco. i è visibile in tutto il $main()$, quindi posso usare il suo identificatore per richiamarla in tutto il main. j la posso usare solo nel blocco in cui è stata definita e, perciò, si definisce 'variabile di blocco',(in realtà anche i è una variabile di blocco, perché viene distrutta alla fine del $main()$). Ciò accade affinché la variabili possa essere definita solo quando se ne necessita (in C, tutte le variabili dovevano essere dichiarate all'inizio del programma).

Vediamo dove sono allocate le variabili di blocco (le uniche che si sono viste finora). Alla fine della compilazione avrò un file eseguibile nel quale, in codice macchina, sono descritte le stesse cose che ho scritto in linguaggio C++. Il file, che in Windows ha estensione *.exe* è un file binario a cui corrispondono certe operazioni eseguibili per la specifica CPU. Mettendo in esecuzione il programma, questo viene copiato in memoria RAM. Il programma avrà quindi bisogno di uno spazio di memoria per definire le variabili. Le variabili di blocco vengono allocate a partire dal fondo della memoria disponibile. Si va in fondo alle ultime locazioni disponibili e ci si alloca una variabile. Questo succede subito per la variabile i del *main*(), poi si incontra la variabile j all'interno di un blocco e la si alloca 'sopra' la variabile i. Dopo che si esce dal blocco in cui quella variabile è stata definita, j viene deallocata, e sopra i c'è un blocco disponibile per una nuova variabile. Le variabili di blocco vengono quindi impilate una sopra l'altra.Via via che incontro una nuova variabile, viene impilata, cioè messa in cima allo *stack* (pila in inglese). Un'implicazione di questa forma è che non ci saranno mai dei buchi. C'è anche un'altra zona di memoria riservata, quella immediatamente successiva al codice macchina, la quale è riservata alle variabile dinamiche (la zona di chiama *HEAP*). La parte sopra il codice è invece riservata alle variabili globali. Sullo *stack* ci finiscono le variabili automatiche (o variabili di blocco o locali); le variabili che finiscono nello *HEAP* sono le variabili dinamiche; le altre variabili sono quelle globali.



Una variabile globale è definita fuori dal *main*().

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

int k; //Visibile nell'intero file

int main() {
        k=10;
        k++;
        cout << k; //11
        int j;  //Visibile nel main
}
```

Il compilatore sa quale sono le uniche e sole variabili globali di tutto il programma, quindi si riserva una parte della memoria per allocarcele. Queste variabili sono dette *statiche*, e vengono allocate una volta per tutte sopra lo spazio del codice macchina. Il tempo di vita delle variabili globali è lo stesso del programma. Non si creano variabili globali *runtime*(durante l'esecuzione

del programma), come invece accade per le variabili di blocco. Quando incontro una nuova variabile durante l'esecuzione, la vado ad inserire in cima allo *stack*, per poi deallocarla arrivati fuori dal blocco.

Il ciclo for Vediamo la grammatica del ciclo for.

```
st-prec;
for(espressione1; condizione; espressione2)
        statement;
st-succ;
```

Per prima cosa si esegue l'espressione1, poi viene valutata la condizione. Se è falsa, si passa subito allo *statement* successivo, altrimenti si entra all'interno del corpo del for, si esegue l'espressione2 e si valuta nuovamente la condizione.

```
// Stampare n asterischi a video

int main(){
        int n; cin >> n;
        for (int i=0; i<n ; i++){
                cout << "*" ;
        }
        return 0;
}
```

La variabile i viene inizializzata una volta sola, e ha come tempo di vita quello di una variabile all'interno del blocco del for. Se la variabile i servisse anche successivamente, potremmo dichiararla prima del ciclo for, e poi, all'interno del ciclo, assegnarle un valore. Uno *statement* accettabile è anche l'istruzione vuota, quindi potrei scrivere

```
for(int i=0; i<n;)
```

e incrementare i successivamente. Allora è legittimo anche

```
int i=0;
for (;i<n;){
        ...
}
```

l'unica differenza è che non do al for lo *statement* iniziale. Posso anche scrivere $for(;;)$, così che continui finché non incontro un *break* all'interno del blocco. Quando incontro un *break*, esco dal ciclo più interno che ne ha provocato l'istruzione.

Le funzioni Potrei scegliere di mettere da parte alcune righe di codice nella misura in cui risolvono un certo sotto-problema. Le funzioni, prima di essere chiamate, devono essere definite. Definiamo la funzione stampaAsterischi:

```
void stampaAsterischi(){
        for (int i=0; i<20; i+)
                cout << "*";
        cout << endl;
        return;
}
```

Posso richiamare questa funzione quante volte voglio. L'idea è quella di andarsi a creare una libreria di funzioni. Ogni funzione ha un certo *return*, un tipo di valore che consente di ritornare nel main per poter procedere con l'istruzione successiva. Tutte le funzioni devono restituire un valore, e il tipo del valore ritornato va specificato durante la definizione stessa. Quando non c'è da restituire alcun valore, si usa il tipo *void*. In questo caso, posso anche omettere la funzione *return*. Finora non abbiamo fatto altro che definire la funzione *main()*. Essa è una funzione come tutte le altre che restituisce un intero. Per convenzione, è 0 se è andato tutto bene, altri valori a scelta negli altri casi. Quando mando in esecuzione il *main()*, viene allocato in memoria e si comincia ad eseguire la sua prima. Una funzione può avere anche degli argomenti, specificati tra parentesi tonde.

```
int sommaFinoAEnne(unsigned int n){...}
```

Ogni funzione crea le sue variabili all'interno di un blocco, quindi le variabili della funzione quando sono chiamate sono impilate nello *stack*, poi cancellate una volta usciti dal blocco della funzione. Quando la funzione è chiamata, la variabile dell'argomento viene allocata in cima allo *stack*, così come la variabile che sarò ritornata dalla funzione. E' come se l'argomento fosse inizializzato ad ogni chiamata della funzione con il valore che gli si è dato dalla chiamata nel *main*. Quando la funzione termina, tutte le variabili locali sono cancellate, tranne la variabile temporanea che immagazzina il valore del risultato di ritorno. Si ha una variabile di cui non conosco il nome che la funzione ritorna nel *main*. Subito dopo, anche questa variabile viene deallocata. L'argomento formale viene dichiarato e inizializzato prima dell'inizio del blocco della funzione. E' qualcosa che avviene *runtime*, cioè solo quando la funzione è chiamata. Io nel *main* specifico un valore in fase di chimata, poi uso questo valore per inizializzare la variabile formale. Dopo che anche il valore di ritorno è stato utilizzato, sullo *stack* non ho più variabili che si riferiscono alla funzione. L'argomento attuale è quello che utilizzo in chiamata, l'argomento formale è quello generico che utilizzo per progettare la funzione. Le variabili globali dichiarate prima della dichiarazione della funzione sono viste anche dalle funzioni.

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

int k=0;
void azzeraK(){
        k=0;
}
void incrementaK(){
        k++;
}

int main(){
        azzeraK();
        cout << k << endl; //k=0
        incrementaK;
        incrementaK;
        incrementaK;
```

```
        cout << k << endl; //k=3
        return 0;
}
```

Conversioni implicite/esplicite In fase di compliazione, si possono avere una serie di conversioni effettuate in modo implicito. Ad esempio:

```
int main(){
        float f=3.4;
        int a=f;
}
```

mi da un *warning*, perché si ha una perdita di informazione(nel tipo intero non è salvata la parte decimale della variabile f), ma la compilazione avviene ugualmente. In questo caso si parla di conversione implicita, ma ci sono dei modi per esplicitare il tipo di conversione che voglio fare.

```
int i= int(f);
```

La funzione *int*() permette di restituire la parte intera della variabile di tipo $float$. Una conversione implicita è

```
if(i){
        ...
}
```

La variabile i, di tipo intero, viene automaticamente convertita in un tipo booleano, secondo la logica per cui $i == false \Leftrightarrow i == 0$.

2 22 Ottobre 2019

Lo Statement Switch Lo statement switch ci permette di fare delle scelte. Vediamone la grammatica.

```
switch-statement
        switch(condizione)
                switch body;

switch body ::=
                {alternative-seq}

alternative ::=
                case-label-seq statement-seq

case-label::=
        case cont-expression:
        default:
```

Vediamo un esempio del suo utilizzo:

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

```
int main(){
        int asterischi;
        cout << "Quanti asterischi? ";
        cin >> asterischi;
        switch (asterischi){
                case 1:
                        cout << "*" << endl;
                        break;
                case 2:
                        cout << "**" << endl;
                        break;
                default:
                        cout << "!" << endl;
        }
                return 0;
}
```

Accanto al *case* posso mettere costanti letterali per il tipo della variabile(in questo caso, dato che la variabile è di tipo *int*, interi). Il caso default ci permette di gestire tutti gli altri casi. Vediamo come si comporta questo *statement*. Esso confronta la condizione con i *case*. Se il valore della variabile è uguale alla costante immessa nel *case*, lo *statement switch* non fa solo la sequenza dello specifico *case*, ma anche tutti i *case* successivi. Se non ci fosse il *break* ed inserissi uno, stamperei sia uno che due che il punto esclamativo. Sfruttando la semantica di questo *statement*, si sarebbe potuto fare anche un'altra cosa.

```
switch(asterischi){
        case 3 : cout << "*" << endl;
        case 2 : cout << "*" << endl;
        case 1 : cout << "*" << endl;
        default: cout << "!"<< endl;
}
```

Se inserisco 3, si attiva sia il caso uno che il due che il tre e mi trovo stampato tre asterischi. Senza il *break*, mi si attivano anche le sequenze successive. Se non voglio che questo accada, mi devo ricordare di aggiungere l'istruzione *break*. Spesso lo switch si usa per le interazioni con l'utente

```
int main(){
        cout << "Inserisci un'alternativa: " << endl;
        cout << "A - prima alternativa. " << endl;
        cout << "B - seconda alternativa. " << endl;
        cout << "C - terza alternativa. " << endl;
        char c; cin >> c;
        switch(c){
                case A:
                        ...
                        break;
                case B:
                        ...
                        break;
                case C:
```

```
                ...
                break;
            default:
              cout <<"Scelta non valida."<< endl;
            return 0;
        }

}
```

Può essere che a più *case* sia legata la stessa parte di codice. Si avrà quindi:

```
case 'b':
case 'B':
        ...
```

Rappresentazione dei numeri interi Nel calcolatore, non posso rappresentare tutto l'insieme $\mathbb{Z}$, ma solamente un suo sotto-insieme simmetrico rispetto a 0. Infatti, avendo a disposizione un numero limitato di bit (siano ad esempio p), ho un numero limitato di combinazioni di 0 e 1, e, di conseguenza, solo certi numeri. Indichiamo con A l'insieme di tutte le combinazioni degli 0 e degli 1 realizzabili con p bit. A ciascuna sequenza posso o associare numeri senza segno (*unsigned int*), o con il segno (*int*)

000	0	0
001	1	1
010	2	-1
011	3	2
100	4	-2
101	5	3
110	6	-3
11	7	4

Serve una tabellina di corrispondenza. Nel caso degli interi allora si può procedere in un modo specifico. Si utilizza una formula matematica per poter esprimere sia numeri positivi che negativi a partire da una certa sequenza di bit, e poi la conversione inversa. Sono state fatte tre proposte, ma l'importante è stabilire una corrispondenza biunivoca per capire quale è il numero associata ad una certa sequenza.

Rappresentazione modulo e segno In questo tipo di rappresentazione dei numeri con il segno, il bit più significativo (quello a sinistra) rappresenta il segno: 0 per i numeri positivi, 1 per i numeri negativi. I successivi bit li uso per rappresentare il modulo, che è alla stregua dei numeri naturali (*div e mod* per il passaggio dal numero alla codifica bianaria). Da ciò, se fossimo su 3 bit:

000	$+0$
001	$+1$
010	$+2$
011	$+3$
100	-0
101	-1
110	-2

111 −3

Questa rappresentazione da luogo ad una rappresentazione simmetrica dove lo zero c'è due volte, sia come più zero che come meno zero.

Ad ogni rappresentazione di un numero intero corrisponde uno e un solo numero naturale espresso in binario (deve essere dato il numero p di bit, altrimenti non si ha la corrispondenza biunivoca). Dare la sequenza 001 e dare 1 è la stessa cosa se ho stabilito su quanti bit sto lavorando. Se non avessi fissato quanti bit, 1 è uguale a 1, 01, 001, 0001 ... e non si ha più la corrispondenza biunivoca.

Con p=3, posso rappresentare da -3 a 3. Questo vuol dire che, nel caso generico, con p bit posso rappresentare da $-2^{p-1}+1$ a $2^{p-1}-1$. Non posso quindi rappresentare -8 in modulo e segno su 4 bit.

Rappresentazione con il complemento a due Se $n > 0$, allora si rappresenta in binario il modulo di n, altrimenti rappresento in binario $2^p - a$. Il modulo lo ottengo con l'algoritmo *div e mod* ma, a seconda del segno del numero, devo prendere o un valore o un altro. L'idea è quella per cui $2^p - a$ è anch'esso un numero positivo. Come si passa dalla sequenza di 0 e 1 al numero di partenza?

- Se la cifra più significativa è zero, allora si ha che il numero corrisponde al modulo della sequenza successiva di bit (numero positivo).
- Se la cifra più significativa è 1, allora si ha che $n = -(2^p - a)$, dove a è il numero dato dalla conversione di tutti i p bit.

3 28 Ottobre 2019

L'intervallo di rappresentabilità per il complemento a due è $2^{p-1}/ + 2^{p-1} - 1$. L'intervallo è diverso rispetto al modulo e segno dato che qui lo zero è rappresentato una volta sola, quindi si ha una sequenza in bit in più (utilizzata in questo caso per un numero negativo). Ad esempio, se $p = 3$,

$$Intervallo : [-2^{3-1}, 2^{3-1} - 1] = [-4, 3]$$

Se il numero a è maggiore di zero, allora si rappresenta il modulo di a con l'algoritmo div e mod, dopo aver verificato che a rientri nel range di rappresentabilità per i p bit dati. Se A è minore di zero, di rappresenta con il div e mod il numero $2^p - |a|$. Ad esempio, per rappresentare -1 su tre bit, devo rappresentare $2^3 - 1$, ossia $(7)_{10} = (111)_2$. C'è un teorema che afferma che, facendo questa operazione di sottrazione, se a è all'interno del range di rappresentabilità, avrò sempre un numero positivo. Tutti i numeri negativi li riconosco subito, perché hanno il bit più significativo uguale a uno. C'è sempre una corrispondenza biunivoca tra il numero naturale su p bit e un intero. Anche per l'operazione inversa, quella che, partendo da numero in base due mi permette di risalire all'intero, c'è una formula.

$$A = (a_{p-1} == 0?)? + A : -(2^p - A)$$

Dove A è il numero naturale corrispondente alla sequenza di bit e a_{p-1} è il bit più significativo (quello più a sinistra).

