

Facciamo i conti.

Matematica per computer

Articolo e programmi

a cura di

Paolo Ferraresi

Immaginiamo di voler scrivere un programma capace di calcolare l'area di un rettangolo di base 3 e altezza 2, ad un certo punto dovrete scrivere un algoritmo simile al seguente.

```
Area=3*2;  
Scrivi Area;
```

Questo programma riesce a calcolare l'area di questo rettangolo.

Il computer riesce a calcolare l'espressione $3*2$, mette il risultato nella variabile Area e poi scrive a video il risultato dell'espressione.

Se scriviamo un programma come questo, però, non siamo stati molto bravi, serve a poco.

Un programma se possibile dovrebbe risolvere una classe di problemi e mai uno specifico, ecco che adattandolo come quello sotto, è possibile, almeno, calcolare l'area di qualsiasi rettangolo.

```
Leggi base;  
Leggi altezza;  
Area=base*altezza;  
Scrivi Area;
```

Questo programma funziona bene con tutti i rettangoli e ne calcola l'area.

Se volessimo però utilizzare lo stesso programma per il calcolo dell'area di un cerchio, sarebbe necessario adattare la formula, e far rieseguire il codice al calcolatore.

A questo punto non potremmo però dire d'aver fatto un programma per il calcolo delle aree, ma al massimo possiamo dire d'averne fatto uno che esegue una formula, se vogliamo eseguire un'altra formula occorre un altro programma!

Diverso sarebbe se riuscissimo a scrivere un programma, che oltre ai parametri della formula, potesse leggere anche la formula da utilizzare di volta in volta.

```
Leggi Formula;  
Leggi parametro_1,parametro_2,...,parametro_n;  
Risultato=Formula.risultato;  
Scrivi Risultato;
```

A parte il fatto che con questo programma l'utente dovrebbe conoscere la formula che intende applicare, **questo è un programma in grado di calcolare (teoricamente) qualsiasi e formula.**

Lo scopo di questo articolo è capire come fa il computer a calcolare qualsiasi espressione. **Comprendere come fa il calcolatore in queste circostanze ci sarà utile in svariati casi, dove non è accettabile dover modificare il programma per risolvere una qualsiasi espressione**, ci comporteremo infatti come fa lui, cercando sulla base di una stringa di caratteri che descrive una espressione di attivare tutti quei processi di calcolo algoritmico necessari al calcolo dell'espressione medesima, complessa a piacere.

Le formule che scriviamo nei nostri programmi non sono codificate nel compilatore, ma ha gli strumenti per analizzarle e calcolarle, qualsiasi esse siano.

Quando noi cambiamo nel primo programma la formula dell'area, per il compilatore non fa differenza, lui l'analizza e genera un risultato differente per ogni formula che inseriamo, quindi lui sa fare questa operazione.

Allora cerchiamo di capire come fa, per poter usare un sistema simile anche nei nostri programmi e, dotarci quindi di questa potente funzione che ci dà la possibilità di adattare i risultati generati dai nostri programmi sulla base di espressioni algebriche qualsiasi lette da un generico INPUT (sia da tastiera o da disco o per assegnazione diretta non importa).

Ci spingeremo anche oltre e andremo a calcolare il risultato di funzioni.

Per fare questo però è importante capire come fa il computer a calcolare un'espressione; nel nostro ragionamento prenderemo come esempio la seguente espressione $(A+B*C-E)*B-(C+A)/D$.

La scienza della computazione.

Siamo nella prima metà del XX secolo quando alcuni scienziati e grandi matematici, tra cui *Alan Mathison Turing*, *Andreij Markov*, *Alonzo Church*, *Stephen C Kleene*, *Emil Leon Post*, e *John von Neumann*, ma anche altri cominciarono a diffondere oltre alla matematica tradizionale anche **calcolo algoritmico**; c'erano sempre più esperimenti volti alla creazione di macchine per computazione e così i matematici del tempo avevano capito che quello sarebbe stato il futuro, e cominciarono a sperimentare problemi e modelli matematici per la computazione.

