

```
1 #include "expr_int.h"
2 #include "expr_float.h"
3 #include <conio.h>
4 #include "stack.h"
5
6
7 using namespace std;
8
9 int main(){
10
11     cout << "Expresion Calculator 2000" << endl
12         << "choose precision([int]->0, [float]->1): \n>> ";
13     int c;
14     c=getch();
15
16     cout << endl << endl;
17
18     if(c-48){
19         expr_float();
20     }
21     else{
22         expr_int();
23     }
24
25     return 0;
26 }
27
```

```

1 #ifndef OPERAND_H
2 #define OPERAND_H
3
4 class operand{
5 public:
6     int p; char o;
7     operand(){}
8     operand(const int pre, const char opd):
9         p(pre), o(opd){}
10    operand& operator()(const int pre, const char opd){
11        p=pre; o=opd;
12        return *this;
13    }
14    bool operator<(const operand& opd){
15        return p<opd.p;
16    }
17    bool operator>(const operand& opd){
18        return p>opd.p;
19    }
20    bool operator<=(const operand& opd){
21        return p<=opd.p;
22    }
23    bool operator>=(const operand& opd){
24        return p>=opd.p;
25    }
26    bool operator&&(const operand& opd){
27        return p && opd.p;
28    }
29    bool operator&&(const int& opd){
30        return p && opd;
31    }
32    bool operator||(const int& opd){
33        return p || opd;
34    }
35    bool operator||(const operand& opd){
36        return p||opd.p;
37    }
38    bool operator!(){
39        return !p;
40    }
41    bool operator==(const operand& opd){
42        return p==opd.p;
43    }
44    bool operator!=(const operand& opd){
45        return p!=opd.p;
46    }
47    bool operator==(const int& m){
48        return p==m;
49    }
50    bool operator!=(const int& m){
51        return p!=m;
52    }
53    bool operator>(const int& m){
54        return p>m;
55    }
56    bool operator<(const int& m){
57        return p<m;
58    }
59    bool operator<=(const int& m){
60        return p<=m;
61    }
62    bool operator>=(const int& m){
63        return p>=m;
64    }
65    operand& operator=(const operand& opd){
66        p=opd.p;
67        o=opd.o;
68        return *this;

```

```
69     }  
70  
71 };  
72 #endif
```

```

1 #ifndef STACK_H
2 #define STACK_H
3
4 template <class t> class stack;
5
6 template <class u>
7 class node{
8 public:
9     node():next(nullptr){}
10    node(u m_var):next(nullptr, var(m_var)){
11 private:
12     u var;
13     node* next;
14     node& operator=(const u& n_var){
15         return this->var = n_var;
16     }
17     friend class stack<u>;
18 };
19
20 template <class t>
21 class stack{
22 private:
23     int m_TOP;
24
25     node<t>* m_current;
26     node<t>* m_start;
27
28 public:
29     // Overloaded Constructors Methods
30     stack(int size = 0){
31         m_TOP=-1;
32         m_current=m_start=nullptr;
33         for(int i(0); i<size; i++)
34             push_back();
35     }
36     stack(int size, const t* array){
37         m_TOP=-1;
38         m_current=m_start=nullptr;
39         for(int i(0); i<size; i++)
40             push_back(t[i]);
41     }
42     stack(int size, const t var){
43         m_TOP=-1;
44         m_current=m_start=nullptr;
45         for(int i(0); i<size; i++)
46             push_back(t);
47     }
48     stack(const stack& obj){
49         m_TOP=-1;
50         m_current=m_start=nullptr;
51
52         for(int i(0); i<=obj.size(); i++){
53             push_back(obj.get(i));
54         }
55     }
56     stack(const stack&& obj){
57         m_TOP=-1;
58         m_current=m_start=nullptr;
59         m_start = obj[0];
60         m_current = obj[obj.size()];
61         obj.m_TOP = -1;
62         obj.m_current = obj.m_start= nullptr;
63         obj.clear();
64     }
65
66     // A Destructor Method
67     ~stack(){
68