$(1111)_2$ su 4 bit: $-(2^4 - 15) = -1$.

Il 99% delle unità aritmetico logiche (ALU) utilizzate nelle CPU sfrutta la rappresentazione in complemento a due. Perché? La CPU legge degli operandi dalla RAM, poi, una volta che li ha, utilizza una certa circuiteria hardwere per elaborarli. ALU si occupa proprio di interi e naturali. Le CPU posso poi avere anche la FPU, per i numeri a virgola mobile. La FPU potrebbe anche non esserci, mentre la ALU non può mancare. ALl'interno dell'ALU, servirebbe, se questa si occupasse solo dei naturali, una nuova circuiteria per le operazioni sui numeri interi. Questo perché ci verrebbe da dire che i naturali e gli interi sono rappresentati in modo diverso, e quindi devono essere elaborati in modo diverso. In realtà si sono accorti che il complemento a due gode di alcune proprietà molto interessanti. Queste proprietà fanno si che l'ALU possa elaborare anche i numeri interi, senza bisogno di una nuova circuiteria, equivalente a una maggior spesa in transistori, più possibilità di errori della produzione....

Supponiamo di voler sommare -3 e 2, entrambi su tre bit. In modulo e segno, si avrebbe $(-3)_{10} = (111)_2$ e $(+2)_{10} = (010)_2$. Sommandoli:

$$111 + 010 = 1001$$

Ma, dovendo lavorare su tre bit, ho $(001)_2$ che corrisponde a $(1)_{10}$. Ciò vuol dire che, con il modulo e segno, non posso sommare degli interi utilizzando la stessa circuiteria dei naturali, ossia l'ALU (questo è il modo standard con cui sono sommati i numeri naturali). Facciamo lo stesso calcolo con il complemento a due. $(-3)_{10} = (101)_2$ e $(+2)_{10} = (010)_2$. Sommandoli:

$$101 + 010 = 111$$

Con il complemento a due, posso sommare le rappresentazioni degli interi come se fossero dei naturali ed ottenere risultati corretti, purché si rimanga nel range di rappresentabilità dei due numeri di partenza. Si ha allora una regola empirica.

Se sommo due numeri negativi, e ottengo, escludendo il bit p+1, un numero positivo, allora la somma dei due numeri non è rappresentabile su p bit. Se sommo due numeri positivi e mi viene fuori un numero negativo, è la stessa cosa. Altrimenti, se ho un numero valido, esso è rappresentabile, al più escludendo il bit in posizione p+1

$-107 + 130 su 9 bit$.

$$(-107)_{10} = (110010101)_2$$
$$(130)_{10} = (010000010)_2$$
$$(130 + (-107))_{10} = (000010111)_2 = (23)_{10}$$

Rappresentazione con bias Con p bit, posso rappresentare $2^p - 1$ numeri naturali. Nella rappresentazione dei numeri interi con bias, è come se traslassi tutti i numeri indietro, per far coincidere con 0 (e quindi con la sua rappresentazione in bit) il più piccolo numero rappresentabile. Avendo p bit, si ha che

$$Bias = 2^{p-1} - 1$$

Partendo dal numero intero a, considero a+bias e lo rappresento come un numero naturale. Allo stesso modo, se ho il corrispondente numero in bit, considero il naturale corrispondente e calcolo a-bias. Intervallo di rappresentabilità:

[-bias,bias+1]

Numeri a virgola fissa Nel rappresentare i numeri reali, si potrebbe andare incontro sia ad un overflow sia ad un undeflow, quando vorrei andare troppo in profondità con il numero. In generale allora, si avrà di fatto un insieme discreto dei numeri reali. Nella rappresentazione a virgola fissa, uso un numero fisso di bit per la parte intera ed uno fisso per la parte frazionaria. Indichiamo con $I_{(r)}$ la parte intera del numero, con $F_{(r)}$ la parte frazionaria.Siano p bit per rappresentare il numero r, f per la parte frazionaria e $(p-f)$ per la parte intera. si ha che:

$$R = a_{p-f-1}...a_0 a_{-1}...a{-f}$$

$$r = \sum_{i=-f}^{p-f-1} a_i \beta^i = a_{p-f-1}\beta^{p-f-1} + \ldots + a_0\beta_0 + a_{-1}\beta^{-1} + ... + a_{-f}\beta^{-f}$$

Ad esempio

$$(+1110.01)_2 = (1*2^3+1*2^2+1*2^1+0*2^0+0*2^{-1}+1*2^{-2}) = 8+4+2+0.25 = 14.25$$

Notare che la virgola non si rappresenta, La parte intera $I_{(r)}$ si rappresenta con le tecniche note mentre, per la parte frazionaria, si usa il seguente procedimento.

Si usa la procedura *parte frazionaria - parte intera.*

$f_0 = F_{(r)}$. Se $f_0 \neq 0$,

$$f_{-1} = F(f_0 * 2) \quad a_{-1} = I(f_0 * 2)$$

$$f_{-2} = F(f_{-1} * 2) \quad a_{-2} = I(f_0 * 2)$$

$$\ldots$$

Finché f_{-j} è uguale a zero oppure si è raggiunta la precisione desiderata.

Esempio: $p = 16$, $f = 5$, $r = +331,6875$.

$$(331)_{10} = (101001011)_2$$

$$f_0 = 0.6875$$

$$f_{-1} = f_(1.375) = 0.375 \quad a_{-1} = I(1.375) = 1$$

$$f_{-2} = f_(0.75) = 0.75 \quad a_{-2} = I(0.75) = 0$$

$$f_{-3} = f_(1.5) = 0.5 \quad a_{-3} = I(1.5) = 1$$

$$f_{-4} = f_(1) = 0 \quad a_{-4} = I(1) = 1$$

$$r = 0000000101001011.1011$$

Spesso, per esprimere r, sarebbero necessarie un numero infinito di cifre. Avendone a disposizione un numero limitato, si effetuerà un troncamento. Ad esempio, $r = 2.3$ con $f = 6$.

$$(2)_{10} = (10)_2$$

$$f_0 = 0.3$$

$$f_{-1} = f_(0.6) = 0.6 \quad a_{-1} = I(0.6) = 0$$

$$f_{-2} = f_(1.2) = 0.2 \quad a_{-2} = I(1.2) = 1$$

$$f_{-3} = f_(0.4) = 0.4 \quad a_{-3} = I(0.4) = 0$$
$$f_{-4} = f_(0.8) = 0.8 \quad a_{-4} = I(0.8) = 0$$
$$f_{-5} = f_(1.6) = 0.6 \quad a_{-5} = I(1.6) = 1$$
$$f_{-6} = f_(1.2) = 1 \quad a_{-6} = I(1.2) = 1$$
$$r = 10.010011$$
$$R = 2.296875$$
$$errore = 0.003125$$

4 29 Ottobre 2019

Le variabili costanti (attributo const) Queste sono delle vere e proprie variabili, con un valore sinistro che quindi occupano spazio in memoria ma che non possono essere modificate. Fintanto che la variabile non è distrutta, avrà sempre il valore con cui è stata inizializzata. Con la variabile costante, posso farci tutto quello che facevo con quelle non costanti. Il compilatore esegue un controllo sul codice per assicurarsi che la variabile costante non sia mai modificata. Non si può creare una variabile costante con l'idea di inizializzarla successivamente. Il valore destro rimane sempre lo stesso. Spesso, nel dichiarare una costante, si usa un identificatore tutto in maiuscolo.

Il tipo derivato riferimento .

```
int main(){
        int i=10;
        int &r =i //Variabile riferimento a intero
}
```

r è una variabile di tipo riferimento. Il suo tipo non è int, ma int&, riferimento a intero. Creando un riferimento per la variabile i, di fatto ne stiamo creando un alias, un altro nome per la variabile. Se andassimo a vedere la tabella delle variabili, i sta sul fondo dello stack in quanto variabile di blocco. L'indirizzo è salvato in base 16 (è la prima cella di memoria della variabile), e i avrebbe due indentificatori, sia i che r. In quanto suo alias, fare *cout* << *i*; o *cout* << *r* stampa la stesa cosa, perché gli identificatori si riferiscono alla medesima locazione in memoria.

```
        r++;
        cout << i; // 11
```

In questo modo creo un alias, un altro nome per una variabile. Il riferimento non può essere modificato successivamente, e neanche inizializzato dopo la sua dichiarazione. Devono avvenire simultaneamente le due cose. Quello che vado a considerare è un tipo derivato. I tipo che finora si sono incontrati sono, ad esempio, int, bool, char, double, float, int&... Dato che il riferimento è un tipo derivato, posso creare un riferimento a qualunque tipo fondamentale. Non posso creare un riferimento ad un literal, ad una costante letterale che non ha un valore sinistro. r è un altro nome per i, si va a considerare la stessa locazione in memoria. Quando però vado a dichiarare r, non sto allocando memoria alla variabile, ma soltanto creando un altro identificatore per la variabile i. Ma a cosa serve il tipo riferimento? Chiediamoci, come si scambiano due variabili?

```
int main(){
        int i=5;
        int j=7;
        int aux=i;
        i=j;
        j=aux
}
```

E se volessi creare una funzione che scambia due variabili?

```
void scambia(int m, int n){
        int aux=m;
        m=n;
        n=aux;
        return;
}
```

Consideriamo questa funzione. In realtà, quando vado a chiamarla con due variabili i e j come argomento nel main, ho gli stessi valori di prima senza alcun cambiamento. Come mai? Vediamo cosa succede sullo stack. E' stata creata la variabile i di valore 5, poi j di valore 7. Si entra poi nella funzione. Non creo un a variabile di ritorno dato che la funzione è di tipo void, tuttavia creo due variabili formali, m di valore 7 e n di valore 5. Per effettuare lo scambio, creo anche la variabile aux di tipo intero. A questo punto avviene lo scambio, ma tra le variabili formali, non su quelle di partenza del main. Infatti, pur avendole scambiate, alla fine del blocco, m, n e aux sono deallocate, ed è come se nulla fosse successo. Ma posso allora scrivere una funzione che agisca direttamente sulle variabili?

```
        void scambia(int& m, int& n){
        int aux=m;
        m=n;
        n=aux;
        return;
}
```

Vediamo, passando un riferimento, cosa succede sullo stack. Le variabili i e j vengono create nel main, poi viene chiamata la funzione scambia. Sullo stack vengono create non delle variabili intere, che occupano una certa memoria, ma solamente degli alias, che si riferiscono quindi alla stessa locazione in memoria. Quando vado a scrivere m=n, questo non si riferisce più a due variabili formali, ma alle stesse variabili del main con nomi diversi. Fuori dalla funzione, mi ritrovo con i valori delle due variabili i e j scambiati. E' come se, all'inizio della funzione, ci fosse

```
int &m =i;
int &n =j;
```

Con il tipo riferimento sono riuscito a lavorare direttamente sulle variabili con cui ho chiamato la funzione. Quindi, in generale, se la funzione deve modificare la variabili, devo passare dei riferimenti, altrimenti dei tipi interi.

Il tipo derivato puntatore .

```
int main(){
        int i=10;
        int* puntatore;
}
```

Questa è una variabile di tipo derivato da intero. Al contrario del tipo riferimento, una variabile di tipo puntatore a intero occupa uno spazio in memoria. Questa variabile ha come contenuto degli indirizzi di memoria di variabili dello stesso tipo. Posso anche creare dei puntatori per altri tipi, che, appunto, hanno come contenuto posizioni delle variabili dello stesso tipo. Tuttavia, non posso scrivere

```
puntatore=i;
```

Perché il puntatore non può prendere il valore destro della variabile i. Si deve usare l'operatore di indirizzo, unario e prefisso, che serve per prendere l'indirizzo di memoria di una varibile.

```
puntatore=&i;
```

& può stare sia per il tipo riferimento, sia per l'operatore unario prefisso di indirizzo, sia per l'and bit a bit. Esso restituisce come valore di ritorno l'indirizzo di memoria di una variabile. Vediamo come stanno le cose sullo stack. A differenza del tipo riferimento, che crea solo un sinonimo, una variabile di tipo puntatore occupa dello spazio. Esso ha come valore destro la locazione di memoria di una variabile di uno stesso tipo. Il tipo ci dice quanta memoria occupa e le operazioni che si possono effettuare. La variabile di tipo puntatore non può assumere valori negativi, dato che le celle di memoria si indicano in progressione a partire dalla cella numero 1. Tutti gli indirizzi di memoria sono quindi valori positivi. La variabile di tipo puntatore si dice che sta puntato alla variabile i.

Per i puntatori, esiste anche l'operatore di dereferenziamento. Questo operatore restituisce il valore destro (ma in realtà anche sinistro) della variabile puntata dall'operatore. *puntatore è come se fosse un altro nome per la variabile i. Ogni volta che lavoro con *puntatore, è come se stessi lavorando con i. Poniamo che io abbia la variabile j. Posso far si che puntatore smetta di puntare i e punti j? Si. In puntatore, con l'assegnamento, finisce l'indirizzo di memoria della variabile j, al posto di i. Ad un puntatore non posso assegnare un literal, dato che gli indirizzi di memoria variano da esecuzione ad esecuzione. Tuttavia, ha senso il puntatore NULL, equivalente all'indirizzo di memoria 0.

Non ha senso un'operazione del tipo

```
int main(){
        int* q;
        *q=5;
}
```

Perché il puntatore a intero q non sta puntando a nulla. Questo potrebbe passare come inosservato al compilatore, creando grossi problemi nell'esecuzione.