Va allo studio del calcolo algoritmico, l'aver messo nero su bianco il procedimento del calcolo d'espressioni algebriche. Infatti qualsiasi studente, anche il meno dotato, fin dalle scuole elementari capisce come funzionano i numeri e come si risolvono semplici espressioni. Dovendo fare i conti con un calcolatore, estremamente stupido, bisognava essere veramente rigorosi senza lasciare proprio nulla all'esperienza o a capacità date per scontate, che un calcolatore non ha.

La matematica ci dice che nell'espressione di cui sopra, è necessario seguire determinate regole per arrivare al risultato, quindi bisogna prima moltiplicare $B \cdot C$, sommarlo ad A , sottrarre E e il tutto va moltiplicato per B , da questo numero che si ottiene bisognerà sottrarre la somma di C e A , divisa per D .

L'osservazione dei matematici portò alla sequenza delle operazioni che un computer dovrebbe compiere che può essere schematizzata da una struttura ad albero dotata di tre proprietà.

1. ogni nodo dell'albero può contenere un operatore oppure una variabile;
2. un nodo contenente un operatore ha i sottoalberi destro e sinistro, che rappresentano gli operandi di sinistra e destra dell'operazione rispettivamente;
3. un nodo contenente una variabile è un nodo terminal;

Come si può vedere, i calcoli immediatamente realizzabili sono quelli che chiudono il triangolo OPERANDO-OPERATORE-

OPERANDO, nell'esempio individuiamo $B \cdot C$ e $C + A$, il risultato di questa operazione

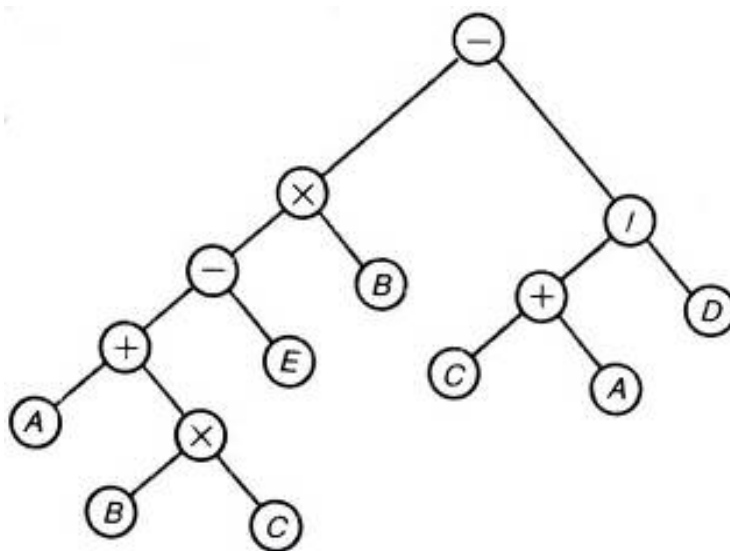
immediatamente realizzabile, va a

sostituire il ramo che prima era indefinito in pratica il risultato di

$B \cdot C$ va a sostituire il segno di moltiplicazione che così diventa l'operando che manca all'operatore $+$ per chiudere il triangolo che consente di ottenere la computazione.

Il calcolatore dovrebbe percorrere l'albero da cima a fondo e viceversa prima di riuscire a calcolare l'esatto risultato dell'espressione.

L'operazione che trasforma un'espressione nella sua forma di scrittura matematica, alla struttura dati rappresentata in figura è estremamente complessa e richiede più scansioni della formula, e questo significa anche un codice molto complesso e un sistema potente per calcolare la formula. Oggi abbiamo tutti almeno un processore Athlon, Pentium4 o PowerPC e almeno mezzo gigabyte di memoria RAM ma all'epoca anche un Commodore64 sarebbe stato miracoloso per quei tempi. Quindi la complessità dell'algoritmo e i requisiti del calcolatore sul quale avrebbe dovuto girare erano fondamentali, ma sicuramente è stato meglio così perché ha portato alla creazione di algoritmi super-ottimizzati ed essenziali, dei veri capolavori d'intelletto, la potenza del cervello



contro le avversità di un'elettronica che non era certo al passo con i tempi di oggi, ma che comunque è stata una tappa fondamentale per tutta la tecnologia di cui godiamo.