```

```

69     clear();
70 }
71
72 // Normal Methods:
73 void push_back(const t& var){
74     m_TOP++;
75     if(m_TOP==0){
76         m_current = new node<t>;
77         m_current->next = nullptr;
78         m_current->var = var;
79         m_start = m_current;
80     } else{
81         m_current->next = new node<t>;
82         m_current = m_current->next;
83         m_current->next = nullptr;
84         m_current->var = var;
85     }
86 }
87
88 void push_back(){
89     m_TOP++;
90     if(m_TOP==0){
91         m_current = new node<t>;
92         m_current->next = nullptr;
93         m_start = m_current;
94     } else{
95         m_current->next = new node<t>;
96         m_current = m_current->next;
97         m_current->next = nullptr;
98     }
99 }
100
101 t pop_out(){
102     t var;
103     if(m_TOP > -1){
104         if(m_TOP==0){
105             var = m_current->var;
106             delete m_current;
107             m_current=m_start=nullptr;
108         }else if(m_TOP==1){
109             var = m_start->next->var;
110             delete m_start->next;
111             m_start->next=nullptr;
112             m_current=m_start;
113         } else{
114             var = m_current->var;
115
116             node<t>* m_del = m_start;
117             for(int i(0); i<(m_TOP-1); i++)
118                 m_del = m_del->next;
119
120             m_current = m_del;
121             delete m_del->next;
122             m_del->next=nullptr;
123         }
124
125         m_TOP--;
126     }
127     return var;
128 }
129
130
131 t get(unsigned int index){
132     t var;
133     node<t>* m_get = m_start;
134     if(index<=m_TOP)
135         for(int i(0); i<index; i++)
136             m_get = m_get->next;

```

```

137         var = m_get->var;
138         return var;
139     }
140     t top(){
141         if(m_TOP > -1){
142             return m_current->var;
143         }
144     }
145     int size(){
146         return m_TOP;
147     }
148     void clear(){
149         if(m_TOP!=-1){
150             node<t>* m_del = m_start;
151             for(int i(0); i<=m_TOP; i++){
152                 m_current = m_del->next;
153                 m_del->next = nullptr;
154                 delete m_del;
155                 m_del = m_current;
156             }
157         }
158         m_TOP = -1;
159         m_current=m_start=nullptr;
160     }
161
162     // Operator methods
163     node<t>* operator[](int index){
164         node<t>* m_get = m_start;
165         if(index<=m_TOP)
166             for(int i(0); i<index; i++){
167                 m_get = m_get->next;
168             }
169         return m_get;
170     }
171     stack& operator=(const stack& obj){
172         clear();
173         for(int i(0); i<=obj.size(); i++){
174             push_back(obj.get(i));
175         }
176         return this;
177     }
178
179
180
181 };
182
183
184 #endif

```

```

1 #include <iostream>
2 #include "operand.h"
3 #include "stack.h"
4 #include <math.h>
5
6 using namespace std;
7
8 #ifndef expr_int_h
9 #define expr_int_h
10
11 int expr_int(){
12
13     cout << "Enter a Numeric Expression ( May include integers,(),*,/,%,^,-,+ ).";
14     while(true){
15
16         int MAXLEN(200);
17         char* raw(new char[MAXLEN]);           // creating a char Array to
18         cout << "\n[int]> ";                     // store user input
19
20         cin.getline( raw , MAXLEN-1 , '\n' ); // taking input from user
21         if(!strlen(raw,MAXLEN)){               // Quit if no input
22             return 0;
23         }
24
25         stack<int> postfix;                     // creating a int stack
26         stack<operand> opd;                     // creating an operand stack
27
28         opd.push_back(operand(-1,'('));        // Pushing an opening bracket
29
30         bool error(0);                         // an error flag
31
32         /* Following loop converts Expression to
33          * postfix and calculates it: */
34         for( int i=0, iflag(0); i<=strlen(raw) ; ++i ){
35
36             //1. For a Literal
37             if((int)(raw[i])-48 >= 0 && (int)(raw[i])-48 <= 9){
38                 if(iflag){
39                     int a =(int)(raw[i])-48 + postfix.pop_out() * 10;
40                     postfix.push_back(a);
41                 } else{
42                     iflag=1;
43                     int a =(int)(raw[i])-48;
44                     postfix.push_back(a);
45                 }
46             }
47
48             //2. For an Operand
49             else if(raw[i] == '(' || raw[i] == ')' ||
50                    raw[i] == '*' || raw[i] == '/' ||
51                    raw[i] == '%' || raw[i] == '-' ||
52                    raw[i] == '+' || raw[i] == '^' ||
53                    raw[i] == ' ' || raw[i] == '\0'){
54
55                 iflag=0; //
56                 int poco; // Operand priority flag
57
58                 // Sets operand priority flag
59                 switch(raw[i]){
60                     case '+':case '-':poco=1;break;
61                     case '*':case '/':case '%':poco=2;break;
62                     case '^':poco=3;break;
63                     case ')':poco=-2;break;
64                     case '(':poco=-1;break;
65                     default: poco=0;break;
66                 }
67
68                 operand dob(poco,raw[i]); // New Operand type