5 4 Novembre 2019

Effetti collaterali Si definisce effetto collaterale il caso in cui una funzione modifica una variabile non locale alla funzione stessa. S tratta quindi della

modifica di una variabile che non viene creata e distrutta all'intero del blocco della funzione stessa.

```
int contatoreglobale;
void azzeracontatore(){
        contatore=0;
}
```

Questa funzione ha un effetto collaterale, perché modifica una varibile non locale. Anche la funzione che scambia due valore ha un effetto collaterale, dato che modifica due variabili inizializzate nel main.

```
void scambia(int* a, int*b){
        int old= *a;
        *a=*b;
        *b=old;
}

int main(){
//...
        scambia(&a,&b);
}
```

Ancora sui puntatori In questo caso come argomento della funzione ho due puntatori, che già nel main puntano a due variabili distinte. Le variabili locali che vengono create nel blocco della funzione puntano alla variabile giusta nel main, pur essendo degli argomenti formali. E' come se, all'inizio del blocco, fossero create due variabili di tipo puntatore ad intero ciascua delle quali ha come valore l'indirizzo di memoria comunicato in fase di chiamata della funzione. Quando dereferenzio il puntatore, sto di fatto andando a considerare la variabile automatica del main. Notare che questa funzione riceve l'argomento per valore, e non per riferimento. Le variabili che sono create all'inzio del blocco della funzione occupano effettivamente uno spazio in memoria, ma, essendo puntatore, con l'operatore * ci si può lavorare modificando i valori delle variabili a cui puntano.

Se volessi inizializzare un puntatore, non potrei scrivere

```
void inizializzapuntatore(int* q){
        q=NULL;
}

int main(){
        int* P;
        inizializzapuntatore(P);
}
```

In questo modo, all'inizio del blocco della funzione si sta creando una variabile sullo stack, ed è questa che vado poi a modificare, non il puntatore P. Devo quindi passare il puntatore P per riferimento, in modo tale che q sia un alias del puntatore che voglio modificare.

Quando vado a creare un puntatore, è bene inizializzarlo subito a NULL. Infatti, il compilatore non va direttamente ad inizializzarlo a NULL, e il puntatore

punta ad una cella di memoria casuale. Per il fatto che un indirizzo di memoria si indica in forma esadecimale e zero è a sua volta un indirizzo di memoria, posso utilizzarli, per mezzo di una conversione implicita, anche come valori di tipo booleano. Consideriamo

```
//...
if(p){
/...
}
//...
```

Se p non è NUll, l'espressione è considerata come vera, e si entra nel blocco dell'if.

Tra i puntatoi e gli interi è definita l'operazione di somma (che invece non esiste tra puntatori).

```
int a=5;
int* p=&a;
int* z=p+3;
```

Nel puntatore z ci finisce il valore $p + (3 * sizeof(\ int))$, cioè l'indirizzo di memoria che succede p di 3 volte il numero di celle di memoria occupate da una variabile di tipo intero. Tra puntatori è poi definita la differenza.

```
int*p;
int*a;
//...
int z=p-a;
```

In z ci finisce la differenza tra la le posizione delle celle di memoria (viste come naturali) diviso la il sizeof(int*), il numero di celle occupate da una varibile di tipo puntatore a intero. Questa cosa è utile lavorando con gli array.

Gli array Un vettore in c++ è una sequenza di n elementi dello stesso tipo che si trovano in celle di memoria contigue. Questo significa che sono variabili omogenee. Per dichiarare un array automatico (una variabile di blocco, che quindi va a collocarsi impilato sullo stack) devo scrivere

```
int main(){
        int a=3;
        double v[a];
}
```

Ogni variabile dell'array ha un nome, che si indica con v[1], v[2]... E' fondamentale che non ci siano buchi tra le locazioni di memoria delle variabili del vettore. Sullo stack, alloco spazio per ciascuna variabile. La prima variabile ad essere creata sullo stack è l'ultima:

```
v[0]
v[1]
v[2]
a
...
```

Ogni componente dell'array è una variabile a se stante, quindi ci posso lavorare in modo autonomo. Ogni varabili ha un indirizzo, quindi posso creare volendo un puntatore che punta ad una variabile dell'array. Ad esempio

```
int main(){
        double* p=&v[0];
        cout << *p;
        p++;
        cout << *p;
}
```

Cosa ho fatto in questo modo? v[0] è una variabile di tipo double, quindi ha un suo indirizzo. Questo è l'indirizzo della prima delle otto celle della variabile. Dal tipo della variabile risalgo al numero di byte che occupa. In p ci finisce l'indirizzo di memoria di v[0], poi ne stampo il valore. p++ non incrementa il puntatore di 1, ma di 1*sizeof(double). Viene quindi incrementato di 8, facendo si che adesso esso punti a v[1]. Le celle consecuitive quindi sono all'interno di una singola variabile così che, aumentando o diminuendo il valore del puntatore, io vada sempre sulla prima cella di una delle variabili dell'array.

In c++, al nome dell'array è associato l'indirizzo della sua prima componente. Posso quindi fare.

```
int main(){
        int v[3];
        int* p=v;
        cout << *p; //v[0]
        p++;
        cout << *p; //v[1]
}
```

Questo vale anche per l'identificatore per le variabili scalari, che si riferisce all'indirizzo di memoria della prima cella della variabile. Il fatto che v[1] corrisponda alla seconda cella dell'array non è un caso. Infatti, [] è a sua volta un operatore, che restituisce *(v + i). [] ha come valore destro il nome di una varibile vettoriale, come valore sinistro un intero. L'operatore considera la cella v+i e ne dereferenzia il valore. Se ho un vettore di tre componenti, non posso considerare v[3], perché questa vuol dire la quarta componente dopo quella di partenza v[0], che quindi non esiste (o meglio, potrebbe essere qualsiasi cosa). Il primo indice dell'array è zero prorpio perché v[0]==*(v+0). Il compilatore, non effettua controlli sull'indice di un array, dato che questi dovrebbero essere fatti runtime, ma impiegherebbero molto tempo e sarebbero dispoendiosi.

Per fare una funzione che lavora sui vettori, devo passare un puntatore alla prima cella di memoria del vettore, e poi un intero che ne rappresenta la lungehzza. Ad esempio, se volessi scambiare gli elementi di due vettori che hanno la stessa lunghezza:

```
void scambia(int* v1, int*v2, lunghezza){
        for(int i=0; i<lunghezza;i++)
                v1[i]=v2[i];
}
```

Se scrivessi solo v1=v2, starei facendo un operazione tra puntatori. Una variabile vettoriale non può cambiare dimensione, perchè, in fase di allocamento, viene posta sullo stack, costretta tra una superiore e una inferiore.

6 5 Novembre 2019

Riferimenti costanti Posso dichiarare una varibile non costante ma usare un certo alias costatante che mi impedisce di modificarla. Questi alias depotenziati possono accedere alla variabile solo in lettura. Questi alias costatni saranno usati per passare alla funzione un riferimento ad una variabile senza che questa possa essere modificata.

```
int main(){
        int i=5;
        const int &j=i;
        j=6; //errore

}
```

Studiare da un'altra parte le variabili costanti a puntatore e puntatore a costante

Le CStringhe Quanto detto per i vettori di tipo inetro vale anche per i vettori di tipo char. Tuttavia, si parla in c++ delle cstringhe. Una cstringa è un qualunque array di caratteri che contenga ad un certo punto il carattere di terminazione '
\0'.

```
char vc[]={'C','i','a','o','\0'};
```

Questo non è solo un array di caratteri, ma una cstringa. Queste stringhe hanno la possibiità di essere stampate con un cout senza dover fare un intero ciclo for.

```
cout << vc; //Ciao
```

Il carattere di fine stringa permette di fare questa stampa. La marca di fine stringa è necessaria quando vado a stampare a schermo per mezzo del cout (e non con una funzione che, sapendo a priori il numero di elementi, li scorre tutti uno dopo l'altro). Il cout infatti va avanti finché non trova un carattere di fine stringa. Senza quello, andrebbe avanti a scorrere le celle di memoria (anche non più appartenenti alla stringa) finché non trova una cella che, in ASII, corrisponde al carattere di fine stringa. Grazie alle proprietà delle cstrighe, posso dichiarne una in questo modo.

```
char vc[] = {"Ciao"};
cout << vc; //Ciao
```

Il compilatore infatti inserisce automaticamente il carattere di fine stringa, in modo che si crei un vettore di cinque elementi, e non di quattro. Può essere utile avere una funzione che mi conta i caratteri della stringa escludendo il carattere terminale.

```
int my_strlen(char* const str){
        int lun=0;
        for(; str[lun]!='\0';lun++)
        return lun;
}
```

Questa funzione ha come argomento costatne il puntatore del primo elemento della stringa, che viene scorsa finché non si trova il carattere di fine stringa. Si usa l'attributo const perché la funzione non può modificare la stringa, qunindi non si possono avere effetti collaterali. Si può anche avere la necessità di una funzione che copi il contenuto di una stringa in un'altra.

```
void my_strcpy(const char*, char*);



int main(){
//...
}

void mystrcpy(const char* str1, char* str2){
        for(int i=0; str1[i]!='\0'; i++)
                str1[i]=str2[i];
        str2[i]='\0';
}
```

Ovviamente, quando vado a fare operazioni di questo tipo, devo sempre assicurarmi che il numero di caratteri della stringa che riceve sia sufficiente per contenere la stringa di partenza. In più, alla fine devo aggiungere il carattere di fine stringa che, per come è fatto il ciclo for, non viene copiato automaticamente. Si sarebbe potuto modificare il main in modo che, se la stringa di partenza fosse

stata troppo lunga, la copia non sarebbe nemmeno avvenuta. Tra le funzioni utili per le stringhe e presenti nella libreria $< cstring >$ c'è la stringcmp, che mi permette di sapre quale delle due parole viene prima sul dizionario. Questa si basa su un ordinamento lessico-grafico. Su dizionario, *ancora* viene prima di *zucca*, non per un fatto di lunghezza, ma perché si vanno a confrontare le lettere. Alla prima disuguaglianza che si incontra, posso giungere ad una conclusione su quale sia 'superiore', altrimenti vado avanti. Si stanno confrontando i numeri naturali corrispondenti al codice ASCII di ciascuna lettera, quindi si andrà avanti finché non si trova una superiore (ad esempio, $'a' <' z'$). Se i caratteri di una stringa sono conclusi ma ce ne sono ancora per la seconda, allora questa è superiore. Se alla fine sono concluse entrambe le stringhe, queste sono uguali. Implementiamo la fuznione my_strcmp, che restituisce 1 se la prima stringa è superiore, -1 se la seconda stringa è superiore, 0 se sono uguali.

```
int my_strcmp(const char* str1, const char* str2){
        int i=0;
        while(str1[i]!='\0' || str2[i]!='\0'){
                if(str1[i]>str2[i])
                        return 1;
                if(str1[i]<str2[i])
                        return 1;
                i++;
        }
        if(str1[i]!='\0' && str2[i]=='\0')
                return 1;
        if(str1[i]=='\0' && str2[i]!='\0')
                return -1;
        return 0;
}
```

Un'altra funzione che potrebbe essere utile è quella che mi cambia un carattere maiuscolo in uno minuscolo, in modo che la parola "Ciao" sia uguale a "ciao".

```
void my_tolower(char* s){
        for(int i=0; s[i]!='\0'; i++){
                if(s[i]>='A' && s[i]>='Z'){
                        s[i]=s[i]-'A'+'a';
                }
        }
}
```

I numeri reali a virgola mobile Nella virgola fissa, l'errore è costatne sia per i numeri molto grandi che per quelli molto piccoli. Quello che conta in realtà sarebbe l'errore relativo, cioè il rapporto tra l'errore e la quantità considerata. Per arrivare a questo risultato, si usa l'approccio che usano i fisici per poter rappresentare, con pochi numeri, sia cifre molto grandi che molto piccole: la notazione scientifica. Sappiamo che $3.14 = 0.314 * 10^1$. Per fare questa operazione, ho isolato quella che viene chiamata mantissa. Esiste, per ogni notazione,

la così etta forma normalizzata, che ha certe caratteristiche per quanto riguarda la parte intera della mantissa. Ad esempio, un numero a virgola fissa può essere scritto come 1110.01 ma anche $0.111001 * due^{100}$, dove 100 indica quattro in base 2. La forma normalizzata per questa notazione con l'esponenziale in base due è quella che ha come parte intera della mantissa 1. in questo caso è $1.11001 * due^{11}$. Notiamo che, dando per scontato l'uno ogni volta, io posso considerare solo la parte frazinaria della mantissa, saltando un uno. Questo indica l'"uno implicito". Un numero in virgola mobile è formato da tre parti: un esponenziale che si rappresenta con bias; la mantissa; un bit per il segno. Per sapere quanti bit usare per le varie parti, si usa lo standard ISEE 754 del 1985. In genere, vi sono tre possibilità.

- Half precision: non supportata su C++, si basa su 16 bit di cui uno per il segno, 5 per l'esponente e 10 per la mantissa.
- Single precision: si basa su 32 bit, di cui 1 per il segno, 8 per l'esponente, 23 per la mantissa. Corrisponde al tipo float su C++
- Double precision: si basa su 64 bit, di cui 1 per il segno, 52 per la mantissa e 11 per l'esponente.Corriponde al tipo double su C++.

Appunti Fondamenti di programmazione

Indice

Vettori in memoria dinamica

Questi vettori saranno chiamati anche vettori dinamici. Queste non sono variabili automatiche, non vengono allocate sullo Stack. Il vantaggio è che posso scegliere io quando distruggerle. Questi vettori sono gli unici candidati a permettere un vettore di dimensione non fissata. Sullo Stack, questo sarebbe impossibile, perché il tempo di vita di una variabile è fissato sulla base della posizione in cui quella variabile è creata. Consideriamo una funzione che vuole restituire un vettore che concatena due vettori. Una prima tentazione può essere quella di scrivere

```
int* concatena(int v1[],int v2[], int dimensione1, int dimensione2){
        Int v3[dimensione1 + dimensione2];
        …
}
```

Il problema di una soluzione di questo tipo è che v3 è una variabile automatica, per cui viene distrutta appena finisce l'esecuzione della funzione in cui è stata creata. Per risolvere questo problema, il vettore deve essere allocato su un'altra parte della memoria, in modo che possa essere utilizzato anche al di fuori del blocco della funzione. Il tempo di vita di una variabile allocata sullo Heap è stabilito dal programmatore. Mentre lo Stack si poteva raffigurare come una pila, lo Heap è più come un mucchio di variabili, dove si perde il concetto di continuità per il fatto che le varie variabili vengono allocate dove c'è posto. Per poter allocare un oggetto sullo Heap, serve l'operatore new, che alloca una variabile del tipo specificato dopo l'operatore (es. new int;) e restituisce l'indirizzo di memoria di tale variabile, che dovrà poi essere salvato in un puntatore. La funzione per concatenare due array allora diventa:

```
        int* concatena(int v1[],int v2[], int dimensione1, int dimensione2)
        int* v3= new int[dimensione1 + dimensione2]; // Alloco sullo heap un vettore
        int posizione=0;
        for(int i=0; i<dimensione1; i++, posizione++){
                v3[posizione]=v1[i];
        }
        for(int i=0; i<dimensione2; i++, posizione++){
                v3[posizione]=v2[i];
        }
        return v3;

}
```

A questo punto risulta necessario un operatore speculare che va sullo heap a partire dall'indirizzo salvato in quel puntatore per eliminare quella locazione di memoria. Per deallocare il vettore serve l'operatore delete.

```
Delete [] v3;
```

Il compilatore tiene traccia del numero di elementi del vettore puntato da v3 a runtime, in modo che l'operatore delete possa deallocare il numero giusto di componenti. Se l'operatore delete

avesse accettato un intero che indica la dimensione dell'array, si sarebbe perso il fattore 'dinamicità' e, in più, si sarebbe potuti incorrere in molti errori. Se con l'operatore new non specifico il numero di componenti, posso anche creare una variabile sullo heap. In questo modo non ho un identificatore per la variabile, ma solo un puntatore in cui mi sono salvato il suo indirizzo e con il quale posso accedervi.