Fu grazie al matematico polacco Jan Lukasewicz che si poté implementare un algoritmo più efficiente per i calcolatori elettronici che stavano nascendo.

Il suo ragionamento si basa sul fatto che il linguaggio formale della nostra matematica è di tipo infisso, ovvero l'elemento più semplice è composto da variabile/numero [operatore variabile/numero] e ogni elemento è legato l'uno all'altro da raggruppamenti mediante parentesi e l'ordine gerarchico degli operatori.

Lukasewicz invece ha creato un sistema di codifica detto **notazione polacca postfissa** (inversa).

Questa notazione è importante per la sua proprietà che semplifica drammaticamente il calcolo di un'espressione. L'algoritmo in grado di processare e calcolare un'espressione codificata in notazione polacca postfissa necessita solo di comprendere il funzionamento di uno stack (o pila).

Lo stack o pila è una struttura dati dove si possono accumulare dati, salvandoli dentro lo stack appunto e poi questi dati possono essere ripresi tenendo presente che la struttura è di tipo LIFO (Last In First Out) ovvero se io salvo A nello stack, poi B nello stack, e adesso estraggo un numero ottengo B, una successiva lettura mi ritorna A, dopodiché lo stack è vuoto, ma è pronto ad accettare nuovi dati.

Ma torniamo all'espressione precedente: $(A+B*C-E)*B-(C+A)/D$.

Per il momento sappiate che l'equivalente in notazione polacca postfissa è : **ABC*+E-B*CA+D/-** .

Se la osserviamo attentamente notiamo che non conosce le parentesi ma gli operatori hanno una posizione ben precisa rispetto agli operandi, oltretutto hanno anche una certa priorità tra loro.

Per questo viene definita postfissa.

Se passiamo la stringa da sinistra a destra incontriamo A, poi B e C dietro a questi due c'è un segno di moltiplicazione, che indica che si dovrà calcolare il prodotto dei due, poi c'è un +, il risultato BC* va a comporre la variabile temporanea che contiene il risultato parziale e si va a sostituire alla sottostringa BC*, immaginiamo di chiamare Z il risultato di questo prodotto. Ora la nostra stringa assumerebbe la seguente forma **AZ+E-B*CA+D/-**, a questo punto $Y=AZ+$, e la formula diventa **YE-B*CA+D/-** ma YE- lo possiamo calcolare e diventa $X=YE-$, allora l'espressione si semplifica ancora e diventa **XB*CA+D/-**, ma XB* lo possiamo calcolare e lo chiamiamo T, allora l'espressione ora diviene **TCA+D/-**, ora quello che si può calcolare è CA+, che chiamiamo S e otteniamo **TSD/-**, e ci calcoliamo SD/ (che infatti corrisponde a CA+D-, ovvero $[C+A]/D$) il risultato lo chiamiamo P, allora abbiamo TP-, quindi se chiamiamo R il risultato abbiamo che $R=T-P$, e possiamo osservare che la sequenza di conti che abbiamo fatto, se fossimo un computer, sarebbe quella che ci porta al risultato dell'espressione iniziale. Si può infine notare che le priorità delle operazioni vengono rispettate ma senza bisogno di parentesi, né di strutture ad albero.

E l'algoritmo di Lukasewicz non necessita di tutte le considerazioni che abbiamo fatto noi e non abbisogna di alcuna delle sostituzioni che abbiamo fatto, porta velocemente al risultato di qualsiasi espressione.

C'è però da notare che un simile approccio, che il processo che porta al calcolo di una espressione è costituito da parti distinte.

La prima si occupa della conversione dalla forma infissa (quella classica che usiamo sempre) a quella postfissa, e questo lo vedremo subito; l'altra parte si occupa invece di calcolare il risultato della forma postfissa.