```

```

69
70 // priority of last operand in stack is smaller
71 if( (dob > 0 && dob >= opd.top()) || dob == -1 ){
72     opd.push_back(dob);
73 }
74
75 // priority of last operand in stack is larger
76 else if( dob > 0 && dob < opd.top()){
77
78     // Gets the last operand in stack
79     operand popped(opd.top().p, opd.top().o);
80
81     // Pop until last operand in stack is of smaller priority
82     while(dob < popped){
83
84         opd.pop_out(); // Delete the last operand
85
86         int b = postfix.get(postfix.size()); // Gets the last two
87         int a = postfix.get(postfix.size()-1); // Numbers form Postfix
88                                             // Stack to work upon
89
90         int r(1); // result variable
91
92         postfix.pop_out(); // Clear the last two
93         postfix.pop_out(); // Number in Postfix
94
95         // Work upon the Numbers
96         switch(popped.o){
97             case '+':r=a+b;break;
98             case '-':r=a-b;break;
99             case '*':r=a*b;break;
100            case '/':r=a/b;break;
101            case '%':r=a%b;break;
102            case '^':for(int i(0); i<b; i++)r*=a;break;
103            default: r=a+b;break;
104        }
105
106        // Push the result back in postfix stack
107        postfix.push_back(r);
108
109        // Get the next operand in stack
110        popped(opd.top().p, opd.top().o);
111    }
112
113    // Now push opernad in stack
114    opd.push_back(dob);
115 }
116
117 // operand is a closing bracket
118 else if(dob == -2 || dob.o == '\0'){
119
120     // Same as above, only that it pops operands
121     // until an opening bracket is found
122     operand popped(opd.top().p, opd.top().o);
123     while(popped != -1){
124         opd.pop_out();
125         int b = postfix.get(postfix.size());
126         int a = postfix.get(postfix.size()-1);
127         int r(1);
128         postfix.pop_out();
129         postfix.pop_out();
130         switch(popped.o){
131             case '+':r=a+b;break;
132             case '-':r=a-b;break;
133             case '*':r=a*b;break;
134             case '/':r=a/b;break;
135             case '%':r=a%b;break;
136             case '^':for(int i(0); i<b; i++)r*=a;break;

```



```

137             default: r=a+b;break;
138         }
139         postfix.push_back(r);
140         popped(opd.top().p, opd.top().o);
141     }
142     opd.pop_out();
143 }
144
145 }
146
147 //3. An Error
148 else{
149     error=1;
150     cout << "-> Invalid String" << endl;
151     break;
152 }
153 }
154
155 if(!error)                // Printing Answer of Expression
156     cout << "=> Answer: " // if No error is present
157         << postfix.top();
158
159 postfix.clear();           //Clearing the stacks for next run
160 opd.clear();
161
162 cout << endl;             //Now Ready for another expression
163
164 }
165 return 0;
166 }
167
168 #endif