Matrici

Una matrice è un array bidimensionale. Esistono anche array a tre dimensioni, che prendono il nome di tensori. Per un matematico, è come se stessimo trattando un oggetto che appartiene a $\mathbb{R}^{mxn}$. Anche in informatica, una matrice è composta da elementi omogenei, tutti dello stesso tipo. In più, se alloco un vettore sullo Stack, tutte le sue componenti saranno consecutive.

Int M[2][3]={ {1,2,3} , {4,5,6} };

Le varie componenti devono essere allocate in memoria una dopo l'atra, ad indirizzi crescenti riga per riga in modo che la componente di riga e colonna 0 abbia l'indirizzo di memoria più piccolo. Ogni componente della matrice ha come nome M[i][j]. Anche in questo caso, le parentesi quadre rappresentano un operatore che mi permette di dereferenziare il giusto indirizzo in memoria. La posizione si ottiene grazie all'aritmetica dei puntatori, dato che M[i][j] corrisponde a

**(M + colonne*i + j)*

Nel caso in cui M sia un puntatore al primo elemento della matrice. Usando questa formula, posso accedere a qualsiasi elemento della matrice.

Una funzione che azzera gli elementi di una matrice può essere scritta in questo modo:

```
void azzera( int* matrice, int colonne, int righe){
        for(int i=0; i<righe; i++){
                for(int j=0;  j<colonne; j++){
                        *(matrice +i*colonne +j)=0;
                }
        }
}
```

Con l'aritmetica dei puntatori, prima mi sposto sulla riga giusta, poi vado a cercare l'elemento giusto sulla colonna. In generale, posso usare la notazione a doppio indice nel main, ma risulta complicato farlo nelle funzioni. Un modo è quello di avere come argomento formale della funzione
Int M[][colonne], cioè una matrice in cui sono specificati solo il numero di colonne. Questa cosa però si può fare solo se il numero di colonne è una variabile globale, per cui non si può modificare esecuzione in esecuzione. In generale allora, si dovrà passare alla funzione un puntatore al primo elemento della matrice ed utilizzare l'aritmetica dei puntatori per accedere ai vari elementi.
Se creo una matrice, il suo tipo non è int, ma int * [4], cioè un puntatore ad un vettore di quattro righe. Scrivere M[i][j] allora corrisponde a scrivere

**(M + i*sizeof(int[4]) + j*sizeof(int)) //Mi sposto prima riga giusta poi sulla colonna*

Con il primo sizeof mi sposto sulla riga giusta, con il secondo sulla colonna giusta. Per passare una matrice automatica alla funzione ed usare il doppio indice devo specificare nel prototipo il numero di colonne, in modo che si possa eseguire l'operazione scritta sopra.

Si può anche allocare una matrice sullo Heap. In questo modo, sapendo il puntatore a puntatore su cui è stato allocato il tutto, è possibile utilizzare la notazione a doppio indice.

```
Int ** matrice = new int*[righe];
For( int i=0; i<righe; i++){
        Matrice[i]= new int[colonne];
}
```

Prima si crea un vettore di puntatori che corrisponde alle righe della matrice, poi su ciascuno di questi puntatori si alloca un altro vettore di tanti elementi quanti il numero di colonne.

Algoritmi di ordinamento

Dato un vettore ordinato, come facciamo a capire se un certo elemento è presente o no? Dobbiamo scrivere una funzione che restituisce true se è presente, false se non lo è. Questa funzione deve necessariamente scorrere tutto il vettore, perché fintanto che non si è arrivato all'ultimo elemento, non si può dire se l'elemento esiste. Se ci sono n componenti, allora si deve fare n controlli. Un algoritmo del genere ha complessità O(n). Il concetto di complessità computazionale è il seguente: O grande è l'ordine della complessità, e ci dice quanto ci vuole per arrivare al risultato in termini asintotici. Il selection sort ha complessità O(n^2), così come il Bubble Sort. Questo perché in entrambi c'è un for dentro un for. Si tratta di passare in rassegna tutti gli elementi di una matrice quadrata. Che succede se voglio trovare l'elemento 7 all'interno di un vettore ordinato? Siccome il vettore è ordinato, posso calcolare l'elemento centrale, quello di posizione intermedia. Se il queryElement è 7 e l'elemento centrale non è 7 ma è maggiore, posso fare la stessa operazione sulla parte precedente del vettore: se l'elemento precedente esiste, allora sta nel sotto vettore di sinistra. A questo punto, si va a fare la stessa cosa. Si pesca ad esempio 5, che è inferiore: se l'elemento è presente, sarà nel sotto vettore di destra. Arrivati al punto in cui c'è un solo elemento, è quello giusto si restituisce true, altrimenti false. Questo è un algoritmo di ricerca binaria. È un vero algoritmo perché si va sempre a dimezzare, quindi prima o poi terminerà. Oltre tutto, la complessità è legata al logaritmo in base due di n. La ricerca di un elemento in un vettore già ordinato è estremamente più efficiente. A seconda del caso, può essere utile ordinare un vettore con un algoritmo efficiente e poi andare a ricercare l'elemento. Se le query sono tante, ne vale la pena.

Vediamo il selection sort. Dato un vettore di partenza, vado ad identificare l'elemento più piccolo. Scambio questo elemento con quello in posizione 0. A questo punto cerco il più piccolo tra quelli di posizione 1…n-1 e lo scambio con quello in posizione 1. Questa cosa va avanti finché no ho l'array ordinato.

```
//SELECTION SORT
#include <iostream>
Using namespace std;
```

```
Int posMinimo( const int* v, int partenza, int arrivo){
        posizioneMinimo=partenza;
        for(int i=partenza+1; i<arrivo; i++){
                if(v[i]<v[posizioneminimo){
                        posizioneminimo=i;
                }
        }
        Return posizioneMinimo;
}

Void scambia(int* v, int infe, int n){
        Int aux = v[infe];
        v[infe]=v[n];
        v[n]=aux;
}



Void selectionSort( int* v, int dimensione){
        for(int i=0; i<dimensione; i++){
                int posizione=posMinimo(v,i,dimensione);
                if(posizione!=i)
                        scambia(v,i,posizione);
        }
}

Int main(){
        …
}
```

Questo algoritmo ha complessità n^2 indipendentemente dall'array, e non riesce a capire quando l'array è ordinato. Un'alternativa è il bubble sort, che ha complessità n^2 solo nel caso peggiore, ma che riesce a riconoscere un array ordinato.

```
//BUBBLE SORT
#include <iostream>
Using namespace std;

Void scambia(int* v, int a, int b){
        Int aux= v[a];
        V[a]=v[b];
        V[b]=aux;
}

BubbleSort(int* v, int dimensione){
        For(int i=0, i<dimensione, i++){
```

```
                Bool ordinato=true;
                for(int j=i; j<dimensione-1; j++){
                        if(v[j]<v[j+1]){
                                scambia(v,j,j+1);
                                ordinato=false;
                        }
                }
                If(ordinato)
                        Return;
        }
}

Int main(){
…
}
```

L'unità di compilazione

Può capitare che convenga implementare un programma su più file sorgente. Un problema complesso potrebbe essere spezzato in più file, ciascuno dei quali con una parte della soluzione, che poi andranno messi insieme in un solo file eseguibile. Questa idea si realizza nei concetti di modulo cliente e modulo servitore.

Da una parte c'è il mondo dei moduli che utilizzano le risorse, dall'altra quello di coloro che cedono le risorse. Questa è un'astrazione molto ricorrente, che prende il nome di paradigma di programmazione. Si divide chi utilizza delle funzioni da chi le implementa. In c++, questa cosa si concretizza avendo da una parte una libreria di funzioni, e dall'altra il main, che utilizzerà le funzioni messe a disposizione dalla libreria. Si deve trovare a questo punto un modo affinché i vari moduli possano comunicare tra di loro. Il punto di partenza è quello di avere un file con estensione .cpp in cui si trova il main, un altro sempre .cpp in cui si realizzano le funzioni che andranno a costituire una libreria. Tuttavia, se teniamo le cose in questo modo, il compilatore, quando va a compilare il main, non sa dell'esistenza delle funzioni, per cui darà errore di compilazione ogni qual volta che saranno chiamate. Si deve allora trovare un modo affiche la compilazione di main.cpp vada a buon fine. La compilazione non coincide con la creazione di un file eseguibile. In questa fase, il codice viene tradotto in codice oggetto in un file con estensione .o, molto di basso livello ma che è ancora rilocabile. Questo file in linguaggio oggetto è intermedio tra il linguaggio c++ e il codice macchina. Affinché si crei un eseguibile, deve entrare in gioco il linker, che, a partire da vari file oggetto, ne verifica la coerenza reciproca e crea un file .exe. Se nel main invoco una funzione che è stata implementata in una libreria, è grazie al linker che sono sicuro che tutto avvenga coerentemente. Questa cosa tuttavia non conclude la faccenda. Infatti, il compilatore può non conoscere l'implementazione di una funzione di libreria quando va a compilare il main, ma deve almeno essere sicuro della sua esistenza. C'è bisogno di far conoscere al main i prototipi di tutte quelle funzioni che saranno poi implementate nella libreria. Sarà poi il linker che effettuerà il collegamento. Affinché si abbia questa cosa, si procede nel seguente modo.

Si divide la libreria in due file. Uno di questi rappresenta l'interfaccia, uno l'implementazione. Nell'interfaccia si mette solamente i prototipi delle funzioni, e si avrà un file .h, mentre nell'altro file

si realizza l'implementazione delle funzioni. A questo punto, basterebbe che sia il file in cui si implementano le funzioni, sia il main fossero a conoscenza di tutti i prototipi delle funzioni, il primo affinché possano essere definite, il secondo perché possano essere usate. Una soluzione sarebbe quello di fare una sorta di copia incolla, il che è possibile tramite la direttiva al preprocessore #include.
Il preprocessore fa una prima passata del file da compilare e, al posto di #include "libreria.h" sostituisce tutto il contenuto del file libreria.h presente nella stessa cartella del file che si dovrà compilare. Questo è un file provvisorio, che sarà poi utilizzato dal compilatore. Con questa direttiva, nel main avrò tutti i prototipi che mi servono.
Questa cosa è comoda sotto vari punti di vista. Se, ad esempio, ho un file main.o e voglio modificare libreria.o, lo posso fare ricompilando solo i file legati alla libreria nella misura in cui non vado a modificare in alcun modo l'interfaccia, ma solo l'implementazione delle varie funzioni(questo per il meccanismo dell'information hiding, visto che al cliente non interessa l'implementazione interna del servitore). Allo stesso modo, posso compilare il main lasciando inalterato il file libreria.o, che magari risulta molto corposo e richiederebbe molto tempo per essere compilato.
Per fare lettura da tastiera e scrittura su schermo, si usa la libreria iostream. Scrivendo #include<iostream>, inserisco nel main tutte le dichiarazioni delle varie funzioni, poi il linker unisce il file compilato del main con iostream.o precompilato.

Vediamo altri meccanismi legati all'utilizzo di più file per la realizzazione di un programma. Poniamo di avere file1.cpp e file2.ccp. È possibile condividere una variabile tra i due file? Questo dipende dal collegamento delle varie variabili, che non è la stessa cosa rispetto al campo di visibilità. Una variabile di blocco può essere vista solo dal momento in cui è creata fino all'uscita del blocco, quindi sicuramente non può essere 'esportata' su un altro file. Restano allora le variabili globali allocate sulla memoria statica. Se una variabile globale ha attributo const, non c'è modo affinché possa essere vista da un'altra parte, dato che ha collegamento interno (possono essere utilizzate solo nella propria unità di compilazione). Al contrario, le variabili globali non const hanno di default collegamento estero, per cui possono essere utilizzate anche in altre unità di compilazione per mezzo della parola chiave extern.

```
//File1.cpp
Int a=4;

//File2.cpp
Int a;
```

Se facessi questo, nella seconda unità di compilazione avrei una variabile intera con identificatore a che non ha nulla a che fare con la variabile nella prima unità di compilazione. Oltre a questo, il linker potrebbe dare problemi, perché in due parti diverse del codice ho la stessa variabile. La soluzione è la seguente.

```
//File2.cpp
extern int a;
```

Facendo questo, quando il compilatore incontra questo statement, non alloca memoria per una nuova variabile, ma lascia la cosa in sospeso aspettando che sia il linker ad effettuare il collegamento. Questo è un particolare caso di dichiarazione senza definizinoe per una variabile. Potendo condividere sia delle variabili che delle funzioni, due team, mettendosi d'accordo sul nome

di funzioni/variabili, possono lavorare in modo autonomo e compilare ciascuno il proprio file. Alla fine, è il linker ad effettuare il collegamento concludendo le cose lasciate in sospeso.
Posso anche cambiare le regole di collegamento tra variabili globali. Infatti, dato che una variabile ha di default collegamento globale, da un altro modulo si potrebbe modificarla andando a compromettere l'esecuzione del programma. Per evitare questo, è preferibile modificare il collegamento aggiungendo nella dichiarazione la parola chiave static.
La definizione di una struttura ha collegamento interno, mentre le sue istanze globali hanno collegamento esterno. Questo potrebbe portare guai nel caso in cui da un'altra parte abbia una struct con lo stesso nome ma struttura interna (membri dati) diversa. Per condividere la struct, potrei inserirne la dichiarazione in un file header che poi sarà incluso in ogni unità di compilazione.