Conversione da forma infissa a forma posfissa

Tutte le strutture microprogrammate, dalle calcolatrici tascabili ai supercomputer usano un sistema di questo tipo, ad esempio un compilatore (che come nel caso del nostro programmino incontra un'espressione) può facilmente generare il codice che computa qualsiasi delle nostre espressioni nel modo seguente:

1. Consideriamo l'elemento successivo dell'operazione inversa.
2. Se è vuoto, passa al punto 6.
3. Se è una variabile la mettiamo nello stack.
4. Se è un operatore, allora: togliamo le prime due variabili dallo stack, generiamo il codice macchina per la generazione del risultato, calcoliamo il risultato e lo spingiamo nello stack.
5. Torniamo al passo 1.
6. Preleviamo il valore dallo stack, questo è il risultato della nostra espressione.

Come si può intuire l'utilizzo di uno stack è di gran lunga più semplice della gestione di un albero binario.

Il nostro problema ora dovrebbe essere quindi la conversione di una qualsivoglia espressione algebrica che immaginiamo di avere contenuta in una stringa di caratteri INF in una stringa di destinazione POST che dovrà contenere l'equivalente espressione ma in notazione polacca postfissa (inversa).

Calcolare il risultato di una formula espressa in notazione polacca postfissa (inversa).

Notiamo che una lista dei nodi dell'albero binario per rappresentazione infissa, secondo un ordinamento in postordine, porta ad una rappresentazione inversa.

Fortunatamente, esiste un algoritmo per generare la forma inversa senza prima ottenere la rappresentazione in albero binario.

C'è innanzitutto una gerarchia legata agli operatori.

Operatore	Gerarchia
^ elevamento a potenza	4
* moltiplicazione	3
/ divisione	3
+ addizione	2
- sottrazione	2
(parentesi	1

La stringa INF viene scandita da sinistra verso destra, un simbolo alla volta, e la stringa polacca inversa POST è così generata:

1. Consideriamo il prossimo simbolo della stringa INF.
2. Se è vuoto, allora togliamo uno alla volta i simboli dallo stack e li aggiungiamo a POST fino a che lo stack è vuoto, e poi terminiamo.
3. Se è una variabile, allora la aggiungiamo alla stringa POST.
4. Se è una (la mettiamo nello stack.
5. Se è una) togliamo uno alla volta i simboli dallo stack e li aggiungiamo alla stringa POST finché da stack non esce una (.
6. Se è un operatore (operatore ultimo corrente) togliamo uno dopo l'altro i simboli dallo stack e li aggiungiamo a POST finché lo stack non è vuoto oppure in cima allo stack (ma non va tolto da stack) c'è un simbolo (operatore) che ha un valore gerarchicamente minore del nuovo operatore, quello appena letto (ultimo corrente), poi inseriamo quest'ultimo in stack.
7. Torniamo al passo 1.

Ecco gli algoritmi originali del grande matematico polacco Lukasewicz, forse mentre scopriva queste cose non avrebbe immaginato il grande successo che avrebbero avuto, anche se il suo nome non è tra i più ricordati comunque resta una scoperta importantissima nella nuova disciplina che era il calcolo algoritmico.

L'implementazione dell'algoritmo contenuto in questo articolo.

Il programma che questo articolo contiene, deve tutto ad un file header di libreria in C++ che ho chiamato tPolacca.h.

In questa libreria sono contenute le seguenti classi:

1. ope
2. value
3. variable
4. symbol
5. varTable
6. InfixMaker
7. PostfixMaker
8. Calc
9. Expression

La prima definisce un operatore, la propria priorità, il numero di argomenti che usa ed una eventuale routine scritta in C (o in C++) che esegue le funzioni richieste.