```

```

1 #include <iostream>
2 #include "operand.h"
3 #include "stack.h"
4 #include <math.h>
5
6 using namespace std;
7
8 #ifndef expr_float_h
9 #define expr_float_h
10
11 int expr_float(){
12
13     cout << "Enter a Numeric Expression ( May include integers,(),*,/,%,^,-,+ ).";
14     while(true){
15
16         int MAXLEN(200);
17         char* raw(new char[MAXLEN]);           // creating a char Array to
18         cout << "\n[float]> ";                 // store user input
19
20         cin.getline( raw , MAXLEN-1 , '\n' ); // taking input from user
21         if(!strlen(raw,MAXLEN)){              // Quit if no input
22             return 0;
23         }
24
25         stack<double> postfix;                // creating a double stack
26         stack<operand> opd;                   // creating an operand stack
27
28         opd.push_back(operand(-1,'('));       // Pushing an opening bracket
29
30         bool error(0);                        // an error flag
31
32         /* Following loop converts Expression to
33          * postfix and calculates it: */
34         for( int i=0, iflag(0), dflag(0); i<=strlen(raw) ; ++i ){
35
36             //1. For a Literal
37             if((int)(raw[i])-48 >= 0 && (int)(raw[i])-48 <= 9 || raw[i]=='.' ){
38                 if(iflag && !dflag){
39                     if(!(raw[i]=='.')){
40                         double a =(float)((int)(raw[i])-48) + postfix.top() * 10;
41                         postfix.pop_out();
42                         dflag=0;
43                         postfix.push_back(a);
44                     }
45                     else if(raw[i]=='.' ){
46                         dflag=1;
47                     }
48                 }
49             }
50             else if(iflag && dflag){
51                 if(!(raw[i]=='.')){
52                     double a =(float)((int)(raw[i])-48)/(pow(10,dflag++))
53                     + postfix.top();
54                     postfix.pop_out();
55                     postfix.push_back(a);
56                 }
57                 else if(raw[i]=='.' ){
58                     error=1;
59                     cout << "-> Invalid String" << endl;
60                     break;
61                 }
62             }
63             else if(!iflag && !dflag){
64                 iflag=1;
65                 int a =(int)(raw[i])-48;
66                 postfix.push_back(a);
67             }
68             else if(!iflag && dflag){

```

```

69         error=1;
70         cout << "-> Invalid String" << endl;
71         break;
72     }
73 }
74
75 //2. For an Operand
76 else if(raw[i] == '(' || raw[i] == ')' ||
77         raw[i] == '*' || raw[i] == '/' ||
78         raw[i] == '%' || raw[i] == '-' ||
79         raw[i] == '+' || raw[i] == '^' ||
80         raw[i] == ' ' || raw[i] == '\\0'){
81     iflag=0;dfalg=0;
82     int poco;
83     switch(raw[i]){
84         case '+':case '-':poco=1;break;
85         case '*':case '/':case '%':poco=2;break;
86         case '^':poco=3;break;
87         case ')':poco=-2;break;
88         case '(':poco=-1;break;
89         default: poco=0;break;
90     }
91     operand dob(poco,raw[i]);
92     if( (dob > 0 && dob >= opd.top()) || dob == -1 ){
93         opd.push_back(dob);
94     }
95     else if( dob > 0 && dob < opd.top()){
96         operand popped(opd.top().p, opd.top().o);
97         while(dob < popped){
98             opd.pop_out();
99             double b = postfix.get(postfix.size());
100             double a = postfix.get(postfix.size()-1);
101             double r(1);
102             postfix.pop_out();
103             postfix.pop_out();
104             switch(popped.o){
105                 case '+':r=a+b;break;
106                 case '-':r=a-b;break;
107                 case '*':r=a*b;break;
108                 case '/':r=a/b;break;
109                 case '%':r=fmod(a,b);break;
110                 case '^':r=pow(a,b);break;
111                 default: r=a+b;break;
112             }
113             postfix.push_back(r);
114             popped(opd.top().p, opd.top().o);
115         }
116         opd.push_back(dob);
117     }
118     else if(dob == -2 || dob.o == '\\0'){
119         operand popped(opd.top().p, opd.top().o);
120         while(popped != -1){
121             opd.pop_out();
122             double b = postfix.get(postfix.size());
123             double a = postfix.get(postfix.size()-1);
124             double r(1);
125             postfix.pop_out();
126             postfix.pop_out();
127             switch(popped.o){
128                 case '+':r=a+b;break;
129                 case '-':r=a-b;break;
130                 case '*':r=a*b;break;
131                 case '/':r=a/b;break;
132                 case '%':r=fmod(a,b);break;
133                 case '^':r=pow(a,b);break;
134                 default: r=a+b;break;
135             }
136             postfix.push_back(r);

```

```
137         popped(opd.top().p, opd.top().o);
138     }
139     opd.pop_out();
140 }
141
142 }
143 //else
144 else{
145     error=1;
146     cout << "-> Invalid String" << endl;
147     break;
148 }
149 }
150 if(!error)
151     cout << "=> Answer: " << postfix.top();
152 postfix.clear();
153 opd.clear();
154 cout << endl ;
155
156 }
157 return 0;
158 }
159
160 #endif
```