Può capitare che tra chiamate successive di una stessa funzione una variabile non debba essere ogni volta deallocata ma se ne debba salvare il valore per la chiamata successiva. Per fare questo, si usa la parola chiave static. La variabile viene inizializzata una solta volta alla prima chiamata della funzione, e ha lo stesso tempo di vita del programma.

Shortcut degli operatori di confronto logico o cortocircuito

```
Int main(){
        Bool a=false;
        int b=2;
        if( a && --b){
                cout << "Primo if " << endl;
        }
        Cout << a<< ' ' << b << endl;
        a= true;
        if( a && --b){
                cout << "Secondo if " << endl;
        }
        Cout << a<< ' ' << b << endl;
        If(a && --b){
                Cout << "Terzo if" << endl;
        }
}
```

L'operatore di and logico valuta prima l'espressione di sinistra. Se è vera, va a valutare l'espressione di destra e restituisce il valore di quest'ultima, ma se e valse restituisce direttamente false senza valutare l'espressione di destra.

Output:
0 2
1 1
1 0

Nell'ultimo caso, a è true, quindi si passa a valutare la seconda espressione. Per come funzione l'operatore di pre-incremento, prima viene decrementato b, poi ne viene restituito il valore che, in questo caso, è 0, quindi false. Non si entra nel corpo del terzo if, ma b viene comunque

decrementato. Questo comportamento dell'operatore && può essere sfruttato. Se c'è da scorrere una lista, si può fare una condizione come

P !=nullptr && p->inf!=a;

Se il comportamento non fosse quello, arriverei ad un punto in cui p punta a nullptr, quindi la prima espressione restituisce false, ma viene comunque eseguita anche la seconda, andando a dereferenziare il puntatore a nullptr. La stessa cosa avviene per l'or logico. Se la prima condizione è vera, viene restituita senza valutare la seconda, altrimenti viene restituito il valore della seconda.

Un esempio di programmazione stile C: la pila.

Prima dell'utilizzo delle classi in programmazione, il modo migliore per risolvere una serie di problemi era quello di creare una struttura dati e poi tutta una serie di funzioni globali per lavorare su di essa. Con le funzioni globali, si andava ad aggiornare l'istanza di quella struttura dati. Questo poteva avvenire per mezzo del paradigma di programmazione client/server, per cui si andava a descrivere la struttura e le funzioni in una certa unità di compilazione che veniva poi linkata con l'utilizzatore. Si ha ad esempio una struct nell'header file, così che nel main se ne possano creare delle istanze aggiornate con le funzioni che hanno i prototipi nello stesso header file. Il fatto che tutti i prototipi siano isolati in un file a sé è fondamentale. Innanzitutto, ci permette di usare la direttiva #include affinché il compilatore non dia problemi senza tuttavia rendere del tutto nota la struttura interna delle funzioni. Oltre a ciò, ci permette di realizzare una funzione f1 che chiami f2 e una f2 che chiami f1 senza alcun problema, visto che il compilatore, in fase di chiamata, sa già dell'esistenza di entrambe. Vediamo adesso cosa è una pila

Una pila è una struttura dati di un certo tipo (nel nostro caso di interi) di dimensione prefissata. L'idea della pila è data dall'espressione LIFO (Last In First Out): si estrae l'ultimo elemento inserito. Per inserire ed estrarre dalla pila, ho bisogno di due funzioni: una pop per estrarre ed una push per inserire. Vediamo come è fatta l'interfaccia.

```
//Pila.h

Const int DIM=100;
struct pila{
        int stack[DIM];
        int top;

        Bool push(pila&, int);
        Bool pop(pila&, int);
        void inip(pila&);
        Bool stampa(const pila&);
        Bool isEmpty(const pila&);
        Bool isFull(const pila&);
        }
```

In ogni funzione l'istanza della pila è passata per riferimento. Questo perché in ogni caso la struttura dati è di grandi dimensioni, e il passaggio per valore porterebbe ad un ampio uso eccessivo dello stack. Per evitare problemi, quando la struttura dati non deve essere modificata, il riferimento è dichiarato costante. Questo è il file header, che mi permette di far comunicare pila.cpp e main.cpp. Il lato client conosce solo la struttura dell' interfaccia ed è del tutto inconsapevole di come le singole funzioni siano internamente organizzate. Questo è un esempio di programmazione in stile C, in cui si aveva una struttura dati che manteneva lo stato dell'oggetto e delle funzioni globali che vi potevano agire. Con questa struttura, io alloco un vettore di dimensione massima prefissata, ed insieme ho un 'puntatore' che mi indica in quale posizione è l'ultimo elemento inserito. Se top vale -1, la pila è vuota, mentre se vale DIM-1 allora è piena. Uno dei problemi che si pone è che la

struttura dati, per quanto abbia un'organizzazione interna, una volta creata non è pronta per essere usata: è necessario che top sia posto a -1 per indicare che la pila è vuota. Per ovviare a questo problema, si crea la funzione inip, il cui compito è proprio quello di porre in uno stato consistente la pila. Appena la struttura dati è posta in uno stato consistente, anche tutte le funzioni che vi interverranno la lasceranno in uno stato consistente. Le funzioni isEmpty e isFull sono funzioni di utilità, che permettono di sapere subito se la pila è piena o vuota. Notare che in questo modo è praticamente impossibile precludere all'utilizzatore l'utilizzo di queste due funzioni di utilità, anche se nascono per essere usate solo come funzione di utilità per la gestione interna della pila.
Un'altra struttura dati tipica dell'implementazione alla C è la coda, che segue la politica del FIFO(First in, first out), in cui si estrae prima il primo elemento inserito.

Il namespace

Se utilizzassi nello stesso programma sia la pila che la coda, entrambe avrebbero, nei relativi file .h una variabile dal nome DIM. Oltre a comportare un errore in fase di compilazione, questa cosa impedisce al client di riferirsi in modo specifico ad una delle due variabili, perché non vi è alcun discriminante. Un modo per risolvere questo conflitto, e che è stato adottato per decadi, è quello di porre, insieme a ciascun identificatore, il nome della libreria a cui si riferisce. Avremo quindi qualcosa come pila_DIM e coda_DIM. Tuttavia, se scegliessi di cambiare il nome di una libreria, dovrei andare a modificare tutti gli identificatori e i punti in cui sono utilizzati. Per risolvere questo problema, si introduce in C++ lo spazio di nomi.

```
Namespace uno{
        Struct st{
                Int a;
                double b;
        };
        void ff (int a) {…}
}
```

```
Namespace due{
        Struct st{
                Int a;
                double b;
        };

}
```

Ho due strutture dati con lo stesso identificatore ma che sono in due diversi spazi di nomi. A questo punto, nel main è possibile scrivere

```
Int main(){
        uno::st struttura1;
        using namespace due;
        st struttura2; //Avendo usato using, automaticamente ci si riferisce al namespace due.
}
```

L'operatore :: è detto di risoluzione di visibilità, e mi permette di riferirmi ad uno specifico spazio di nomi. All'inizio di ogni programma usiamo using namespace std; perché l'istanza della classe stream sono inseriti all'interno di questo spazio di nomi. Questa è stata una scelta dovuta al fatto che i namespace sono stati introdotti dopo la creazione del linguaggio, e tutto quello che c'era di standard è stato buttato dentro questo namespace predefinito dall'identificatore std. Esiste poi uno spazio di nomi senza nome dove finiscono tutte le variabili e le funzioni dichiarate al di fuori dai blocchi. Questo perché tutti gli identificatori (di funzioni o di variabili) devono appartenere ad uno spazio di nomi. In questo modo si ha la retro-compatibilità con tutti quei programmi che, non usando uno spazio di nomi, non avevano posto le variabili globali all'interno di uno di questi.

Introduzione alle classi

Abbiamo visto cosa significasse in C lavorare con una struttura dati con il paradigma client/server e con funzioni globali che mantenevano consistente questa struttura. Tuttavia, si possono osservare tutta una serie di problemi non indifferenti. È necessario che, dopo ogni creazione di un'istanza di pila, la struttura dati sia posta in uno stato consistente invocando su di essa la funzione inip. Se questo non avviene, tutte le funzioni che lavorano sulla pila si comportano in modo abbastanza aleatorio, in quanto potrebbero andare a lavorare su chissà quale regione di memoria. Se si fosse allocato la pila sullo heap, sarebbe servita una funzione distruggiPila da invocare ogni volta che quell'istanza non serviva più, e anche questa cosa poteva sfuggire a colui che utilizza l'istanza della pila. Il fatto che è ci sono molte cosse di cui ricordarsi, altrimenti il codice perde di determinismo.
Un'altra cosa non troppo felice è che posso andare direttamente nel main a modificare l'istanza della pila senza necessariamente dovermi rifare alle funzioni. L'utilizzatore può compromettere la pila portandola ad uno stato non consistente. L'idea è quella per cui l'utilizzatore non dovrebbe conoscere ed avere accesso alla struttura della pila. Si deve astrarre la struttura dati, in modo che solo un gruppo scelto di funzioni possano andare a modificarla. Questo è il principio dell'INFORMATION HIDING. Questo principio cerca di risolvere un problema di sicurezza, per cui in nessun modo l'utilizzatore sarà capace di porre la struttura dati in uno stato non più consistente.
Un altro problema delle strutture dati alla C è che potrei creare una funzione non di libreria che mi va a modificare l'istanza della struttura dati. Per andare incontro a questa forma di astrazione, in C++ sono state introdotte le classi.

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

Const int DIM=5;

Class pila{
        Int top;
        Int stack[DIM];
Public:
        Bool (push int);
        Bool pop( int);
        void inip();
        Bool stampa() const ;
        Bool isEmpty() const ;
        Bool isFull() const ;
};
```

In una classe, tutto quello che non sta nella parte public è considerato privato. Non solo posso definire una serie di informazioni associate al tipo pila, ma posso anche dichiarare tutte solo le funzioni che vanno ad agire su tale struttura dati. Dato che la classe occupa memoria e definisce anche quali operazioni la possono coinvolgere, questo rappresenta un vero tipo definito dall'utente, non come la struct che rappresentava solo un surrogato di tipi. Le funzioni dichiarate nella parte pubblica possono andare a lavorare sui membri della parte privata. Nelle funzioni membro di una classe si nota che non si passa l'istanza della classe come argomento.
Un tipo ha delle caratteristiche ben precise: ha una dimensione in memoria, nel caso data dalla somma delle dimensioni degli elementi che lo compongono, definisce che valori possono assumere le variabili che lo compongono e che operazioni si possono effettuare sulle variabili di quel tipo. Quando creo un'istanza di una classe, le funzioni non occupano memoria. Le funzioni membro, così come quelle globali dichiarate nel file header, andranno poi implementate. Ma vediamo come si utilizza una classe.

```
//main.cpp
#include "pila.h"
```

```
Int main(){
        Pila st;
        St. inip();
}
```

La funzioni inip si chiama con l'operatore di selettore di membro su una certa istanza di una classe. In questo modo, supero le limitazioni del linguaggio che mi dicevano i soli tipi disponibili al programmatore. A questo punto, sono poche le cose che non si possono customizzare(ad esempio, non posso introdurre nuovi simboli che facciano da operatori). Una classe è un tipo di dato astratto che comprende una serie di variabili interne eventualmente private e tutte le funzioni che possono agire su di esse.
Con la pila alla C, potevo modificare a mio piacimento il puntatore top e gli elementi della pila. In questo modo, posso accedere alle varie cose solo con le funzioni membro. L'idea è quella per cui l'utilizzatore debba conoscere il meno possibile della struttura interna, mentre rendo disponili solo il modo con cui è possibile utilizzare questo tipo di dato.
Un'altra cosa che accadeva è che tutte le volte che creavo un'istanza pila mi dovevo ricordare di porla in uno stato consistente usando la funzione inip. Nelle classi, io posso aggiungere tra le funzioni membro una senza valore di ritorno e con lo stesso nome della classe. Questo è un costruttore, una funzione che, se esiste, viene invocata ogni volta che si crea un'istanza di quella classe. Grazie al costruttore, ho risolto il problema della messa in stato consistente della struttura dati. Anche le classi, per tutta la serie di problemi legate al compilatore, è bene che siano definite mediante il paradigma cliente/servitore.

```
//pila.cpp
#include "pila.h"

pila::pila(){
   top=-1;
}
bool pila::empty(){
return top==-1;
}

bool pila::full(){
   return top==DIM-1;
}
```

L'obbiettivo della nostra analisi delle classi è quello di arrivare a definire un tipo che sembri essere del tutto nativo del linguaggio.

Quando si implementa una funzione membro, ci si riferisce ad un'istanza generica di una classe, che poi diventa la specifica istanza su cui è applicata la funzione. La differenza per il compilatore non è tanta rispetto alla struttura dati gestita con le funzioni globali.

Argomenti di default per le funzioni

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

Void fun(int, double, int);

Int main(){
        Fun(5,6.7,3);
```

}

Void fun(int a, double b, int c=3){ … }
Si possono assegnare dei valori di default agli argomenti delle funzioni. Questo è bene farlo nell'implementazione della funzione, non nella dichiarazione. L'unico vincolo è che se assegno un valore di default ad un certo argomento, allora lo devo assegnare anche a tutti gli argomenti successivi. Se abbiamo previsto degli argomenti di default, nel main posso invocare la funzione anche solo con un numero limitato di argomenti: se implementati infatti, ai restanti sarà assegnato il valore di default. È il compilatore che verifica che la chiamata sia stata eseguita in modo corretto

Overloading di funzioni

Un'altra cosa che riguarda le signatura delle funzioni è l'Overloading: posso definire due funzioni che abbiano lo stesso identificatore ma diversi tipi come argomento o numero di argomenti differente. Ad esempio, posso definire una funzione funzione radice quadrata differente per interi e double. A seconda degli argomenti passati in fase di chiamata, è il compilatore che sceglie quale delle due versioni chiamare. Notiamo che ci deve essere per il compilatore sempre la possibilità di discriminare quale delle due funzioni si vuole invocare. Poniamo ad esempio di scrivere

Double sqrt(int a, int b=0);
Double sqrt(int c);

…

Sqrt(3);

IL compilatore non riesce a capire se si tratta della funzione con un solo argomento di tipo intero o due argomenti dove al secondo è assegnato un valore di default. Il tipo di ritorno non va bene per disambiguare tra le due chiamate, dato che, non si capisce in fase di chiamata che tipo di valore di ritorno si vuole. La signatura va dall'inizio dell'identificatore in poi.