Una tabella di operatori viene così, predefinita:

```
const ope _O[] =
{
    ope("^",4,fpow,2), // La potenza ha la gerarchia maggiore
    ope("*",3,fmul,2), ope("/",3,fdiv,2),
    ope("+",2,fadd,2), ope("-",2,fsub,2),
    ope(")",1,NFUNC), // la parentesi è trattata dall'algoritmo!
    ope("(",1,NFUNC),
    ope("ln(",1,flog),
    ope("exp(",1,fexp),
    ope("sen(",1,fsen),
    ope("cos(",1,fcos),
    ope("tan(",1,ftan),
    ope("arctan(",1,farctan),
    ope("sqrt(",1,fsqrt),
    ope("",-1,0)
};
```

Abbiamo quindi detto al nostro programma che le espressioni che potrà trattare avranno gli operatori (e le funzioni)

^, *, /, +, -,), (, ln(), exp(), sen(), cos(), tan(), arctan(), sqrt().

La **classe value** incorpora un valore che la funzione di calcolo utilizzerà nei propri calcoli, tipicamente conterrà le costanti.

Ecco un esempio di definizione della classe value: value a(10),a2(a);

I costruttori chiedono un double o un altro value (costruttore di copia).

```
class value
{
    public:
    value(const double& x);
    value(const value& v);
    virtual double get() const;
    virtual void set(const double& x);
    virtual std::string getName() const;
    virtual value& operator=(const value& rv);
    virtual operator double() const;
    friend std::ostream& operator<<(std::ostream& out, const value& V);
};
```

Un metodo get() restituisce il valore e set(double) lo imposta.

Le altre funzioni membro virtuali servono quando andiamo ad introdurre variable, che è una classe che deriva direttamente da value.

La classe variable è appunto un value dotato di un nome.

```
class variable : public value
{
    public:
    variable();
    template<typename S>
    variable(const S& Name,const double& x);
    template<typename S>
    variable(const S& Name,const value& x);
    variable(const variable& v);
    bool isLinkedVar() const;
    virtual std::string getName() const;
    variable& operator=(const value& rv);
    variable& operator=(const variable& rv);
    virtual double get() const;
    virtual void set(const double& x);
    virtual operator double() const;
    friend std::ostream& operator << (std::ostream& out, const variable& V);
};
```

Esempi di dichiarazioni valide sono variable a(),b("pippo",25),b2(b);

Il primo definisce una variabile non ancora assegnata, la seconda definisce pippo al valore 25 e la terza copia pippo.

Il metodo isLinked ci dice se una variabile è di prima istanza o se invece è una copia.

Il metodo getName() ritorna il nome della variabile, get() il suo valore e set(double) lo imposta.

La **classe varTable** è una tabella di variabili utilizzate in una espressione.

```
class varTable
{
    public:
    varTable();
    varTable(varTable& VT);
    ~varTable();
    // inserisce una nuova var, se ritorna 1 c'era già
    int Var(variable* v);
    int Var(const std::string& name,const double& Value);
    // cerca una variabile
    variable* search(const std::string& name);
    // crea una lista con i nomi delle variabili in uso
    unsigned int varList(std::list<std::string>& vl);
    friend std::ostream& operator << (std::ostream& out, const varTable& V);
};
```

Ecco un esempio di come si può inizializzare una varTable

```
// varTable V;
// V.Var(new variable("a",20));
// V.Var(new variable("b",20));
// V.Var(new variable("c",30));
// cout << "\nVARTABLE:\n" << V << endl;
```

I metodi Var inseriscono una variabile nell'elenco oppure ne aggiornano il suo valore.

Il metodo search restituisce un puntatore ad una variabile se viene trovata oppure ritorna zero.

Il metodo varList crea una lista di stringhe contenenti i nomi delle variabili in uso.

La **classe symbol** è il minimo elemento della notazione sia infissa che postfissa, rappresenta o un operatore, o un valore o una variabile.

Sia la notazione infissa che quella postfissa si esprimono come una lista di questi simboli.

L'uso di questa classe è esclusivo di InfixMaker e PostfixMaker, e non è interessante come le altre classi, si limita solamente a dare un'interfaccia comune agli operatori, ai valori o alle variabili, in modo che si riesca a realizzare l'algoritmo che da una espressione crei una notazione infissa (come lista di simboli), l'algoritmo che data una lista di simboli in notazione infissa crei una lista di simboli in notazione postfissa e successivamente l'algoritmo che leggendo una lista di simboli in notazione polacca postfissa riesca a determinare il risultato di qualunque espressione.

