Modern C++ - Expressions

5. Operators and expressions

Expressions, Expression evaluation

Expressions are formed from *operators* and *values* or *variables*. The most simple expressions are similar to the mathematical expressions.

```
A + B * C
```

What is the meaning?

```
A + (B * C)
```

Because the *precedence* of multiplication is higher/stronger than the addition.

What happens, when the operators are on the same precedence level?

```
A * B / C * D
```

In FORTRAN77, it was not defined, what is the meaning of this expression:

```
((A * B) / C) * D
(A * B * D) / C
(A / C) * B * D
```

...and if A, B, C, D are INTEGERs, the final result could be different.

In modern languages, expressions are defined by *precedence* and *associativity rules*.

In some languages, like Java, associativity always *left-to-right*. In C++ associativity is defined by precedence rules. In most precedence groups it is left-to-right, but for *unary*, *ternary* and *assignment* operators they are right-to-left.

Operators in C++

Precedence	Operator	Description	Assoc
Scope	::	scope qualifier	L->R
Postfix	++	postfix increment	L->R
	– –	postfix decrement	
	()	function call	

Precedence	Operator	Description	Assoc
	[]	array index	
	.	struct/union member access	
	->	member access via pointer	
	type()	functional cast	
	type{}	functional cast (C++11)	
Unary	++	prefix increment	R->L
	– –	prefix decrement	
	+	unary plus	
	–	unary minus	
	!	logical negation	
	~	binary negation	
	(type)	type conversion	
	*	pointer indirection	
	&	address of	
	sizeof	size of type/object	
	new new[]	dynamic memory allocation	
	delete delete[]	dynamic memory deallocation	
Member ptr	.* ->*	pointer-to-member	L->R
Multiplicative	* / %	multiplication, division, remainder	L->R
Additive	+ –	addition, substraction	L->R
Shift	« »	bitwise left/right shift	L->R
Relational	< <= > >=	relational operations	L->R
Equality	== !=	equal, not equal	L->R
Bitwise	&	bitwise AND	L->R
	^	bitwise XOR (exclusive OR)	L->R
	|	bitwise OR	L->R
Logical	&&	logical AND	L->R
	||	logical OR	L->R
Ternary	? :	conditional expression	R->L
Throw	throw	throw excepsion	R->L
Assignment	=	assignment	R->L
	+= –=	compound assignment	
	*= /= %=		
	«= »=		

Precedence	Operator	Description	Assoc
	&= |= ^=		
Comma	,	sequence operator	L->R

There are other operators which are never ambigous:

```
const_cast  static_cast  dynamic_cast  reinterpret_cast
typeid  sizeof...  noexcept  alignof
```

There are alternative spellings for boolean operators:

||	or
&&	and
!	not

Evaluation of expressions

Although, expressions are defined by precedence and associativity, *how* the expression is evaluated is implementation defined.

What will print the following program?

```cpp
#include <iostream>
int main()
{
  int i = 1;
  std::cout << "i = " << i << ", ++i = " << ++i << std::endl;
  return 0;
}
```

Surprisingly, the program can print the following:

```
$ ./a.out
i = 2, ++i = 2
```

In fact, some compilers on some platforms can also print:

```
$ ./a.out
i = 1, ++i = 2
```

The *evaluation order* may not be defined for expressions.

Exceptions are those operations which are *sequence points*:

1. shortcut logical operators (&& ||)
2. conditional operator (? :)
3. comma operator

Also, when a function is called, first the operands should have evaluated, (but parameter evaluation happens in undefined order between the parameters).

Look at the following example:

```cpp
 1 #include <iostream>
 2 int f()
 3 {
 4   std::cout << "f" << std::endl;
 5   return 2;
 6 }
 7 int g()
 8 {
 9   std::cout << "g" << std::endl;
10   return 1;
11 }
12 int h()
13 {
14   std::cout << "h" << std::endl;
15   return 0;
16 }
17 void func(int fpar, int gpar, int hpar)
18 {
19   std::cout << "(f() == g() == h()) == " << (fpar == gpar == hpar) << std::endl;
20 }
21 int main()
22 {
23   func(f(),g(),h());
24   return 0;
25 }
```

```
$ g++ -ansi -pedantic -Wall -W op.cpp
op.cpp: In function ‘void func(int, int, int)’:
op.cpp:19:51: warning: suggest parentheses around comparison in operand of ‘==’ [-Wparentheses]
   std::cout << "(f() == g() == h()) == " << (fpar == gpar == hpar) <<
```

```
std::endl;
                                                    ^

gsd@ken:~/tmp$ ./a.out
h
g
f
(f() == g() == h()) == 1
```

In the example the result of the expression **1** is well-defined. This should be the result regardless of compilers and platforms. However, the *order of evaluation* of the expression is not defined and may be dependent of the actual platform, compiler, or even compilation flags (e.g. optimizations).