Il tipo enumerazione

CI sono delle situazioni in cui si ha una variabile che può assumere un numero discreto di valori. Pensiamo ad esempio ad un semaforo che può avere come valori di stato solo Verde, Giallo o Rosso. Oppure i giorni della settimana, che sono solo sette. Cose del genere risultano molto frequente far si che una variabile posso assumere solo un certo numeri di valori può essere importante. Non vale la pena creare strutture complessa per una variabile di questo tipo. Si usa allora il tipo enumerato. Sono variabili che di per sé non hanno alcun valore se non l'etichetta che si assegna loro. All'interno del codice, ci permette di capire in quali dei casi discreti ci troviamo. Con uno switch, si gestisce bene questa situazione. La parola chiave del tipo enumerato è enum. Si da il nome al tipo, poi si specificano le varie etichette che quella variabile può assumere.

enum Giorni{LUN,MAR,MER,GIO,VEN,SAB,DOM}:

Una variabile di questo tipo può assumere solo questi sette valori, rappresentati da queste etichette. Un utilizzo tipico è il seguente:

```
switch (oggi){
        case LUN:
        case MAR:
        …
        case DOM:
}
```

Una tipica incomprensione è la seguente: cosa succede se faccio una cout di un tipo enumerato? Ho a video un valore intero. Sembra (come effettivamente è) che il linguaggio crei delle costanti intere ciascuna delle quali ha un valore intero. La prima etichetta ha come valore 0, poi 1 e così via.
È come se ci fosse scritto:

```
const int LUN=0;
const int MAR=1;
…
```

Proprio per questa struttura interna, io posso creare una variabile di tipo intero e assegnargli una delle etichette del tipo enumerato. Tuttavia, non si può assegnare un intero ad un tipo enumerato? Perché? Perché volendo potrei anche cambiare gli interi associati a tutti o anche ad una sola delle etichette del tipo enumerato. Quindi a priori non posso sapere se quel dato intero è associato ad una delle etichette del tipo enumerato. Il tipo enumerato è allora un vero tipo, dato che una sua istanza ha un certo spazio in memoria (quello di un intero), può assumere una serie di valori fissati e ci sono solo una certa gamma di operazioni che si possono fare con questo tipo.

Alcuni aspetti delle classi

Poniamo di fare la seguente funzione globale:

```
double modulo (complesso c){
        double m=sqrt(c.reale()*c.reale()+c.immaginario()*c.immaginario());
}
```

Questa funzione non viola il principio dell'information hiding, dato che non accede direttamente alla parte privata dell'istanza di complesso, ma usa delle funzioni membro della classe stessa. Questo è un modo per estendere le potenzialità della classe nel caso in cui non ci sia la possibilità di modificarla direttamente.
Così come accade per i tipi standard del linguaggio, anche l'istanza di una classe possono essere passate come argomento di una funzione o essere restituite come valore di ritorno.

```
Complesso coniugato(Complesso c){
        Return Complesso(c.reale(),-c.immaginario());
}
```

Quando entro nella funzione, mi creo una variabile senza nome per il valore di ritorno e una variabile formale copia di quella passata come parametro attuale. A quel punto, copio nel valore di ritorno un'istanza che ha la stessa parte reale di quella passata come argomento ma parte immaginaria

opposta. Nel main, il valore di ritorno può essere assegnato ad un'istanza della classe complesso. Nel fare questo, si usa l'operatore = definito di default per le classi, tale per cui si ha una copia dei vari membri dati dell'istanza.
L'information hiding ci permette di modificare la struttura interna della classe senza minimamente modificare l'utilizzo della stessa. Se ad un certo punto invece che avere un numero complesso definito da parte reale e immaginaria ne volessi uno definito da modulo e argomento, mi basta modificare tutte le funzioni in modo che facciano la stessa cosa, ma per colui che usa l'istanza della classe non vi è alcuna conseguenza.

Quando creo una classe, è necessario andare ad inizializzare una serie di valori tra i membri dati, in modo che ogni istanza sia in uno stato consistente (è il caso in cui per la pila si metteva il puntatore top a -1). Per questo motivo esiste il costruttore.
In generale, quando suo la dot notation per invocare una funzione membro, il compilatore sostituisce la chiamata con un'invocazione della funzione stessa avente come primo argomento l'indirizzo dell'istanza su cui è stata chiamata la classe. Tutte le funzioni membro hanno quindi come primo argomento implicito l'indirizzo ad un'istanza della classe stessa.

p.inip() ↔ Pila::inip(&p);

Ma perché l'indirizzo? Perché quando si ha una funzione membro, noi usiamo il generico membro della classe, ma in realtà il compilatore sostituisce ogni riferimento al membro dati con this-> membro. Ad esempio

```
void pila::inip(){
        This->top=-1;
}
```

L'istanza su cui è chiamata la funzione prende il nome di *this, o meglio, si ha un puntatore con identificatore this che punta a quella specifica istanza, e ogni riferimento a quell'istanza viene fatto con l'operatore freccia. this è un puntatore costante perché, in una certa funzione, deve sempre riferirsi all'istanza su cui è stata chiamata la funzione membro. Allora, creare una funzione globale che opera su un'istanza di una struct o creare una classe e le varie funzioni membro è la stessa cosa, dato che anche le funzioni membro hanno un argomento fittizio.

```
Void pila::inip(pila* const this){
        this->top=-1;
}
```

Io passo l'indirizzo dell'istanza che deve essere modificata, e il puntatore che si riferisce a questa prende il nome di this. Stampando a video this con una cout, ottengo lo stesso indirizzo di memoria dell'oggetto su cui era stata chiamata la funzione. Poniamo di avere una funzione che scala di n sia la parte reale che quella immaginaria di una certa istanza, e poniamo che io voglia poter concatenare la chiamata alla funzione:

c.scala(3).scala(2);

Dato che l'operatore di selettore di membro è associativo da sinistra verso destra, prima si ha la scala di tre, poi di due. Affinché le cose vadano come ci si aspetta, è necessario che la prima scala

abbia come valore di ritorno l'istanza della classe, in modo che anche la seconda funzione possa fare quello che deve fare. Un modo potrebbe essere :

```
complesso scala(double s){
        …
        return *this;
}
```

Vediamo cosa succede sullo Stack: Io invoco la funzione scala la prima volta su c. Mi si crea una variabile senza nome per il valore di ritorno e una di tipo double per l'argomento spassato per valore. La funzione, fatto il suo lavoro, copia nel valore di ritorno il contenuto di *this (costruttore di copia di default), dealloca tutto quello che è stato creato nella funzione e chiama la funzione scala sull'oggetto creato come copia nel valore di ritorno. Questo significa che la scala non ha alcun effetto sull'istanza di partenza. Per far si che la funzione lavori sempre sullo stesso oggetto, devo passare come valore di ritorno un riferimento all'istanza della classe. In questo modo, la possibilità di concatenare le chiamate di una funzione rende tutto più intuitivo.

Il costruttore è una funzione che è invocata in automatico ogni volta che si crea un'istanza di quella classe. È una funzione senza valore di ritorno con lo stesso nome della classe. Essendo una funzione, può avere degli argomenti passati, ma, se quello è l'unico costruttore che ho, tutte le volta che creo un oggetto è passargli sempre lo stesso numero/tipo di parametri. Se io creo un costruttore della classe complesso che ha come parametri due reali, sto dicendo che non si può creare un complesso senza avere due valori iniziali con cui inizializzarli. A questo punto possiamo sfruttare l'Overloading delle funzioni per creare un secondo costruttore senza parametri, che inizializza di default la parte reale e quella immaginaria di un'istanza a zero. Un'alternativa è avere un unico costruttore con dei valori di default come argomenti. In questo modo, sarà impossibile avere un'istanza di una classe che non sia inizializzata.

La memoria dinamica nelle classi

Vogliamo creare la classe stringa, in modo da poter trattare le stringhe come un vero e proprio tipo. Uno dei problemi principali che possono venire è che non sappiamo come gestire la dimensione della stringa che può essere contenuta. Non possiamo ad esempio scrivere tra i membri dati della classe str[1000 +1], perché in questo modo da una parte allocherei 1001 bytes per ogni stringa anche se magari ne basterebbero due, dall'altra non potrei superare questa dimensione. Allo stesso tempo, non posso neanche inserire il numero di elementi a seconda delle necessità a runtime. Questo perché IL COMPILATORE DEVE SAPERE IN FASE DI COMPILAZIONE LA DIMENSIONE (SIZEOF) DI OGNI ISTANZA DI TIPO CLASSE. Servirebbe allora una variabile globale, ma ci sarebbero gli stessi problemi. La soluzione allora è avere tra i membri dato un puntatore a carattere. In questo modo, ogni istanza di una classe occuperà sullo Stack solo un byte, quello del puntatore a carattere tra i membri dato. A questo punto, nel costruttore posso inizializzare il costruttore a null, in modo da rendere consistente l'istanza. Poi, non posso fare una strcpy di una stringa costante a questo puntatore, dato che sta puntando o a null o ad un'area di memoria che non è propria del programma, e si potrebbe fare solo danni. L'idea è quella di allocare sullo Heap una stringa e far si che il valore di ritorno sia salvato nel puntatore a carattere. Così, posso creare la stringa di dimensione strlen(str)+1, senza sprecare memoria ma usandone il minimo indispensabile. Il puntatore occupa una sezione sullo Stack, la stringa sullo Heap.

Avere un membro dati che punta a qualcosa sullo Heap può essere pericoloso nel caso in cui accade che si esca dal blocco in cui è stata dichiarata l'istanza di una classe e questa venga deallocata sullo Stack. Infatti, sullo Heap si continua ad avere un'area di memoria occupata dalla stringa. Come risolvere? Esiste la possibilità di dichiarare per una classe un distruttore, una funzione membro da un nome particolare senza argomenti ne valore di ritorno che è chiamato ogni volta che un'istanza viene deallocata. Nel distruttore si può effettuare le delete, in modo che tutto possa funzionare bene senza lasciare buchi in memoria. Lo Heap funziona anche per le classi, nel senso che posso creare un'istanza sullo Heap ed inizializzarla in fare di allocazione.

```
Int main(){
        Stringa * ps;
        ps= new stringa("stringa sullo Heap");
}
```

Questo perché anche delle istanze dei tipi fondamentali possono essere allocate sullo Heap, e tutto funziona coerentemente.

Adesso però, vanno risolti alcuni problemi. Per le classi, esiste un operatore di assegnamento definito di default che permette di effettuare una copia membro a membro dei campi dati della classe. Questo può essere utile per la classe complesso, ma risulta nocivo per la classe stringa.

```
Int main(){
        Stringa s1("oggi");
        {
                Stringa s2=s1;
        }
}
```

Dentro al secondo blocco il puntatore di s2 punta sullo Heap alla stessa stringa di s1. Quando si esce dal blocco però, viene chiamato il distruttore proprio su quell'area di memoria, portando ad uno stato non più consistente la prima stringa. Subito dopo l'inizializzazione, il puntatore di s2 sullo Stack punta al primo carattere della stringa "Oggi\0" sullo Heap.
D'altronde, non ci si poteva aspettare altro, perché questo è il comportamento standard dell'assegnamento per le classi. Un problema del tutto equivalente è il costruttore di copia. Anche questo esiste di default per le classi, ma a noi ne serve uno che ci permetta di allocare nuova memoria per ciascuna nuova istanza e di salvarvi il contenuto dell'area di memoria sullo Heap dell'istanza di partenza. Ma come si può fare per modificare il costruttore di copia?

Sicuramente, all'interno della sua dichiarazione, non si può passare la stringa per valore. Infatti, in questo caso si avrebbe una contraddizione: quando si passa una stringa per valore si chiama il costruttore di copia perché il contenuto dell'argomento attuale sia copiato in quello formale. Ma noi stiamo usando questo argomento proprio per definire il costruttore di copia, quindi cadremmo in contraddizione. Per questo motivo, è necessario passare l'argomento per riferimento, magari con l'attributo const.

Il costruttore di copia così definito viene invocato in tre casi: quando si inizializza un'istanza con un'altra precedentemente definita, quando si passa un argomento per valore e quando si ha come

valore di ritorno per una funzione un oggetto classe (si ha la copia dell'istanza in quella variabile fittizia creata appena si entra nel blocco della funzione).

IL COSTRUTTORE DI COPIA HA BISOGNO DI UN PARAMETRO PASSATO PER RIFERIMENTO.

La ricorsine

Un algoritmo ricorsivo è un algoritmo definito mediante sé stesso. Una funzione ricorsiva usa una copia di sé stessa con degli argomenti diversi. Il fattoriale ad esempio è espresso in termini di sé stesso:

n! = n * (n-1) * (n-2) * … * 2 * 1

Se volessimo scrivere un programma che calcola il fattoriale di un numero:

```
unsigned int fatt (unsigned int a){
        if(a==0)
                return 1;
        else
                return a*fatt(a-1);
}
```

Se invoco fatt(3), la funzione viene chiamata tre volte consecutivamente, in modo che i vari valori di return possano essere poi utilizzati insieme. La ricorsione ha sempre una struttura generale in cu si ha un caso base (il caso in cui non avviene la ricorsione) e i casi in cui avviene la ricorsione. Questo è chiamato passo ricorsivo. C'è sempre anche un parametro di controllo che viene utilizzato affinché si possano gestire le diverse chiamate della funzione. Le funzioni ricorsive risultano sicuramente pratiche, ma non efficienti se confrontate alla rispettiva formula iterativa.
Un'applicazione sulle liste è la seguente:

```
void insfonndo(elem*&testa, int valore){
        If(testa==nullptr){
                Testa = new elem;
                testa->succ=nullptr;
                resta->numero=valore;
        }
        Else
                insfonndo(testa->succ, int valore){
}
```

Ridefinizione degli operatori per le classi

La funzione somma nella classe complesso è una funzione globale che ritorna per valore un'istanza della classe complesso ottenuta dalla somma dei membri dato di quelle passate come argomento per valore. È possibile far si che al posto di una funzione di questo tipo si utilizzi l'operatore somma che viene usato anche per i tipi fondamentali? Certo! Per farlo, bisogna ridefinire la funzione operator+ come funzione globale, e la struttura poi è la stessa della somma.

```
Complesso operator+(complesso a, complesso b){
        Complesso c(a.reale()+b.reale(),a.immaginario()+b.immaginario();
        return c;
}
```

Questa cosa funziona perché alla scrittura c1 + c2 il compilatore sostituisce operator+(c1,c2);. Appena il compilatore si accorge che esiste una funzione globale con cui è possibile sostituire quella scrittura, lo fa. Questa cosa è comoda nella misura in cui si rende l'idea che il tipo classe che abbiamo creato sia quasi un tipo fondamentale, su cui possiamo lavorare con gli stessi operatori.

