

In C programs, side-effects are common practice:

```
while( (c=getchar()) != EOF ) putchar(c);
```

(in Pascal, such side-effects must be buried in function defs).

In C, the meaning of “expression” is *nonstandard*.

In common usage (i.e, in mathematics and in most programming languages), expressions are **referentially transparent** :

- an expression’s only significance is its value
- any of an expression’s subexpressions can be replaced by an equivalent subexpression with no effect on value of the expression
- expression evaluation causes no side-effects

In C, however, “expressions” are best understood as sequences of operations (\Rightarrow loss of abstraction).

“Expressions” having side-effects interact with another “feature”: Except for the *four* operators

```
&& || ?: ,
```

the order of evaluation of operators’ operands—and of functions’ arguments—is **not defined** in the language specification; it is *compiler-dependent*.

Example:

```
int a = 0, b = (a = 5) + a;
```

If `(a = 5)` is evaluated first, `b` is initialized to 10;
if `a` is evaluated first, `b` is initialized to 5.

Simulating the evaluation of expressions containing side-effect operators:

- `int n = 3; int x = ++n * n++ - --n;`

Assuming **left-to-right** evaluation:

```
++n * n++ - --n
 $\rightsquigarrow$  (++n * n++) - --n where n = 3
 $\rightsquigarrow$  (4 * n++) - --n where n = 4
 $\rightsquigarrow$  (4 * 4) - --n where n = 5
 $\rightsquigarrow$  16 - --n where n = 5
 $\rightsquigarrow$  16 - 4 where n = 4
 $\rightsquigarrow$  12 where n = 4
```

Assuming **right-to-left** evaluation:

```
++n * n++ - --n
 $\rightsquigarrow$  (++n * n++) - --n where n = 3
 $\rightsquigarrow$  (++n * n++) - 2 where n = 2
 $\rightsquigarrow$  (++n * 2) - 2 where n = 3
 $\rightsquigarrow$  (4 * 2) - 2 where n = 4
 $\rightsquigarrow$  8 - 2 where n = 4
 $\rightsquigarrow$  6 where n = 4
```

- `int n = 2; n = (n = 4) * n;`

Assuming **left-to-right** evaluation:

```
n = (n = 4) * n
 $\rightsquigarrow$  n = (n = 4) * n where n = 2
 $\rightsquigarrow$  n = 4 * n where n = 4
 $\rightsquigarrow$  n = 4 * 4 where n = 4
 $\rightsquigarrow$  n = 16 where n = 4
 $\rightsquigarrow$  16 where n = 16
```

Assuming **right-to-left** evaluation:

```
n = (n = 4) * n
 $\rightsquigarrow$  n = (n = 4) * n where n = 2
 $\rightsquigarrow$  n = (n = 4) * 2 where n = 2
 $\rightsquigarrow$  n = 4 * 2 where n = 4
 $\rightsquigarrow$  n = 8 where n = 4
 $\rightsquigarrow$  8 where n = 8
```

A more realistic example (C++):

```
Stack s; ... s.push( 1 ); s.push( 2 ); ...
cout << s.pop() << s.pop() << endl;
```

may produce either

```
1 2
```

or

```
2 1
```

depending on the compiler/platform.

Type definitions

Pascal:

```
type <name> = <type>
```

examples:

```
type dir = (north, east, south, west)
```

```
type intF = function( char ) : integer
```

C:

```
typedef <type> <name>
```

examples:

```
typedef enum{north,east,south,west} dir;
```

```
enum dir {north,east,south,west};
```

```
typedef int intF( char );
```

after which these are equivalent:

```
void f( int g( char ) ){...}
```

```
void f( intF g ){...}
```

Haskell:

```
type <name> = <type>
```

examples:

```
type String = [Char];
```

```
type IntF = Char->Int;
```

(C’s **typedef** and Haskell’s **type** don’t define new types— they define synonyms for existing types)

Cartesian products

Pascal:

type constructor:

```
record { <names> : <type> ; }+ end
<names> ::= <name> | <name> , <names>
```

value construction:

assignment to individual fields

value selector:

```
<record designer> . <name>
```

C:

type constructor:

```
struct [ <name> ] 'f' { (type) <name> ; }+ 'f'
```

value constructor:

```
'f' { expression ; }+ 'f'
```

also assignment to individual fields

value selector:

```
<record designer> . <name>
```

Haskell:

- *anonymous* product types: **tuples**

type constructor: (<type> { , <type> })

value constructor: (<expr> { , <expr> })

example:

```
( True, "grit" ) :: (Bool,String)
```

- *named* product type: **algebraic types**
a generalization of *enumeration*

Algebraic types

This facility enables the Haskell programmer to define types that are truly new— not mere synonyms for existing types.

The simplest kind of algebraic type is the enumerated type described above.

Algebraic types can also be used to define **product types**. These are in effect labeled *n*-tuples, and can often be used in place of tuples.

Example:

```
> data Time = HMS Int Int Float
      HMS           constructor
      Int, Int, Float fields' types
                (hours, minutes, seconds)
      HMS :: Int -> Int -> Float -> Time
```

A constructor is like a function, but it doesn't compute anything; it merely "packages" its arguments.

Functions taking *Time* arguments can use the constructor in pattern-matching to "extract" the components:

```
> toSeconds :: Time -> Float
> toSeconds (HMS h m s)
  = fromInt (60 * (60 * h + m)) + s
```

so

```
Main> toSeconds (HMS 3 4 2.1)
11042.1
```

Constructors are unique, and signal their values' types.

Often a problem can be solved using either an algebraic type or a tuple type.

An alternative definition of *Time* (as a synonym for a tuple type):

```
> type Time = (Int, Int, Float)
```

algebraic types' advantages :

- constructor makes code clearer
- constructor prevents mistaking some other (Int,Int,Float) combination for a *Time* value
- Hugs error messages may lose synonym names, but always preserve algebraic type names

tuples' advantages :

- more compact
- allow use of polymorphic functions such as *fst* and *zip*.
- need no type definition
- better for "on-the-fly" packaging

Algebraic types can be used to distinguish between distinct uses of the same type.

```
> data Force = Newtons Float | Pounds Float
```

—prevents mistaken-identity accidents (e.g., Mars Polar Lander).

Sum-of-products algebraic types (a generalization of enumeration and product types)

```
> data Temp = Celsius Float | Fahrenheit Float
```

—multiple constructors, each with argument(s).

These can be viewed as

- alternative implementations of the type
- subtypes

Functions use pattern-matching to determine to which subtype an argument belongs.

Example:

```
> freezing :: Temp -> Bool
> freezing (Celsius t) = t <= 0.0
> freezing (Fahrenheit t) = t <= 32.0
```

Constructors' arities can differ, as in

```
> data Shape = Circ Float | Rect Float Float
```

Two instances of *Shape*:

```
Circ 3.0      Rect 45.9 87.6
```

Some functions on *Shape*:

```
> isRound :: Shape -> Bool
> isRound (Circ _) = True
> isRound (Rect _ _) = False
```

```
> area :: Shape -> Float
> area (Circ r) = pi * r * r
> area (Rect h w) = h * w
```