Si costruisce passando o un operatore, o un valore o una variabile e fra i metodi ricordiamo solo:

```
// ritorna il puntatore all'operatore contenuto dal simbolo
const ope* getOperatorPtr() const;
// ritorna il tipo di simbolo
int SymbolType() const;
// controlla il tipo di simbolo
bool isOperator() const { return smbType==T_OPERATOR; }
bool isValue() const { return smbType==T_VALUE; }
bool isVariable() const { return smbType==T_ASCIIIMB; }
virtual operator double() const;
virtual double get() const;
```

La classe InfixMaker

Può semplicemente convertire qualsiasi espressione in forma infissa di simboli.

Ad esempio le istruzioni:

```
InfixMaker M("-1+2.34567e4+exp((-3.2+2.0))");  
cout << "\nInfix\n" << M << endl;
```

generano il seguente risultato:

```
0  
-  
1  
+  
23456.7  
+  
exp(  
(  
0  
-  
3.2  
+  
2  
)  
)
```

Sono la sequenza di operazioni infisse, ovvero nella forma con le quali le persone normalmente hanno a che fare, sono in pratica l'espressione con la quale noi normalmente siamo abituati a trattare.

```
class InfixMaker  
{  
    typedef std::list<symbol*>::iterator iterator;  
    typedef std::list<symbol*>::const_iterator const_iterator;  
    public:  
        InfixMaker(const char* origin,varTable* Vt=0);  
        ~InfixMaker();  
        iterator begin();  
        iterator end();  
        const_iterator begin() const;  
        const_iterator end() const;  
        int size() const;  
        varTable* VTableAddress() const;  
        friend std::ostream& operator << (std::ostream& out, const InfixMaker& X);  
};
```

Il costruttore converte immediatamente l'espressione in lista di simboli.

I metodi begin() ed end() sono iteratori a questa lista di simboli, una volta ottenuta una istanza ad InfixMaker elencare i simboli è semplice:

```
InfixMaker M("a+b*c-d");  
for(InfixMaker::iterator it=M.begin();it!=M.end();++it)  
{  
    cout << **it << endl;  
}
```

Il metodo size() ritorna il numero di simboli utilizzati per l'espressione.

Il metodo VTableAddress() restituisce un puntatore alla varTable utilizzata.

La classe PostfixMaker

```
class PostfixMaker
{
public:
typedef std::list<symbol*>::iterator iterator;
typedef std::list<symbol*>::const_iterator const_iterator;
    PostfixMaker(const char* origin,varTable* Vt=0);
    ~PostfixMaker();
    inline varTable* VTableAddress() const;
    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;
    int error() const;
    friend std::ostream& operator << (std::ostream& out, PostfixMaker& X);
};
```

Questa classe accetta una stringa con una espressione e la converte in notazione polacca postfissa. Ad esempio il seguente codice:

```
PostfixMaker M("a+b*c-d");
cout << M << endl;
/* oppure
for(PostfixMaker::iterator it=M.begin();it!=M.end();++it)
{
    cout << **it << endl;
}
*/
```

Converte l'espressione e la mostra a video.

Il metodo VTableAddress() ritorna, come per InfixMaker, il puntatore alla varTable, che consente quindi di modificare il valore di ogni variabile.

Il metodo error va interrogato alla fine delle operazioni per vedere se c'è un errore (se è zero, nessun errore è presente).

La classe Calc è utilizzata dalla classe expression e serve per convertire una lista di simboli in notazione polacca postfissa in calcoli fino ad arrivare al risultato finito di una espressione.

Realizza in pratica il secondo algoritmo che abbiamo enunciato prima, per la soluzione di espressioni in notazione polacca postfissa.

Consiglio a chi è interessato di esaminare il listato sorgente fornito con questo articolo per vedere l'implementazione.