Ciò detto, esiste anche un altro modo per ridefinire la somma tra complessi, ed è quella di dichiarare la funzione operator+ come funzione membro. In questo modo, oltre al vantaggio della facilità di lettura, si ha anche che la funzione può accedere anche ai membri dato privati, avendo gli stessi privilegi delle funzioni membro dell'accesso ai campi dati delle istanze. In questo modo, c1 + c2 viene sostituito con c1.operator+(c2) che a sua volta corrisponde a opeator+(&c1,c2);

Esistono allora due modi per implementare la ridefinizione di un operatore di questo tipo: come funzione globale, che ha due argomenti espliciti di tipo complesso e come funzione membro, che ha un argomento esplicito e uno implicito, corrispondente all'indirizzo dell'istanza su cui viene chiamata la funzione. Le due cose sono del tutto equivalente, soprattutto perché a livello visivo non si ha alcuna differenza tra la prima e la seconda implementazione. Ci sono alcune limitazioni della ridefinizione degli operatori: non posso cambiare l'arieta e l'associatività; non posso aggiungere nuovi simboli; se ridefinisco un operatore, almeno uno degli argomenti deve essere un tipo definito dall'utente, non posso ridefinire gli operatori per i tipi fondamentali.

Nonostante ci sia per le classi un operatore di assegnamento di default, è possibile ridefinirlo per le classi che utilizzano la memoria dinamica. Prendiamo il seguente caso:

```
stringa s1("oggi");
stringa s2=s1;
stringa s3("Domani");
s3=s1;
..
```

Facendo questa cosa, I puntatori str delle varie istanze stanno puntando alla stessa area di memoria, ma non è proprio quello che ci serve, visto che l'assegnamento dovrebbe creare delle copie vere e proprie, autonome le une dalle altre. In più, si perde l'indirizzo della stringa "Domani" sullo Heap.

Per evitare questi problemi, si hanno due tipi di soluzione. La prima consiste nel ridefinire l'operatore di assegnamento come funzione membro nella parte privata, in modo che non possa essere utilizzato. Questo prende il nome di mascheramento. Allo stesso modo, si può ridefinire in modo tale che la situazione sullo Heap rimanga stabile e che non ci siamo grossi problemi. Bisogna però fare attenzione ad un paio di cose. Ad esempio, tutti gli operatori possono essere definiti sia come funzione membro che come funzioni globali. Al contrario, l'operatore di assegnamento può essere solo ridefinito come funzione membro. Come fa anche l'assegnamento standard, deve ritornare l'istanza stessa su cui si è effettuato l'assegnamento, quindi un riferimento alla classe e si fa un return *this. Questo è necessario affinché si possa concatenare l'operatore. C'è una cosa

fondamentale a cui fare attenzione. Poniamo di avere un operatore di assegnamento che effettua una delete affinché allo stesso puntatore possa essere assegnata una nuova area di memoria sullo Heap. Cosa succede se facessi s1=s1? La memoria sullo Heap viene cancellata, ma non vi è alcun dato nell'istanza di destra da cui prendere informazioni, visto che anche questo è cancellato. Si deve allora fare una verifica preliminare affinché l'operatore di assegnamento non faccia nulla nel caso in cui si effettua un'operazione del genere. L'idea è quella di confrontare il puntatore this con l'indirizzo dell'istanza usata sulla destra. Se sono uguali, vuol dire che si ha lo stesso oggetto. In generale, non possono effettuare un'operazione di confronto direttamente sulle istanza della classe, perché presumibilmente gli operatori di confronto non sono stati ridefiniti. Questo controllo è utile nella misura in cui si potrebbero anche avere degli alias, e, utilizzando la classe, non ci rendevamo conto di cosa stesse succedendo.

Letterali di tipo classe

Per i tipi fondamentali, esistono le costanti letterali. Ad esempio, 'a' è una costante letterale per il tipo char, 8 per int… Queste sono le costanti letterali, ed esistono ad esempio anche per i tipi derivati: {3,4,5} è una costante letterale per un array di interi di dimensione 3. E se mi servisse una costante letterale per una classe, ad esempio per poter inizializzare direttamente un array di complessi?

Complesso vc[] = {complesso(1,1),complesso(2,1)};

In questo modo ho specificato due constanti letterali per il tipo complesso. Si tratta di una chiamata esplicita al costruttore: il compilatore infatti chiama il costruttore per inizializzare questo oggetto che è la cosa più vicina che c'è ai valori destri per le classi.

Costruttori di conversione

Si parla di costruttore di conversione ogni qual volta che si ha un costruttore con un solo parametro e questo è di tipo fondamentale.

Stringa (char);
Nulla mi vieta di aggiungere questo terzo costruttore nel caso in cui la stringa sia inizializzare solo con un carattere. Questa funzione prende il nome di costruttore di conversione, visto che parto da un tipo fondamentale e da questo creo un tipo derivato. Posso utilizzare questa cosa per creare delle costanti letterali. Ad esempio, se ho ridefinito operator+ come concatenazione per le stringhe, posso fare

S1 = s2 + 'g';

Il tipo fondamentale viene promosso a tipo classe usando il costruttore di conversione. In realtà, questa operazione è effettuata in automatico dal compilatore per i tipi fondamentali: vengono promossi a quelli più consoni a seconda della necessità.

Si può anche avere l'operazione speculare, cioè passare da un'istanza di una certa classe ad un tipo fondamentale. Per questo, si usano gli operatori di conversione int(), bool()…

Poniamo di voler usare un oggetto come condizione per un if. Dentro la condizione dell'if, ci fa un'espressione che possa ritornare un valore di verità, quindi mi serve un operatore che in automatico faccia la conversione da, ad esempio, complesso a bool, in modo coerente a come ho specificato io.

Per dichiarare un operatore di conversione, si usa operator bool() senza valore di ritorno.

Const e static per le classi

Poniamo di avere un'istanza di una classe dichiarata con attributo const. In questo caso, non posso utilizzare le funzioni membro finora dichiarate, perché il compilatore non sa se queste vanno a modificare o meno l'istanza dell'oggetto. Allora, tutte le funzioni membro che non modificano la classe devono essere dichiarate con l'attributo const, in modo che possano essere utilizzate su oggetti costanti. Per fare questo:

```
double reale() const;
```

Si possono dichiarare due funzioni, una con const e uno senza. Questo perché l'attributo rientra nella signatura, quindi entra in gioco l'Overloading delle funzioni.

Lo stesso attributo const potrebbe essere, invece che per l'istanza di una classe, solo per un membro dato. Potrei decidere che per un complesso solo la parte reale possa essere modificata, mentre quella reale debba rimanere costante.

```
Class complesso{
        Private:
                const double im;
                double re;
        public:
                …
};
```

Non ho problema nell'utilizzo della classe fintanto che non si va a modificare il membro dati costanti. Ma come faccio a inizializzarlo? Infatti, essendo il costruttore una funzione membro, rientra nelle funzioni che non possono modificare tale membro, Allo stesso tempo, se si accordasse al costruttore un privilegio speciale, potrei farci tutto quello che voglio, cosa non coerente con quanto detto. Per permettere questa coda, si usa la lista di inizializzazione.

```
Complesso (double r, double i):
Re =r,
{
        …
}
```

Bisogna effettuare l'assegnamento in questa zona a metà tra il prototipo e il costruttore.

Un'altra coda che potrebbe far comodo è avere una variabile comune a tutte le istanza di una classe. Ad esempio, si potrebbero sfruttare per contare momento per momento quante istanze della

relativa classe sono presenti in memoria. In questo modo, il costruttore incrementerebbe l'istanza, il distruttore la decrementerebbe. La cosa più intuitiva è che, essendo questa variabile comune a tutte le istanze della classe, non sarà posizionata sullo Stack con un tempo di vita predeterminato, ma finirà sunna memoria statica, in modo che sia deallocato solo con la fine del programma.
Questa variabile static dovrà essere inizializzata. Per farlo, fuori dai blocchi si scrive:

```
int complesso::contatore=0,

class complesso{

        static contatore;
        ...
}
```

Per stampare il valore di contatore, ho opzioni diverse a seconda che la variabile sia public o private. Nel primo caso, vi posso accedere con la dot notation su un'istanza, nel secondo caso mi serve una funzione che mi ci faccia accedere. Per questo esistono le funzioni statiche nelle classi, che possono essere invocate senza avere necessariamente un istanza su cui essere chiamate e possono accedere ai membri privati dichiarati come static. Le funzioni statiche non hanno il puntatore this, visto che sono chiamate prescindendo dalle singole istanze. Per usare queste funzioni, si usa l'operatore di risoluzione di visibilità.

IOSTREAM e operatori di uscita

Le classi che permettono l'ingresso da tastiera e l'uscita su schermo sono presenti dentro il file iostream. Qui cui sono due classi, una per l'ingresso e una per l'uscita. La classe ostream è quella che si occupa dell'uscita. In generale, questo è reso possibile mediante la ridefinizione degli operator di shift a sinistra per diversi tipi di dato, come interi, double… La classe ostream ha una struttura di questo genere

```
Namesapce std{
        Class ostream{
                Private:
                        buffer char[];
                        //Variabili che mantengono lo stato dell'istanza
                        Ostream(const ostream&);
                        //Il costruttore di copia è mascherato, in modo che non si possa creare una
                        //una nuova istanza della classe complesso. Questo indica che le istanze
                        //di ostream devono essere passare per riferimento.
                Public:
                        ostream & operator<<(int);
                        ostream & operator<<(double);
                        ostream & operator<<(char);
                        ostream & operator<<(const char*);
                        …

        }cout;
```

}

Dopo aver dichiarato la classe ostream, se ne crea un'istanza che ha come identificatore cout. Con il fatto che è mascherato il costruttore di copia, si ha che cout è l'unica istanza che può scrivere su schermo.

È fondamentale studiare i membri dati di ostream. C'è un buffer di caratteri che corrisponde ad una sorta di flusso, uno stream appunto. Tutto quello che viene inserito dentro al buffer finisce poi sullo schermo. Oltre a questo, si hanno una serie di variabili che mantengono lo stato dello stream di uscita. Di fatto, quello che stampo a video sono solo caratteri, nient'altro, e per fare questo si usa l'operatore di shift a sinistra ridefinito per la classe ostream, che prende come argomento implicito l'indirizzo di cout e come secondo argomento quello che voglio andare a stampare, reso possibile grazie al meccanismo dell'Overloading. Ma io posso ridefinire l'operatore di uscita in modo da stampare un'istanza di una classe complesso usando cout << ? L'idea sarebbe quella di aggiungere dentro il file iostream la definizione della funzione operator<<(complesso); ma questo implicherebbe di dover ricompilare il file iostream, molto pesante. Oltretutto, le cose non sono così semplificabili, visto che la classe ostream sfrutta molto il concetto di ereditarietà.

Vediamo allora come ragiona il compilatore per arrivare ad una soluzione. Se incontra cout << c1; si fa due domande: esiste una funzione globale che ridefinisce lo shift a sinistra in modo che abbia come primo argomento un riferimento ad un'istanza di ostream e come secondo un'istanza di complesso? Poi: esiste io iostream una funzione membro che ridefinisca l'uscita per la classe complesso? Non potendo seguire la seconda strada, e necessario adottare e seguire la prima. Dobbiamo dichiarare una funzione globale che prenda come primo argomento un riferimento ad un'istanza di ostream, come secondo un'istanza di complesso.

Nel fare questo, l'istanza di ostream si deve passare per riferimento dato che non esiste il costruttore di copia, l'istanza di complesso è bene che sia passata come riferimento costante, in modo tale da non occupare troppa area di memoria sullo Stack ogni volta qq che si va a stampare. Nella funzione, si userà os, che in realtà è un riferimento all'unica istanza di ostream, cioè cout. Come valore di ritorno si avrà un riferimento allo stesso stream, in modo che << possa essere concatenato.

Usando os, sto andando a modificare il buffer così che possa stampare a video. Sarebbe comodo dichiarare la funzione con l'attributo friend dentro la parte pubblica della classe, in modo che si possa accedere ai membri privati. Nella classe ostream è definita anche l'istanza cerr, che convenzionalmente è utilizzata per i messaggi di errore. In particolare, è possibile reindirizzare uno stream su un file, in modo che magari cout stami a video mentre e cerr stampi su file.

La classe per l'ingresso ha una struttura abbastanza simile, e la ridefinizione di >> per la classe complesso ci permetterà di fare cin >>;

Se sono nel main e ho una variabile intera, posso inserire da tastiera un valore che viene salvato dentro questa variabile. Anche qui, come per ostream, l'entrata è gestita con un buffer: finché non si preme invio, i dati non vengono inviati, facendo sì che si possa modificare quanto scritto. Il funzionamento della cin è il seguente: scarta spazi bianchi fin tanto che è possibile, salva i dati successivi nel buffer e, se incontra uno spazio bianco, ignora tutto il resto fino all'invio.

_ \t - 7 4 _ 3 enter

Inserendo questo, nell'intero ci finisce il numero -74. I problemi si hanno se provo ad inserire un carattere al posto di un numero in una variabile di tipo int. Il carattere infatti non viene prelevato, al lettura fallisce e non si sa bene cosa ci sia finito in a. Lo standard infatti non specifica quello che succede se non si riesce ad inserire un intero. Per informare l'utente che qualcosa è andato storto, si pone lo stream in uno stato di errore. Se uno stream viene posto in uno stato di fail, non si potranno effettuare ulteriori letture, ma si può resettare il tutto con la funzione membro clear(). In questo modo, è come se stessi dicendo che hai capito di avere un problema ma che questo non toglie che si possano effettuare ulteriori letture. Ad esempio, poniamo di dover leggere interi da file finché non si incontra un carattere '.'. Se questo accade, la lettura si interrompe e lo stream viene posto in uno stato di file. Adesso, resettiamo il valore e proviamo a leggere il carattere che ci aveva bloccati. Se si tratta di un '.' Allora tutto è avvenuto correttamente, altrimenti, se è un altro carattere, non è successo quello che si voleva.