Common errors

In the following we describe a few common mistakes regarding operator precendence of C++ programs.

Wrong precedence

```
/*
 * LIKELY BAD!
 */
#include <stdio.h>
int main()
{
  int mask = 0x01;
  if ( mask & 0x10 == 16 )
  {
     printf("This is strange!\n");
  }
  printf("mask = %d, 0x10 = %d,  mask & 0x10 = %d,  mask & 0x10  == 16 = %d\n",
          mask,      0x10,       mask & 0x10,       mask & 0x10  == 16);
  return 0;
}
```

```
$ g++ -ansi -pedantic -Wall -W  f.cpp
f.cpp: In function ‘main’:
f.cpp:6:13: warning: suggest parentheses around comparison in operand of ‘&’ [-Wparentheses]
```

```
   if ( mask & 0x10 == 16 )
             ^
f.cpp:11:58: warning: suggest parentheses around comparison in operand of ‘&’
[-Wparentheses]
           mask,      0x10,       mask & 0x10,       mask & 0x10  == 16);
                                                          ^

$ ./a.out
This is strange!
mask = 1, 0x10 = 16,  mask & 0x10 = 0,  mask & 0x10  == 16 = 1
$
```

Bitwise operator precedence is lower than equation relation precedence. Use parentheses to express your intentions.

Correct way

```
/*
 * OK!
 */
#include <stdio.h>
int main()
{
  int mask = 0x01;
  if ( (mask & 0x10) == 16 )
  {
     printf("This is strange!\n");
  }
  printf("mask = %d, 0x10 = %d, (mask & 0x10) = %d, (mask & 0x10) == 16 = %d\n",
          mask,      0x10,       mask & 0x10,       (mask & 0x10) == 16);
  return 0;
}
```

```
$ g++ -ansi -pedantic -Wall -W  f.c
gsd@Kubuntu-e1:~/ftp$ ./a.out
mask = 1, 0x10 = 16, (mask & 0x10) = 0, (mask & 0x10) == 16 = 0
```

Assignment vs equality check

```
/*
 * BAD!
 */
#include <stdio.h>
int main()
{
  int i = 0;
  if ( i = 1 )
  {
    printf("This is strange!\n");
  }
  return 0;
}
```

```
$ g++ -ansi -pedantic -Wall -W  f.c
f.cpp: In function ‘main’:
f.cpp:6:3: warning: suggest parentheses around assignment used as truth value
[-Wparentheses]
  if ( i = 1 )
  ^
$ ./a.out
This is strange!
```

The safe way

If you compare a literal with a value always put the literal on the left side!

```
#include <stdio.h>
int main()
{
  int i = 0;
  if ( 1 = i )
  {
    printf("This is strange!\n");
  }
  return 0;
}
```

```
$ gcc -ansi -pedantic -Wall -W  f.c
f.cpp: In function ‘main’:
f.cpp:6:10: error: lvalue required as left operand of assignment
   if ( 1 = i )
          ^
```

Missing sequence point

```
/*
 * BAD!
 */
#include <stdio.h>
int main()
{
  int t[10];
  int i = 0;
  while( i < 10 )
  {
    t[i] = i++;
  }
  for ( i = 0; i < 10; ++i )
  {
    printf("%d ", t[i]);
  }
  return 0;
}
```

```
$ g++ -ansi -pedantic -Wall -W  f.c
f.cpp: In function ‘main’:
f.cpp:9:13: warning: operation on ‘i’ may be undefined [-Wsequence-point]
   t[i] = i++;
           ^
$ ./a.out
613478496 0 1 2 3 4 5 6 7 8
$
```

Do not try to be too C++-ish! Write your intentions in the most simple way. If you need events happenning is sequential order, use sequence points.

Correct way

```
/*
 * OK
 */
#include <stdio.h>
int main()
{
  int t[10];
  int i = 0;
  while( i < 10 )
  {
    t[i] = i;
    ++i;
  }
  for ( i = 0; i < 10; ++i )
  {
    printf("%d ", t[i]);
  }
  return 0;
}
```