Subprograms, parameter passing

separation of concern: a subprogram has a single role (theoretically)
level of abstraction - especially for
- functional programming paradigm
- procedural paradigm
also a unit of implemenation (Fortran, C-like)
reusable component

joke: do not make a function called "nuke_budapest"!!!

Originally: function: abstracting an expression
procedure: abstracting a comman or "action"
Today: mixed (side effects, "void" return type, etc.)

abstraction: make something more general than the concrete usage:
generalization mechanism:
- parameter passing
- overloading

Parameter passing
==========

formal parameter(s): in the function definition: double sin(double x)
actual parameter(s)/argument(s): in the function call: sin(.5)

by address (cím szerint)
by value (érték szerint)
by result (eredmény szerint)
by name (név szerint)

by address
==========

The formal parameter and the actual argument is the same memory location

FORTRAN, Pascal, C++ reference par, Java ref types.

Advantages:
- simple, cheap
- no copy happens on parameter passing
- works for output parameters
Issues:
- the effect inside teh function body is
not separated from the outer environment
- how to pass non-variables (expressions, constants)

by value
========

The formal parameter acts like local variable initialized by the arguments
  (usually by copy)

Pascal var, C, C++, Java built-in

Advantages:
 - easier to understand
 - better separation from the environment
 - not too complex to implement
Issues:
 - not for output Fixes: return value, or use pointers
 - copy might be expensive: Fixes: move semantics or passing (const) reference

by result
=========

The same as by value, but returning from function
  the output parameter replaces the actual argument

Ada, ?

Advantages:
 - like for by value
Issues:
 - how to pass non-variables (expressions, constants)

by name
=======

Passing a "closure", not the value

Algol 60, Simula 67, Algol 68

Advantages:
 - very scientific
Issues:
 - hard and expensive to implement by "closue"
 - sometimes hard to understand

Overloading
==========

different functions with the same name (but with different signatures)
sometimes called as "named polymorphism"

```
int    pow(int, int)       i^j
double pow(double, double)

f( int, int)
f( int)

typdef int length_t;
using lenght_t = int;

int    max(int i, int j);
double max(double x, double y);

double z = max(f(max( 1, 3.14)), 1);
std::cout << max(1, 3.14)

f(length_t)

f( 0, 3.14, "Hello")

void f(int,  int)
void f(long, float)

 f(1,3.14)

template <typename T>
class std::vector
{
  iterator       begin();         --->   __begin( vector *this)
  const_iterator begin() const;   --->   __begin( const vector *this)
}
vector<int> v;         v.begin()    ---> __begin(&v)
const vector<int> cv; cv.begin()    ---> __begin(&cv)
```

Recursion
=========

```
int fact(int n)
{
 if ( i == 1 )
   return 1;
 else
   return n*fact(n-1);   // tail recursion
}
```