Nel cin il concetto di errore è molto importante, anche perché questo accade molto più spesso rispetto agli errori di ostream.
Il fatto che ci sia un buffer e non una lettura diretta ci permette di fare questo:

```
cin >> a;
cin >> b;
cin >> c;
```

e inserire 1_2_3
In a finisce il primo numero, ma il resto non è cancellato dal buffer, e viene usato per le letture successive.

Lo stato è implementato con un unsigned int gestito con operatori di or, not e and logico con dei valori standard. Noi però ce li possiamo immaginare come quattro booleani, cioè good, fail, bad e eof (end of file).

Nella parte pubblica, si ha la ridefinizione dello shift a sinistra, in questo modo

```
&cin operator >> (int &);
&cin operator >> (char &);
&cin operator >> (char* &);
```

Ovviamente, per le operazioni di ingresso, gli argomenti devono essere passati per riferimento, visto che possono essere modificati. Come abbiamo ragionato per la classe di uscita, possiamo ridefinire l'ingresso per la classe complesso ridefinendo istream & operator>>(istream&, complesso &);

Poniamo che io ritenga valida solo una scrittura del tipo (33,65.3);

Allora devo fare questo:

```
istream& operator>>(istream& is, complesso & c){
char carattere;
cin>>carattere;
```

```
        double valorere;
        double valorem;
        if(carattere!= '(')
                is.clear(iOS::failbit);
                return is;
        }
        cin>>valorere;
        cin>>carattere;
        if(carattere!=','){
                is.clear(iOS::failbit);
                return is;
        }
        cin >> valorem;
        cin>>carattere;
        if(carattere!=')'){
                is.clear(iOS::failbit);
                return is;
        }
        re=valorere;
        im=valoreim;
        return is;

}
```

Se inserisco un valore che non volevo, posso bloccare l'inserimento mettendo lo stream in uno stato di errore. La funzione clear() è utlizzate per fare queste operazioni: passandole il valore failbit nel namespace ios, possono porre lo stream di ingresso in uno stato di fail.

Lo stato di cin è salvato in un unsigned di 8 bit, quattro di questi usati per lo stato. Abbiamo il failbit, il badbit, il eofbit e il goodbit. Affinché le varie cose possano essere gestite, sono stati posti nel namespace ios questi quattro valori:
failbit è un unsigned che ha tutti 0 tranne quello che ha la posizione del bit fail sullo stato.
goodbit è un unsigned che ha tutti 0 tranne quello che ha la posizione del bit good sullo stato.
badbit è un unsigned che ha tutti 0 tranne quello che ha la posizione del bit bad sullo stato.
eofbit è un unsigned che ha tutti 0 tranne quello che ha la posizione del bit eof sullo stato.
Tutti questi sono dei const unsinged int, e costituiscono una sorta di maschera. La clear, quando viene chiamata con uno di questi come argomento, effettua un or logico tra il proprio stato e il valore passato come argomento. Se non viene passato alcun argomento, la clear usa come maschera un valore che permette di mettere tutti i bit a 0 tranne quello di good.
Per questo motivo, per porre lo stream in uno stato di errore:
cin.clear(ios::failbit);
Esistono anche operatori di conversioni per cin e cout, che mi restituscono true solo se gli stream sono in stati good.

Ingresso e uscita da file

Come le classi per l'entrata e uscita standard, anche quelle che riguardano l'entrata e l'uscita da file sono organizzate con una certa gerarchia. C'è la classe fstream, le cui istanze, a seconda di come vengono inizializzate, permettono sia l'ingresso e l'uscita, ma ci sono anche le più specifiche ifstream e ofstream. Creando una delle istanze di questa classe, si va poi ad aprire il file con la funzione open, per cui si specifica tra virgolette il nome del file che si vuole aprire presente nella cartella del programma. Anche gli stream per l'ingresso e l'uscita da file hanno dei bit di stato, che possono essere usati per effettuare dei controlli sulla corretta apertura del file (magari non avevamo i privilegi necessari per aprire un certo file).

```
Ofstream of;
of.open("uscita.txt");
if(!of.good()){ exit(0); );
```

Con exit, si può interrompere il programma nel punto in cui si vuole. A questo punto, effettuata l'apertura del file, si può usare l'operatore di shift a sinistra per scrivere sul file, sfruttando la codifica ASCII. La differenza con la cout è che si sta andando a scrivere su uno stream differente, tuttavia anch'esso bufferizzato. L'idea che sta alla base di queste operazioni in C++ è quella di immaginare in generale l'ingresso e l'uscita come due flussi. Se il file non è presente nella cartella, esso viene creato.
La stessa idea che sta dietro al cin si può riproporre per la classe dell'ingresso su file.

```
Int main(){
        Ifstream iff;
        iff.open("ingresso.txt");
        if(!iff.good())
                exit(0);
        else {
                int aux;
                while (iff>>aux)
                        cout << "valore letto : " << aux << endl;
                iff.clear();
                char ch; iff >> ch;
                if(ch == '.')
                        cout << "ok";
                else
                        exit(1);
        }
}
```

Dentro il while si procede finché si incontrano valori compatibili con il tipo di aux. Leggendo 7 in codifica ASCII, si converte a complemento a due e poi si salva nella variabile. Così come l'ingresso da tastiera, in questo modo si ignorano eventuali spazi, e lo stream va in uno stato di errore (fail) se viene letto un carattere non compatibile con l'intero. Questo può essere sfruttato per avere un certo tipo di lettura: se vogliamo leggere numeri fino al punto, si può controllare che l'ultimo carattere sia effettivamente un punto. L'operatore di shift a destra/sinistra per i file ritorna un riferimento allo stesso oggetto, in modo che la chiamata possa essere concatenata, mentre si ha anche un operatore di conversione da fstream a booleano che ritorna good solo nel caso in cui i bit di stato sia a good.

Ogni volta che apriamo il file in lettura, stiamo cancellando il contenuto precedente all'interno del file. Questo perché per il processore è molto laborioso aggiungere dati all'inizio di un file. Per risolvere questa cosa, si ha l'opzione per aggiungere i dati sul fondo del file, in append. Per fare questo, si può usare un'istanza di fstream e specificare con open la modalità di apertura del file, se in input, in output o in append. Si può anche aprire il file in modalità diverse nello stesso programma, purché ogni volta ci ricordiamo di chiuderlo con la funzione close();

```
ff.open("aaa.txt", ios::out);
ff.close();
ff.open("aaa.txt",ios::out | ios::app)
```

Tutti questi termini sono degli unsigned presenti nello spazio di nomi ios, che mi permettono di specificare il tipo di apertura che voglio.
Nel caso in cui mi dimenticassi di chiudere un file, questo viene chiuso una volta che, all'uscita dal blocco o dal programma, viene chiamato il distruttore per la classe fstream.

LE UNION

Le union sono molto simili alle struct, anche se presentano delle differenze notevoli.

```
Union                                                                esempio{
        unsigned int u;
        float f;
};

Struct punto{
        Float a;
        Float b;
};

…
Punto a;
esempio b;
Cout << sizeof(a) ; // 8, quattro per ciascun membro della struct
Cout << sizeof(b); // 4
```

Evidentemente c'è qualcosa di diverso rispetto alle struct, visto che la dimensione è diversa pur avendo due membri che in complessivo dovrebbero occupare 4 byte. L'idea è che nella union lo spazio sullo Stack per ciascun membro sia sovrapposto, i soliti 4 byte sono disponibili sia per l'unsigned sia per il float, in modo che i due non possano coesistere contemporaneamente. Ad esempio, noi possiamo fare:

```
e.f=5.6f;
e.u=45u;
```

In questo modo, con il secondo assegnamento, ho cancellato la costante letterale di tipo float specificata per la variabile f della union. Adesso, nella stessa area di memoria, c'è il valore unsinged 45.

Nelle union si sfrutta proprio il fatto che il compilatore sovrapponga le aree di memoria. La dimensione della union è determinata dal membro dati con occupazione di memoria maggiore. Posso usare una union per vedere qual è la rappresentazione interna di un float. Ad esempio, potrei assegnare a f un valore float e poi stampare il valore di quell'area di memoria come se fosse unsigned int, in modo da vedere come viene rappresentato.
Le union si possono usare quando si devono specificare due variabili per un dato che non possono coesistere, in modo che si usi o l'una o l'altra.

Un'altra cosa importante sono le funzioni inline. Il compilatore, se una funzione è dichiarata come inline, non chiama la funzione, ma sostituisce in quello spazio la funzione stessa, in modo che l'esecuzione risulti più veloce. Questo si fa per evitare di perdere tempo in una chiamata di funzione molto breve. Quando una funzione membro è dichiarata all'interno della dichiarazione della classe, essa è automaticamente inline.

Ultimi aspetti delle classi

Si pssono avere delle classi come membro dato di altre classi

```
Class recod{
        Stringa nome;
        stringa cognome;
}
```

In questa classe, i membri dato sono istanze della classe stringa. Questo non crea problemi se non nella misura in cui dobbiamo trovare un modo per chiamare il costruttore della classe stringa in modo che possano essere portate in uno stato consistente. Ad,esempio, il costruttore di record prende due cstringhe costant const char* e const char*. Come facciamo a chiamare il costruttore delle due stringhe in modo da copiarvi questi dati?

Sullo stack stiamo creando un'istanza di record , che a sua volta contiene due oggetti di tipo stringa, ciascuno dei quali allocano un puntatore a carattere. Non potendo chiamare il costruttore con la dot notation, la soluzione è quella di usare la lista di inizialzzaizone, così come si era fatto con i membri dato const.

```
Record::record(const char* n, const char* c):
Nome(n), cognomen(c)
{
        …
}
```

Il vantaggio di questa cosa è che in ogni funzione di record, compreso il costruttore, le due stringhe sono in uno stato consistente.
Se come membro dato delle classia avessi un alias, anch'esso dovrebbe essere inizializzato nella lista di inizializzazione, visto che non si può inizializzare un riferimento non contestualmente alla sua dichiarazione.

Preprocessore

Il preprocessore è il primo programma che viene usato quando si va a creare il file eseguibile. Dopo entra il gioco il compilatore e in ultima istanza il linker per collegare i vari file oggetto. Il preprocessore è chiamato in causa ogni volta che si incontra il simbolo #, il quale indica proprio le direttiva al preprocessore.

#include serve per includere un file. Di fatto, il preprocessore va a ricercare il file o in una cartella di default se seguono parentesi angolari o nella cartella coerrente se seguono doppie virgolette. Esso opera un semplice copia e incola dei dati nel file specificato, in modo che il compilatore abbia, ad esempio, tutte le dichiarazioni delle funzioni a portata di mano in testa al programma.

#define definisce una Macro, ossia associa ad un simbolo una serie di caratteri.

#define CONST = 123

Int k = CONST;

Il preprocessore esamina il file sorgente, e ad ogni occorrenza di CONST vi sostiuisce la serie di caratteri ad essi associati. Questo modo di programmare è tipo del passato, e in C++ si preferiscono le variabili costatni in quanto risultano tipizzate, mentre queste sono solo combinazioni di carattere su cui non può esssere fatto alcun controllo di tipo.

Possiamo con il preprocessore anche permettere l'ignoramento di una certa porzione di testo, tramite le direttive #ifdef e #endif

#ifdef MACRO equivale a dire: se è stata definita la seguente macro, allora considerami questo finché non si incontra una direttiva associata, come un #else, un #elif o un #endif. Questa cosa può essere utilie per permettere il multipiattaforma, dato che le Macro possono essere create annche da riga di comando in fase di compilazione:

Se definisco la costatne LINUX, vuol dire che sono in ambiente linux con tutta una serie di specificità, quindi compila una certa parate di codice, altrimenti, se sono in Windows, compilane un'altra.

Una delle cose più utili a cui serve il preprocessore è il segeunte. Immaginiamo di avere un file .h che vogliamo includere. Se lo includessimo più volte, avremmo degli errori di complazione, visto che per il compilatore vorrebbe dire avere più dichiarazioni delle stesse cose (classi, funzioni…) che andrebbero in conflitto tra loro. Serve allora la possibilità di evitare questo, dicendo:

Se non esiste questo simbolo, allora definiscilo e aggiungi al programma il file header. Se il file header è già stato incluso, allora il simbolo era già stato definito, quindi non fare niente.

```
#ifndef COMPITO_H
#define COMPITO_H
…
#ENDIF
```

Tutte le librerie standard hanno una cosa del genere all''inizio dei rispettivi file header, in modo che una successiva include della stessa sia ignorata.

Ultime cose del programma

Il valore di ritorno di una funzione è come se fosse salvato in una variabile temporanea creata all'inizio del blocco della funzione, prima ancora degli argomenti formali. Posso stampare il valore di ritorno di una funzione, ma in realtà quello che fa il compilatore è salvare questo valore in una variabile temporanea e poi stampare il vaolore di questa varaibile.
Sullo stack, prima viene creata questa varibaile temporanea senza nome, poi gli argomenti formali. In questa, viene copiato il valore che voglio ritornare, e costituisce il tramite fra il resto del programma e il blocco della funzione. Quando questo non serve più, viene deallocato, ed è come se non fosse successo nulla.

Il costruttore di copia viene chiamato dal compilatore in tre casi: quando si ha un inizializzazione di un'istanza di quella classe con argomento un'altra istanza; quando si passa per valore un'istanza di una classe ad una funzione; qundo si ha come valore di ritonro l'istanza di una classe.

Una funzione può avere come valore di ritorno anche un riferimento. In quesot caso, il riferimento rappresenta un valore sinistro, e posso usarlo per modificare il valore di ritorno stesso. Questo ha senso solo nella misura in cui il riferimento si riferisca a qualcosa che esisteva anche prima del blocco della funzione stessa. Se passo un riferimento ad una variabile locale alla funzione stessa, ho errore, visto che la variabile è deallocata all'uscita dal blocco. Un riferimento come valore di ritorno è tipico delle classi quando si vuol permettere la concatenazione di una funzione membro. In quesot caso, la seconda funzione deve lavorare sull'istanza stessa. Posso anche passare un riferiemnto a qualcosa allocato sullo heap: in generale, mi basta che esista anche al di fuori della funzione.