La classe expression

Questa classe, che utilizza le altre descritte prima, è la soluzione finale, chiavi in mano, per la soluzione run-time di espressioni algebriche.

```
class expression
{
    public:
        expression(const char* ExpressionText);
        expression(compiledTimeFunc CompiledTimeFunction);
        ~expression();
        void assign(const variable& Va);
        template<typename T,typename S>
        void assign(const T& VarName,const S& Value);
        void assing(const double x);
        int error() const;
        double execute();
        double execute(const double x) const;
        varTable* getVarTable() const;
};
```

Consente di immettere a tempo d'esecuzione del programma un'espressione contenuta in un'array di caratteri, quindi è possibile inizializzare le variabili con **assign** ed ottenere il risultato con **execute**.

Ecco un esempio per il calcolo del capitale dopo un anno ad un tasso fisso d'interesse:

```
expression C("C0*exp(n*t)");
C.assign("c0",1000.0);
C.assign("n",0.05);
C.assign("t",1.0);
varTable* V=C.getVarTable();
variable* C0=V->search("C0");
variable* t=V->search("t");
variable* n=V->search("N");
if (C0 && t && n)
{
    cout << "Il capitale " << *C0
    << " dopo anni: " << *n
    << " al tasso fisso " << *t
    << " vale : " << C.execute() << endl;
}
```

Se ci fosse interessato solo il risultato, avremmo potuto invocare il metodo execute() subito dopo l'ultimo assegnamento di variabile.

E' bene controllare il valore restituito dal metodo search in quanto se la variabile non viene trovata, questo ritorna zero, il nostro programma potrebbe "impazzire". Al fine di ridurre le possibilità di errore, automaticamente il programma trasforma tutti i nomi di variabile con caratteri minuscoli. Al fine di rendere questa classe flessibile ed utilizzabile in altri contesti, è perfino possibile istanziare la classe expression con una funzione scritta a livello di codice sorgente (dove il nostro compilatore C++ ha applicato la notazione polacca postfissa per noi, e lo avrà fatto anche meglio del nostro piccolo esempio introduttivo), definiamo una funzione:

```
double miaFunc(double x) // queste sono funzioni di una sola variabile!
{
    return std::sin(x);
}
```

e richiamiamo così un'istanza di expression

```
expression C( miaFunc );
cout << "Il valore della funzione per x=1.57 e' : " << C.execute(1.57) << endl;
```

Il programma di test

Il programma di test, che è possibile lanciare dopo la compilazione del progetto, chiede d'inserire un'espressione, poi chiede d'inserire le variabili, quindi calcola il risultato.

Ecco:

```
#include <iostream>
#include <cassert>
#include "tPolacca.h"
using namespace std;
bool varChange(expression& p , bool force=false)
{
    list<string> v;
    p.getVarTable()->varList(v);
    if (v.size())
    {
        bool wchange=false;
        if (!force)
        {
            cout << "Desideri modificare le variabili ? (y/n) ";
            char c;
            cin >> c;
            cin.ignore();
            if (c=='y' || c=='Y') wchange=true;
        }
        if (wchange || force)
        {
            for(list<string>::iterator it=v.begin(); it!=v.end(); ++it)
            {
                cout << *it << " = ";
                double val;
                cin >> val;
                p.assign(*it, val);
            }
            cin.ignore();
            return true;
        }
    } else
    {
        cout << "L'espressione non ha variabili ma solo costanti\n";
    }
    return false;
}

int main()
{
    cout << "TEST per il calcolo di espressioni mediante\n";
    cout << "la conversione in notazione polacca postfissa\n";

    while(1)
    {
        cout << "Inserisci una espressione, es. C0*exp(n*t)\n";
        cout << "INVIO per terminare\n";
        string t_exp;
        getline(cin, t_exp);
        if (!t_exp.size()) break;
        expression E(t_exp.c_str());
        assert(E.error()==0);
        varChange(E, true);
        do
        {
            cout << "con\n" << *E.getVarTable() << endl;
            cout << "il risultato e' : " << E.execute() << endl;
        } while (varChange(E));
    };
    return 0;
}
```

L'importanza di questi algoritmi

A parte il mio esempio, che sicuramente non passerà alla storia come la più bella implementazione di notazione polacca posfissa, è importante capire cosa ci sta dietro e perché sono così importanti. E' semplicissimo scrivere un programma che calcola il valore di una data espressione, è talmente semplice che noi nemmeno ci accorgiamo di tutto il lavoro che il calcolatore svolge per noi, e non riusciamo a comprenderne l'importanza.

Ecco un velocissimo esempio:

```
// PROGRAMMA x CALCOLO del CAPITALE
#include <iostream>
#include <cmath>
using namespace std;

double Capital( double C0 , double n , double t )
{
    return C0*exp(n*t);
}

int main()
{
    double C0,n,t;
    cout << "Programma per il calcolo del capitale.\n\n";
    cout << "Inserisci il capitale iniziale ? ";
    cin >> C0;
    cout << "Inserisci il numero di anni ? ";
    cin >> n;
    cout << "Inserisci il tasso % annuo ? ";
    cin >> t;
    cout << "\nIl capitale rivalutato diviene: "
    << Capital(C0,n,t);
    cout << "\n\nPremi il tasto INVIO per terminare...";
    std::cin.ignore();
    std::cin.get();
    return 0;
}
```

Dieci righe ed il programma è fatto, ora bisogna compilarlo con un compilatore C++ e poi lanciarlo e vedere che effettivamente svolge il lavoro per il quale lo abbiamo creato.

Cosa è successo? E' successo che attraverso il linguaggio di programmazione C++ ed un compilatore abbiamo istruito il calcolatore a svolgere un compito che avevamo in mente per lui. Il file chiamato capitale.cpp che contiene il testo che si vede sopra (il codice sorgente) dopo la compilazione diventa un eseguibile (chiamato capitale o capitale.exe a seconda del sistema operativo sul quale programiamo) e ogni volta che lo eseguiamo si comporterà sempre nello stesso modo, svolgendo le istruzioni che gli abbiamo dato: leggi dei valori e applica su questi una **formula assegnata** (ovvero **nota al compilatore al momento della creazione del file eseguibile**). Capire come ha fatto il compilatore, partendo da un testo, come il codice sorgente a interpretare e calcolare la formula che noi gli abbiamo assegnato ci aiuta a sviluppare dei sistemi nei quali non sarà più indispensabile fornire a tempo di compilazione le formule su cui dovrà cimentarsi con i calcoli, ma potremo, con le tecniche che già utilizza (senza che noi ne avessimo consapevolezza), definire alla bisogna (**nel tempo in cui il programma va in esecuzione, detto run-time**) formule per le quali c'interessa conoscere i risultati, magari memorizzarle su disco e aprire quelle che servono e quando servono, insomma riusciamo a superare il limite fondamentale del programmino di dieci righe per il calcolo del capitale. Questo limite, risiede nel fatto, che se decidiamo di

utilizzare un'altra formula, non possiamo più usare il programma eseguibile ma è necessario aggiornare il codice sorgente, quindi ricompilare il codice e lanciare un nuovo eseguibile. Ovviamente, l'esecuzione run-time, delle nostre espressioni, richiede un costo computazionale maggiore, dato che l'espressione deve essere analizzata non appena la presentiamo al nostro computer, mentre invece nel compilato, viene analizzata una sola volta e quando il programma è stato lanciato, il calcolatore sa già benissimo come si deve comportare, ma in più di una occasione il costo computazionale è ben compensato dalla possibilità di poter inserire e calcolare funzioni quando lo decidiamo noi, nei nostri programmi.

Download

Ci sono due versioni, una per Unix/Linux, l'altra per Microsoft Windows.

Entrambe comprendono il codice sorgente.

La prima s'installa con:

```
./configure  
make  
make install
```

L'altra invece ha un setup.

Scarica la versione per Unix / Linux (GNU automake)

Scarica la versione per Microsoft Windows.

Grazie per la cortese attenzione,

Paolo Ferraresi.

Per commenti o richieste

e-mail: micro-byte@mondohobby.it